DOCA Documentation v3.5.0

Table of Contents

1 DOCA Framework 95
1.1 Introduction 95
1.2 DOCA Packages 95
1.2.1 DOCA-Host 95
1.2.2 BlueField-Bundle (BF-Bundle) 96
1.2.3 BlueField-Firmware Bundle (BF-FWBundle) 96
1.3 BlueField Devices 96
2 DOCA SDK 97
2.1 DOCA Overview 97
2.1.1 Introduction 97
2.1.2 Installation 101
2.1.3 API 101
2.1.4 Programming Guides 101
2.1.5 Applications 102
2.1.6 Tools 102
2.1.7 Services 102
2.2 DOCA Release Notes 102
2.2.1 Introduction 103
2.2.2 Installation Notes 103
2.2.3 Supported Device Speeds 103
2.2.4 Technical Support 104
2.2.5 Changes and New Features 105
2.2.5.1 1. DOCA Framework New Features and Updates 105
2.2.5.2 2. DOCA-Host and DOCA Drivers New Features and Updates 106
2.2.5.3 3. BlueField-3 Firmware Changes and New Features 106
2.2.5.4 4. BlueField-2 Firmware Changes and New Features 106
2.2.5.5 5. API Changes 106
2.2.5.6 Customer Affecting Changes 107
2.2.5.6.1 Changes in This Release 107
2.2.5.6.2 Changes Planned for Future Releases 108
2.2.5.6.3 Changes from Earlier Releases 113
2.2.5.6.4 Discontinued Features 118
2.2.5.7 Declared Unsupported Features 118
2.2.6 General Support 118

2.2.6.1 Supported Platforms and NIC Firmware Versions 118
2.2.6.2 Embedded DOCA BlueField System Components 119
2.2.6.3 Embedded DOCA Drivers 119
2.2.6.4 CUDA Compatibility with DOCA 121
2.2.6.5 DOCA Packages 121
2.2.6.6 BlueField Operating System 122
2.2.6.7 Supported Host OS per DOCA-Host Installation Profile 123
2.2.6.8 Storage Supported DOCA-Host Operating Systems 126
2.2.6.9 DOCA-Host Version Interoperability 127
2.2.6.10 Supported Devices 127
2.2.7 Bug Fixes in This Version 127
2.2.7.1 DOCA Framework Bug Fixes 127
2.2.7.2 DOCA-Host and DOCA Drivers Bug Fixes 134
2.2.7.3 BSP Bug Fixes 135
2.2.7.4 BMC Bug Fixes 137
2.2.7.5 BlueField-3 Firmware Bug Fixes 137
2.2.7.6 BlueField-2 Firmware Bug Fixes 140
2.2.8 Known Issues 141
2.2.8.1 DOCA Framework Known Issues 141
2.2.8.2 DOCA-Host and DOCA Drivers Known Issues 149
2.2.8.3 BlueField-3 Firmware Known Issues 154
2.2.8.4 BlueField-2 Firmware Known Issues 160
2.2.9 BlueField Firmware Cables 165
2.2.9.1 BlueField-3 Validated and Supported Cables and Modules 165
2.2.9.1.1 Cables Lifecycle Legend 165
2.2.9.1.2 NDR / 400GbE / 800GbE Cables 166
2.2.9.1.3 HDR / 200GbE Cables 167
2.2.9.1.4 EDR / 100GbE Cables 171
2.2.9.1.5 FDR / 56GbE Cables 175
2.2.9.1.6 40GbE Cables 176
2.2.9.1.7 25GbE Cables 177
2.2.9.1.8 10GbE Cables 179
2.2.9.1.9 1GbE Cables 180
2.2.9.1.10 Supported 3rd Party Cables and Modules 180
2.2.9.2 BlueField-2 Validated and Supported Cables and Modules 188
2.2.9.2.1 Cables Lifecycle Legend 188
2.2.9.2.2 NDR Cables 189
2.2.9.2.3 HDR / 200GbE / 400GbE Cables 191
2.2.9.2.4 EDR / 100GbE Cables 196
2.2.9.2.5 FDR / 56GbE Cables 199
2.2.9.2.6 40GbE Cables 200

2.2.9.2.7 25GbE Cables 202
2.2.9.2.8 10GbE Cables 203
2.2.9.2.9 1GbE Cables 204
2.2.9.2.10 Supported 3rd Party Cables and Modules 205
2.3 BlueField and DOCA User Types 209
2.3.1 Introduction 209
2.3.2 DOCA Components 210
2.3.3 BlueField Networking Platform User Types 212
2.3.3.1 BlueField Administrator 212
2.3.3.2 DOCA Developer 213
2.4 DOCA EULA 213
2.4.1 End-User License Agreement 213
2.4.2 DOCA Legal Notices and 3rd Party Licenses 219
3 Quick Start for BlueField Developers 222
3.1 DOCA Developer Quick Start Guide 222
3.1.1 Introduction 222
3.1.2 Install BlueField Networking Platform 222
3.1.3 Install DOCA Software Package 222
3.1.4 Access BlueField 222
3.1.5 Run Reference DOCA Application 223
3.1.6 More Information 223
4 Installation and Setup 224
4.1 DOCA Profiles 224
4.1.1 Introduction 224
4.1.2 Which Profile to Install? 225
4.1.3 Profile Details 226
4.1.3.1 doca-all 226
4.1.3.2 doca-networking 226
4.1.3.3 doca-ofed 226
4.1.3.4 doca-roce 227
4.1.3.5 doca-host-basic 227
4.1.4 Supported Devices 227
4.1.5 Profile Installation Notes 228
4.2 BlueField Modes of Operation 228
4.2.1 Introduction 228
4.2.1.1 NIC Mode 228

4.2.1.2 DPU Mode 229
4.2.1.2.1 DPU Mode Architecture 229
4.2.1.2.2 InfiniBand in DPU Mode 230
4.2.1.3 DPU Mode in Zero Trust 230
4.2.2 Moving Between Operation Modes 231
4.2.2.1 Identifying Which Mode BlueField is Currently Operating In 233
4.2.2.2 Changing Mode 235
4.2.2.2.1 Using External Host Command Line 235
4.2.2.2.2 Using External Host UEFI Menu 236
4.2.2.2.3 Using Arm OS Command Line 238
4.2.2.2.4 Using Arm UEFI Menu 239
4.2.2.2.5 Using DPU-BMC Redfish 240
4.2.2.2.6 Using Platform-BMC NC-SI OEM Commands 241
4.3 DOCA Installation Guide for Linux 242
4.3.1 Introduction 242
4.3.2 Supported Networking Platforms 242
4.3.3 Hardware Prerequisites 243
4.3.4 DOCA Packages 243
4.3.5 Supported Host OS per DOCA-Host Installation Profile 243
4.3.6 Setting Up Build Environment for Developers 243
4.3.7 Available BF-Bundle Installation and Upgrade Methods 243
4.3.8 DOCA-Host Installation and Upgrade 244
4.3.8.1 Uninstalling Software from Host 244
4.3.8.2 Installing Software on Host 245
4.3.8.3 Upgrading Software on Host 246
4.3.8.4 Package and Kernel Module Signing 247
4.3.8.4.1 Package Signing 247
4.3.8.4.2 Kernel Module Signing 247
4.3.8.5 DOCA Extra Package and doca-kernel-support 247
4.3.8.6 Storage Installation 250
4.3.8.6.1 VMA Installation 250
4.3.8.6.2 XLIO Installation 250
4.3.8.7 Installing Proprietary Packages 251
4.3.8.7.1 Download and Install MFT and Firmware 251
4.3.8.8 DOCA-Host Installation and DKMS Management Guide 252
4.3.8.8.1 Introduction 252
4.3.8.8.2 Understanding DKMS 252
4.3.8.8.3 Supported Platforms and Profiles 252
4.3.8.8.4 Installation Procedure 253

4.3.8.8.5 Secure Boot and Package Signing ... 253
4.3.9 BF-Bundle Installation and Upgrade ... 254
4.3.9.1 Installing DOCA-Host ... 254
4.3.9.2 Installing Software on BlueField Using BF-Bundle ... 254
4.3.9.2.1 Option 1 – No Pre-defined Password ... 254
4.3.9.2.2 Option 2 – Set Pre-defined Password ... 255
4.3.9.3 Upgrading BlueField Using Standard Linux Tools ... 257
4.3.9.3.1 Upgrading DEB-based Distributions ... 258
4.3.9.3.2 Upgrading RPM-based Distributions ... 260
4.3.9.4 Installing Multiple BlueField Devices with DOCA Installer Tool ... 263
4.3.9.4.1 Prerequisites ... 264
4.3.9.4.2 DOCA Installer Operation Modes ... 264
4.3.9.4.3 Configuration File Options ... 265
4.3.9.4.4 Logs and Debuggability ... 266
4.3.9.4.5 DOCA Installer Standalone Flags ... 267
4.3.9.4.6 BlueField-2 Limitation ... 269
4.3.9.5 Building Custom BFB Installation Image ... 270
4.3.10 Additional SDKs for DOCA ... 270
4.3.10.1 Installing CUDA on NVIDIA Converged Accelerator ... 270
4.3.10.1.1 Configuring Operation Mode ... 271
4.3.10.1.2 Downloading and Installing CUDA Toolkit and Driver ... 271
4.3.10.1.3 GPUDirect RDMA ... 272
4.3.10.2 Installing Rivermax on BlueField ... 272
4.3.10.2.1 Downloading Rivermax Driver ... 273
4.3.10.2.2 Installing Rivermax Driver ... 273
4.3.10.2.3 Installing Rivermax Libraries from DOCA ... 273
4.4 DOCA Developer Guide ... 273
4.4.1 Introduction ... 274
4.4.2 Developing Using BlueField Networking Platform ... 274
4.4.2.1 Setup ... 274
4.4.2.2 Development ... 276
4.4.2.3 Testing ... 276
4.4.2.4 Publishing ... 276
4.4.3 Developing Without BlueField Networking Platform ... 277
4.4.3.1 Setup ... 274
4.4.3.2 Development ... 276
4.4.3.3 Testing ... 279
4.4.3.4 Publishing ... 276
5 DOCA Programming Guide ... 280
5.1 DOCA Programming Overview ... 280

5.1.1 Hardware Architecture 280
5.1.1.1 Arm Cores 281
5.1.1.2 Accelerated Programmable Pipeline 281
5.1.1.3 Data-Path Accelerator 281
5.1.2 DOCA SDK Architecture 282
5.1.2.1 Device Subsystem 282
5.1.2.1.1 Device Abstraction 282
5.1.2.1.2 DOCA Device Types 282
5.1.2.2 Memory Management Subsystem 283
5.1.2.3 Execution Model 284
5.1.2.3.1 Task and Event Model 285
5.1.2.3.2 Progress Engine – Managing Asynchronous Processing 285
5.2 DOCA Compatibility Policy 286
5.2.1 Introduction and Purpose 286
5.2.2 Defining Compatibility 287
5.2.2.1 Layers 287
5.2.2.2 Type of Compatibility 287
5.2.2.3 Direction of Compatibility 287
5.2.3 Software Release and Versioning Policy 287
5.2.4 DOCA Compatibility Policy Types 288
5.2.5 DOCA Dependency Compatibility Policy 288
5.2.5.1 Key Objectives 288
5.2.5.2 DOCA Dependency Policy Levels 289
5.2.5.2.1 Level 1: Perpetual Dependency Compatibility 289
5.2.5.2.2 Level 2: Limited Dependency Compatibility 289
5.2.5.3 Application of Policies 289
5.2.5.4 Update Methodology 290
5.2.6 DOCA SDK Compatibility Policy 290
5.2.6.1 DOCA SDK Versioning 290
5.2.6.2 DOCA SDK API Backward Compatibility 290
5.2.6.2.1 Source Compatibility 291
5.2.6.2.2 Binary Compatibility 291
5.2.6.2.3 Behavioral Compatibility 291
5.2.6.3 Backward Compatibility for Non-Opaque Structs Using CTSD 291
5.2.6.3.1 Library Version Management 292
5.2.6.3.2 User Responsibility 292
5.2.6.4 DPA SDK Compatibility Considerations 292
5.2.6.5 DOCA SDK Protocol Compatibility 293
5.2.6.6 DOCA SDK Dependencies Compatibility 293

5.3 DOCA Development Best Practices ... 293
5.3.1 Capability Checking ... 293
5.3.1.1 Device Capability ... 294
5.3.1.2 Library Capability ... 294
5.3.1.3 Core Capability ... 294
5.3.2 Debuggability ... 295
5.3.2.1 Return Values ... 295
5.3.2.2 SDK Logging ... 295
5.4 DOCA Libraries ... 295
5.4.1 DOCA Common ... 296
5.4.1.1 DOCA Core ... 296
5.4.1.1.1 Introduction ... 296
5.4.1.1.2 Prerequisites ... 298
5.4.1.1.3 Changes From Previous Releases ... 298
5.4.1.1.4 Architecture ... 298
5.4.1.1.5 DOCA Core Samples ... 341
5.4.1.1.6 Backward Compatibility of DOCA Core doca_buf ... 345
5.4.1.1.7 Sync Event ... 346
5.4.1.1.8 Mmap Advise ... 363
5.4.1.2 DOCA Log ... 368
5.4.1.2.1 Log Verbosity Level Enumerations ... 368
5.4.1.2.2 Logging Backends ... 369
5.4.1.2.3 Enabling DOCA SDK Libraries Logging ... 369
5.4.1.2.4 Enabling DOCA Application Logging ... 370
5.4.1.2.5 Logging DOCA Application Messages ... 371
5.4.1.2.6 Controlling Verbosity Level of Loggers ... 371
5.4.1.2.7 DOCA Log Aggregation ... 372
5.4.1.2.8 DOCA Log Samples ... 373
5.4.2 DOCA Flow ... 375
5.4.2.1 Introduction ... 375
5.4.2.2 Prerequisites ... 376
5.4.2.3 Architecture ... 377
5.4.2.4 Steering Domains ... 377
5.4.2.4.1 List of Steering Domains ... 377
5.4.2.4.2 Domains in VNF Mode ... 378
5.4.2.4.3 Domains in Switch Mode ... 378
5.4.2.5 API ... 382
5.4.2.6 Flow Life Cycle ... 382
5.4.2.6.1 Initialization Flow ... 382
5.4.2.6.2 Start Point ... 385

5.4.2.6.3 Port Operation State 386
5.4.2.6.4 Create Pipe and Pipe Entry 388
5.4.2.6.5 Teardown 424
5.4.2.7 Metadata 425
5.4.2.7.1 Metadata Actions 425
5.4.2.7.2 Behavior and Usage Considerations 425
5.4.2.7.3 Metadata Field Breakdown 425
5.4.2.7.4 Pipe Design Consideration 426
5.4.2.7.5 Scratch Area Usage by Pipe Type 426
5.4.2.7.6 Supported Metadata Fields per Hardware and Mode 427
5.4.2.8 Custom Header 427
5.4.2.8.1 Network Header Properties 428
5.4.2.8.2 IPv4 Header Definition Example 428
5.4.2.8.3 GENEVE Header Definition Example 429
5.4.2.8.4 GENEVE Option Definition Example 430
5.4.2.8.5 UDP Header Definition Example 430
5.4.2.8.6 Custom Header Lifecycle 431
5.4.2.8.7 Match and Actions Fields 432
5.4.2.8.8 Known Limitations 433
5.4.2.8.9 NIC Configuration for Custom Header Support 433
5.4.2.9 Segment Routing over IPv6 (SRv6) 434
5.4.2.9.1 SRv6 Prerequisites 434
5.4.2.9.2 SRv6 Actions Memory 434
5.4.2.9.3 SRv6 PUSH Operation 435
5.4.2.10 Packet Processing 437
5.4.2.11 Debug and Trace Features 438
5.4.2.11.1 Installation 439
5.4.2.11.2 Using Trace Libraries 439
5.4.2.11.3 Trace Features 439
5.4.2.12 DOCA Flow Samples 440
5.4.2.12.1 Sample Prerequisites 440
5.4.2.12.2 Running the Sample 440
5.4.2.13 Field String Mapping 441
5.4.2.13.1 Field String Supported Actions 441
5.4.2.13.2 Non-Matchable Field Strings 448
5.4.2.14 DOCA Flow Connection Tracking 449
5.4.2.14.1 Introduction 449
5.4.2.14.2 Architecture 449
5.4.2.14.3 CT Pipe Configuration 451
5.4.2.14.4 IPv6 Support 452
5.4.2.14.5 Prerequisites 452

5.4.2.14.6 Actions ... 454
5.4.2.14.7 Changeable Forward ... 456
5.4.2.14.8 Entry Iterator ... 457
5.4.2.14.9 API ... 457
5.4.2.14.10 DOCA Flow Connection Tracking Samples ... 462
5.4.2.15 DOCA Flow Tune Server ... 463
5.4.2.15.1 Introduction ... 464
5.4.2.15.2 Prerequisites ... 464
5.4.2.15.3 Configuration ... 464
5.4.2.15.4 API ... 465
5.4.3 DPA Subsystem ... 466
5.4.3.1 Multiple Processes on Multiple Execution Units ... 467
5.4.3.2 DPA RTOS ... 468
5.4.3.3 DPA Memory and Caches ... 468
5.4.3.4 DPA Access to NIC Accelerators ... 469
5.4.3.5 DPA Development ... 469
5.4.3.5.1 Overview ... 469
5.4.3.5.2 Flex IO SDK ... 482
5.4.3.5.3 DPA Application Authentication ... 496
5.4.3.5.4 Distribution of Execution Units Across Multiple Applications ... 516
5.4.3.5.5 Known Limitations ... 517
5.4.3.6 DOCA DPA ... 518
5.4.3.6.1 Introduction ... 518
5.4.3.6.2 Prerequisites ... 518
5.4.3.6.3 Library Changes From Previous Releases ... 518
5.4.3.6.4 Development Flow ... 520
5.4.3.6.5 Software Architecture ... 522
5.4.3.6.6 Hello World Example ... 545
5.4.3.6.7 Samples ... 548
5.4.3.7 DOCA DPA Comms ... 556
5.4.3.7.1 Introduction ... 556
5.4.3.7.2 Architecture ... 556
5.4.3.8 DOCA DPA Verbs ... 560
5.4.3.8.1 Overview ... 560
5.4.3.8.2 Deployment ... 560
5.4.3.8.3 Architecture ... 561
5.4.3.8.4 Core Concepts ... 563
5.4.3.8.5 API Reference ... 563
5.4.3.8.6 Integration with Host-side DOCA RDMA Verbs ... 568
5.4.3.8.7 Usage Patterns ... 570
5.4.3.8.8 Optimizing Performance with Compile-Time Constants ... 571

5.4.4 DOCA PCC 571
5.4.4.1 1. Introduction 572
5.4.4.2 2. Prerequisites 572
5.4.4.2.1 2.1.1. Enabling DOCA PCC 572
5.4.4.2.2 2.1.2. Configuration Notes 573
5.4.4.2.3 2.1.3. DPACC Tool 573
5.4.4.3 3. Changes From Previous Releases 573
5.4.4.3.1 3.1. Changes in 3.3.0 573
5.4.4.4 4. Dependencies 573
5.4.4.5 5. Architecture 573
5.4.4.5.1 5.1. Host Library 573
5.4.4.5.2 5.2. Device Libraries 574
5.4.4.5.3 5.3. Development Flow 575
5.4.4.5.4 5.4. System Design 576
5.4.4.6 6. API 577
5.4.4.6.1 6.1. Host API 577
5.4.4.6.2 6.2. Device API 584
5.4.4.7 7. Histograms 588
5.4.4.7.1 7.1. Histogram Concepts 588
5.4.4.7.2 7.2. Histogram Lifecycle 589
5.4.4.7.3 7.3. Device API Reference 590
5.4.4.7.4 7.4. Remote Management via PPCC 591
5.4.5 DOCA DMA 592
5.4.5.1 Introduction 592
5.4.5.2 Prerequisites 592
5.4.5.3 Library Changes From Previous Releases 592
5.4.5.3.1 Changes in 2.9.0 592
5.4.5.4 Environment 592
5.4.5.5 Architecture 593
5.4.5.5.1 Objects 594
5.4.5.6 Configuration Phase 594
5.4.5.6.1 Configurations 594
5.4.5.6.2 Device Support 595
5.4.5.6.3 Buffer Support 595
5.4.5.7 Execution Phase 595
5.4.5.7.1 Tasks 595
5.4.5.7.2 Events 596
5.4.5.8 State Machine 596
5.4.5.8.1 Idle 597
5.4.5.8.2 Starting 597
5.4.5.8.3 Running 597

5.4.5.8.4 Stopping 597
5.4.5.9 Alternative Datapath Options 598
5.4.5.9.1 GPU Datapath 598
5.4.5.10 DOCA DMA Samples 598
5.4.5.10.1 Running the Samples 599
5.4.5.10.2 Samples 599
5.4.6 DOCA Comch 601
5.4.6.1 Introduction 601
5.4.6.2 Prerequisites 602
5.4.6.3 Changes From Previous Release 602
5.4.6.4 Environment 602
5.4.6.5 Architecture 603
5.4.6.5.1 Objects 604
5.4.6.5.2 Security Considerations 604
5.4.6.5.3 Initialization Flow 604
5.4.6.5.4 Teardown Flow 607
5.4.6.5.5 MsgQ (DPA Communication) 608
5.4.6.6 Configuration Phase 609
5.4.6.6.1 Configurations 609
5.4.6.6.2 Device Support 610
5.4.6.6.3 Buffer Support 611
5.4.6.7 Execution Phase 611
5.4.6.7.1 Tasks 611
5.4.6.7.2 Events 614
5.4.6.8 State Machine 616
5.4.6.8.1 Idle 616
5.4.6.8.2 Starting 617
5.4.6.8.3 Running 617
5.4.6.8.4 Stopping 617
5.4.6.9 Alternative Datapath Options 618
5.4.6.9.1 DPA Support 618
5.4.6.9.2 GPU Datapath Support 621
5.4.6.10 DOCA Comch Samples 622
5.4.6.10.1 Running the Samples 622
5.4.6.10.2 Samples 623
5.4.7 DOCA UROM 625
5.4.7.1 Introduction 625
5.4.7.2 Prerequisites 625
5.4.7.3 Architecture 626
5.4.7.3.1 UROM Deployment 626
5.4.7.3.2 UROM Framework 626

5.4.7.3.3 UROM Installation 628
5.4.7.4 API 630
5.4.7.4.1 DOCA_UROM_SERVICE_FILE 630
5.4.7.4.2 doca_urom_service 631
5.4.7.4.3 doca_urom_service_plugin_info 631
5.4.7.4.4 doca_urom_service_get_workers_by_gid_task 631
5.4.7.4.5 doca_urom_service_create 631
5.4.7.4.6 doca_urom_service_destroy 632
5.4.7.4.7 doca_urom_service_set_max_comm_msg_size 632
5.4.7.4.8 doca_urom_service_as_ctx 632
5.4.7.4.9 doca_urom_service_get_plugins_list 632
5.4.7.4.10 doca_urom_service_get_cpuset 633
5.4.7.4.11 doca_urom_service_get_workers_by_gid_task_allocate_init 633
5.4.7.4.12 doca_urom_service_get_workers_by_gid_task_release 633
5.4.7.4.13 doca_urom_service_get_workers_by_gid_task_as_task 633
5.4.7.4.14 doca_urom_service_get_workers_by_gid_task_get_workers_count 634
5.4.7.4.15 doca_urom_service_get_workers_by_gid_task_get_worker_ids 634
5.4.7.4.16 doca_urom_worker 634
5.4.7.4.17 doca_urom_worker_cmd_task 634
5.4.7.4.18 doca_urom_worker_cmd_task_completion_cb_t 635
5.4.7.4.19 doca_urom_worker_create 635
5.4.7.4.20 doca_urom_worker_destroy 635
5.4.7.4.21 doca_urom_worker_set_service 635
5.4.7.4.22 doca_urom_worker_set_id 636
5.4.7.4.23 doca_urom_worker_set_gid 636
5.4.7.4.24 doca_urom_worker_set_plugins 636
5.4.7.4.25 doca_urom_worker_set_env 636
5.4.7.4.26 doca_urom_worker_as_ctx 637
5.4.7.4.27 doca_urom_worker_cmd_task_allocate_init 637
5.4.7.4.28 doca_urom_worker_cmd_task_release 637
5.4.7.4.29 doca_urom_worker_cmd_task_set_plugin 637
5.4.7.4.30 doca_urom_worker_cmd_task_set_cb 638
5.4.7.4.31 doca_urom_worker_cmd_task_get_payload 638
5.4.7.4.32 doca_urom_worker_cmd_task_get_response 638
5.4.7.4.33 doca_urom_worker_cmd_task_get_user_data 638
5.4.7.4.34 doca_urom_worker_cmd_task_as_task 638
5.4.7.4.35 doca_urom_domain 639
5.4.7.4.36 doca_urom_domain_allgather_cb_t 639
5.4.7.4.37 doca_urom_domain_req_test_cb_t 639
5.4.7.4.38 doca_urom_domain_req_free_cb_t 639
5.4.7.4.39 doca_urom_domain_oob_coll 640

5.4.7.4.40 doca_urom_domain_create ... 640
5.4.7.4.41 doca_urom_domain_destroy ... 640
5.4.7.4.42 doca_urom_domain_set_workers ... 641
5.4.7.4.43 doca_urom_domain_add_buffer ... 641
5.4.7.4.44 doca_urom_domain_set_oob ... 641
5.4.7.4.45 doca_urom_domain_as_ctx ... 641
5.4.7.5 Execution Model ... 642
5.4.7.6 UROM Building Blocks ... 642
5.4.7.6.1 Program Flow ... 642
5.4.7.6.2 Plugin Development ... 645
5.4.7.7 DOCA UROM Samples ... 648
5.4.7.7.1 Sample Prerequisite ... 648
5.4.7.7.2 Running the Sample ... 648
5.4.7.7.3 UROM Plugin Samples ... 649
5.4.7.7.4 UROM Program Samples ... 649
5.4.8 DOCA RDMA ... 651
5.4.8.1 1. Introduction ... 651
5.4.8.2 2. Prerequisites ... 652
5.4.8.3 3. Changes From Previous Release ... 652
5.4.8.4 4. Environment ... 652
5.4.8.5 5. Architecture ... 652
5.4.8.5.1 5.1. Objects ... 653
5.4.8.6 6. Configuration Phase ... 653
5.4.8.6.1 6.1. Configurations ... 653
5.4.8.6.2 6.2. Device Support ... 655
5.4.8.6.3 6.3. Buffer Support ... 655
5.4.8.6.4 6.4. Multiple Connections ... 655
5.4.8.6.5 6.5. Establishing RDMA Connections ... 656
5.4.8.7 7. Execution Phase ... 661
5.4.8.7.1 7.1. Tasks ... 661
5.4.8.7.2 7.2. Events ... 677
5.4.8.8 8. State Machine ... 677
5.4.8.8.1 8.1.1. Idle ... 677
5.4.8.8.2 8.1.2. Starting ... 677
5.4.8.8.3 8.1.3. Running ... 677
5.4.8.8.4 8.1.4. Stopping ... 678
5.4.8.9 9. Alternative Datapath Options ... 678
5.4.8.9.1 9.1. DPA Datapath ... 678
5.4.8.9.2 9.2. GPU Datapath ... 679
5.4.8.10 10. DOCA RDMA Samples ... 679
5.4.8.10.1 10.1. Running the Samples ... 679

5.4.8.10.2 10.2. Configuring Scalable Functions for DOCA RDMA Samples 681
5.4.8.10.3 10.3. Samples 682
5.4.9 DOCA RDMA Verbs 692
5.4.9.1 1. Introduction 692
5.4.9.2 2. Prerequisites 692
5.4.9.3 3. Environment 693
5.4.9.4 4. Architecture 693
5.4.9.4.1 4.1. Objects 693
5.4.9.4.2 4.2. IB Verbs Bridge 702
5.4.9.5 5. Alternative Data Path Options 703
5.4.9.5.1 5.1. DPA Data Path 703
5.4.9.5.2 5.2. GPU Data Path 704
5.4.9.6 6. Samples 705
5.4.9.6.1 6.1. Verbs Server Client 705
5.4.9.6.2 6.2. Verbs Wait CQ WR 706
5.4.10 DOCA Ethernet 707
5.4.10.1 1. Overview 707
5.4.10.1.1 1.1. Key Features 708
5.4.10.1.2 1.2. High Level Usage Flow 708
5.4.10.2 2. Key Concepts 708
5.4.10.3 3. Prerequisites & Environment 709
5.4.10.4 4. Changes From Previous Releases 709
5.4.10.4.1 4.1. Changes in 3.5.0 709
5.4.10.5 5. Architecture 710
5.4.10.5.1 5.1. DOCA ETH RXQ 710
5.4.10.5.2 5.2. 2 DOCA ETH TXQ 715
5.4.10.6 6. Configuration Phase 716
5.4.10.6.1 6.1. Mandatory Configurations 716
5.4.10.6.2 6.2. Optional Configurations 717
5.4.10.6.3 6.3. Device Support 718
5.4.10.6.4 6.4. Buffer Support 719
5.4.10.7 7. Execution Phase 719
5.4.10.7.1 7.1. Common Behaviour 719
5.4.10.7.2 7.2. Receiving Packets 719
5.4.10.7.3 7.3. Sending Packets 720
5.4.10.7.4 7.4. Typical Call Sequence 721
5.4.10.8 8. State Machine 722
5.4.10.9 9. 9. GPU Datapath 723
5.4.10.10 10. Performance Considerations 724
5.4.10.11 11. DOCA ETH Samples 725
5.4.10.11.1 11.1. Running the Samples 725

5.4.10.11.2 11.2. Common Sample Skeleton 726
5.4.10.11.3 11.3. TXQ Samples 726
5.4.10.11.4 11.4. RXQ Samples 728
5.4.11 DOCA GPUNetIO 730
5.4.11.1 Introduction 730
5.4.11.2 Changes From Previous Releases 733
5.4.11.2.1 Changes in 3.4.0 733
5.4.11.3 General Performance and Best Practices 733
5.4.11.3.1 Build Configuration 733
5.4.11.3.2 Hardware and System Setup 733
5.4.11.3.3 Programming Guidelines 734
5.4.11.3.4 Permissions 734
5.4.11.4 Navigation Hub 734
5.4.11.5 GPUNetIO Installation and Setup 734
5.4.11.5.1 NIC Configuration 736
5.4.11.5.2 PCIe Configuration 738
5.4.11.5.3 GPU Configuration 739
5.4.11.5.4 DGX Spark 743
5.4.11.6 GPUNetIO Architecture and Design 744
5.4.11.6.1 Architecture 744
5.4.11.6.2 GDAKI Data Path for Different Protocols 746
5.4.11.7 GPUNetIO API Reference 751
5.4.11.7.1 CPU Functions 751
5.4.11.7.2 DOCA PE 757
5.4.11.7.3 GPU Functions – Ethernet 757
5.4.11.7.4 GPU Functions – Verbs 766
5.4.11.7.5 GPU Functions – RDMA 772
5.4.11.7.6 GPU Functions – DMA 777
5.4.11.7.7 GPU Functions – Comch 778
5.4.11.8 GPUNetIO Sample Guide 780
5.4.11.8.1 Ethernet Send Wait Time 780
5.4.11.8.2 Ethernet Simple Receive 783
5.4.11.8.3 Ethernet Simple Send 785
5.4.11.8.4 RDMA Client Server 786
5.4.11.8.5 Verbs Samples 789
5.4.11.8.6 GPU DMA Copy 794
5.4.12 DOCA App Shield 794
5.4.12.1 Introduction 794
5.4.12.2 Prerequisites 795
5.4.12.3 Dependencies 797
5.4.12.4 API 797

5.4.12.4.1 doca_apsh_dma_dev_set ... 797
5.4.12.4.2 Capabilities Per System ... 797
5.4.12.5 App Shield Initialization and Teardown ... 805
5.4.12.5.1 doca_apsh_ctx ... 805
5.4.12.5.2 doca_apsh_system ... 806
5.4.12.5.3 doca_apsh_config.py Tool ... 807
5.4.12.6 DOCA App Shield Samples ... 808
5.4.12.6.1 Sample Prerequisites ... 808
5.4.12.6.2 Running the Sample ... 808
5.4.12.6.3 Samples ... 809
5.4.13 DOCA Compress ... 816
5.4.13.1 Introduction ... 816
5.4.13.2 Prerequisites ... 816
5.4.13.3 Changes From Previous Releases ... 816
5.4.13.3.1 Changes in 2.9.0 ... 816
5.4.13.4 Environment ... 816
5.4.13.5 Architecture ... 817
5.4.13.5.1 Supported Compress/Decompress Algorithms ... 817
5.4.13.5.2 Supported Checksum Methods ... 817
5.4.13.5.3 Objects ... 818
5.4.13.5.4 Source and Destination Location ... 818
5.4.13.6 Configuration Phase ... 819
5.4.13.6.1 Configurations ... 819
5.4.13.6.2 Device Support ... 819
5.4.13.6.3 Buffer Support ... 819
5.4.13.7 Execution Phase ... 820
5.4.13.7.1 Tasks ... 820
5.4.13.7.2 Events ... 825
5.4.13.8 State Machine ... 825
5.4.13.8.1 States ... 826
5.4.13.9 Alternative Datapath Options ... 827
5.4.13.10 DOCA Compress Samples ... 827
5.4.13.10.1 Running the Sample ... 827
5.4.13.10.2 Samples ... 828
5.4.13.10.3 Backward Compatibility ... 829
5.4.14 DOCA SHA ... 830
5.4.14.1 Introduction ... 830
5.4.14.2 Prerequisites ... 830
5.4.14.3 Environment ... 830
5.4.14.4 Architecture ... 830
5.4.14.4.1 Objects ... 832

5.4.14.5 Configuration Phase 832
5.4.14.5.1 Configurations 832
5.4.14.5.2 Device Support 833
5.4.14.5.3 Buffer Support 833
5.4.14.6 Execution Phase 833
5.4.14.6.1 Tasks 833
5.4.14.6.2 Events 837
5.4.14.7 State Machine 837
5.4.14.7.1 Idle 837
5.4.14.7.2 Starting 838
5.4.14.7.3 Running 838
5.4.14.7.4 Stopping 838
5.4.14.8 Alternative Datapath Options 838
5.4.14.9 DOCA SHA Samples 838
5.4.14.9.1 Running the Samples 839
5.4.14.9.2 Samples 839
5.4.15 DOCA Erasure Coding 841
5.4.15.1 Introduction 841
5.4.15.1.1 Glossary 841
5.4.15.2 Prerequisites 841
5.4.15.3 Changes From Previous Releases 842
5.4.15.3.1 Changes in 2.10.0 842
5.4.15.4 Environment 842
5.4.15.5 Architecture 842
5.4.15.5.1 Flows 843
5.4.15.5.2 Create Redundancy Blocks 843
5.4.15.5.3 Recover Block 844
5.4.15.5.4 Objects 845
5.4.15.6 Configuration Phase 845
5.4.15.6.1 Configurations 846
5.4.15.6.2 Device Support 846
5.4.15.6.3 Buffer Support 846
5.4.15.7 Execution Phase 847
5.4.15.7.1 Matrix Generate 847
5.4.15.7.2 Tasks 849
5.4.15.8 DOCA Erasure Coding Samples 856
5.4.15.8.1 Sample Prerequisites 856
5.4.15.8.2 Running the Sample 856
5.4.15.8.3 Samples 857
5.4.16 DOCA AES-GCM 859
5.4.16.1 Introduction 859

5.4.16.2 Prerequisites 859
5.4.16.3 Changes From Previous Releases 859
5.4.16.3.1 Changes in 2.9.0 859
5.4.16.4 Environment 860
5.4.16.5 Architecture 860
5.4.16.5.1 Objects 861
5.4.16.6 Configuration Phase 861
5.4.16.6.1 Configurations 861
5.4.16.6.2 Device Support 862
5.4.16.6.3 Buffer Support 862
5.4.16.7 Execution Phase 862
5.4.16.7.1 Tasks 862
5.4.16.7.2 Events 866
5.4.16.8 State Machine 866
5.4.16.8.1 Idle 866
5.4.16.8.2 Starting 866
5.4.16.8.3 Running 866
5.4.16.8.4 Stopping 867
5.4.16.9 Alternative Datapath Options 867
5.4.16.10 DOCA AES-GCM Samples 867
5.4.16.10.1 Running the Samples 867
5.4.16.10.2 Samples 868
5.4.17 DOCA Rivermax 869
5.4.17.1 Introduction 870
5.4.17.2 Prerequisites 870
5.4.17.3 Environment 870
5.4.17.3.1 System Requirements 870
5.4.17.3.2 Performance Configuration 870
5.4.17.3.3 User Privileges 870
5.4.17.4 Linux Kernel Capabilities Requirement 871
5.4.17.5 Architecture 871
5.4.17.5.1 Objects 871
5.4.17.6 Library Features 872
5.4.17.6.1 Header/Payload Split 872
5.4.17.6.2 Hardware-Accelerated Packet Placement 872
5.4.17.7 Configuration Phase 872
5.4.17.7.1 Configurations 872
5.4.17.7.2 Device Support 873
5.4.17.7.3 Buffer Support 873
5.4.17.8 Execution Phase 873
5.4.17.8.1 Events 873

5.4.17.8.2 Runtime Configurations 874
5.4.17.9 State Machine 874
5.4.17.9.1 Idle 874
5.4.17.9.2 Starting 875
5.4.17.9.3 Running 875
5.4.17.9.4 Stopping 875
5.4.17.10 DOCA Rivermax Samples 875
5.4.17.10.1 Running the Samples 876
5.4.17.10.2 Samples 876
5.4.18 DOCA Telemetry Exporter 879
5.4.18.1 Introduction 880
5.4.18.2 Architecture 881
5.4.18.2.1 DOCA Telemetry Exporter API Walkthrough 881
5.4.18.2.2 DOCA Telemetry Exporter NetFlow API Walkthrough 883
5.4.18.2.3 DOCA Telemetry Exporter Metrics API Walkthrough 884
5.4.18.2.4 DOCA Telemetry Exporter OTLP Logs API Walkthrough 887
5.4.18.3 API 890
5.4.18.3.1 DOCA Telemetry Exporter Buffer Attributes 890
5.4.18.3.2 DOCA Telemetry Exporter File Write Attributes 891
5.4.18.3.3 DOCA Telemetry Exporter IPC Attributes 891
5.4.18.3.4 DOCA Telemetry Exporter Source Attributes 892
5.4.18.3.5 DOCA Telemetry Exporter Netflow Collector Attributes 892
5.4.18.3.6 DOCA Telemetry Exporter Source Report 892
5.4.18.3.7 DOCA Telemetry Exporter Schema Add Type 893
5.4.18.3.8 DOCA Telemetry Exporter Metrics 893
5.4.18.3.9 DOCA Telemetry Exporter Logs 895
5.4.18.4 Telemetry Event Data Format 896
5.4.18.5 Data Outputs 897
5.4.18.5.1 Inter-process Communication 897
5.4.18.5.2 NetFlow 900
5.4.18.5.3 Fluent Bit 900
5.4.18.5.4 Prometheus Endpoint 900
5.4.18.5.5 Prometheus Remote Write 901
5.4.18.5.6 Open Telemetry 901
5.4.18.6 DOCA Telemetry Exporter Samples 901
5.4.18.6.1 Running the Sample 902
5.4.18.6.2 Samples 902
5.4.19 DOCA Telemetry 904
5.4.19.1 Introduction 904
5.4.19.2 Supported Libraries 905
5.4.19.3 DOCA Telemetry Adp Retx 905

5.4.19.3.1 Introduction 905
5.4.19.3.2 Prerequisites 905
5.4.19.3.3 Environment 906
5.4.19.3.4 Architecture 906
5.4.19.3.5 Configuration Phase 907
5.4.19.3.6 Execution Phase 908
5.4.19.3.7 State Machine 909
5.4.19.3.8 Alternative Datapath Options 910
5.4.19.3.9 DOCA Telemetry Adp Retx Sample 910
5.4.19.4 DOCA Telemetry Diag 911
5.4.19.4.1 Introduction 911
5.4.19.4.2 Prerequisites 911
5.4.19.4.3 Environment 912
5.4.19.4.4 Architecture 912
5.4.19.4.5 Configuration Phase 914
5.4.19.4.6 Execution Phase 919
5.4.19.4.7 State Machine 919
5.4.19.4.8 Alternative Datapath Options 921
5.4.19.4.9 DOCA Telemetry Diagnostics Sample 921
5.4.19.4.10 Visualization of Data IDs 922
5.4.19.4.11 Appendix – List of Supported Data IDs 923
5.4.19.4.12 DOCA Telemetry Diag for Windows 928
5.4.19.5 DOCA Telemetry DPA 944
5.4.19.5.1 Introduction 944
5.4.19.5.2 Prerequisites 944
5.4.19.5.3 Environment 945
5.4.19.5.4 Architecture 946
5.4.19.5.5 Configuration Phase 946
5.4.19.5.6 Execution Phase 948
5.4.19.5.7 State Machine 950
5.4.19.5.8 Alternative Datapath Options 951
5.4.19.6 DOCA Telemetry PCC 952
5.4.19.6.1 Introduction 952
5.4.19.6.2 Prerequisites 952
5.4.19.6.3 Environment 953
5.4.19.6.4 Architecture 954
5.4.19.6.5 Schema Retrieval 954
5.4.19.6.6 Run-time Retrieval 958
5.4.19.6.7 State Machine 960
5.4.19.6.8 Alternative Datapath Options 961
5.4.19.6.9 DOCA Telemetry PCC Sample 961

5.4.19.7 DOCA Telemetry PCI 962
5.4.19.7.1 Introduction 962
5.4.19.7.2 Prerequisites 962
5.4.19.7.3 Environment 963
5.4.19.7.4 Architecture 964
5.4.19.7.5 Configuration Phase 964
5.4.19.7.6 Execution Phase 965
5.4.19.7.7 Alternative Datapath Options 967
5.4.19.8 DOCA Telemetry PHY 967
5.4.19.8.1 Introduction 967
5.4.19.8.2 Prerequisites 967
5.4.19.8.3 Environment 968
5.4.19.8.4 Architecture 968
5.4.19.8.5 Configuration Phase 969
5.4.19.8.6 Execution Phase 972
5.4.19.8.7 State Machine 973
5.4.19.8.8 Alternative Datapath Options 974
5.4.19.8.9 DOCA Telemetry Phy Sample 974
5.4.20 DOCA Device Emulation 975
5.4.20.1 Introduction 975
5.4.20.2 Known Limitations 976
5.4.20.3 DOCA DevEmu PCI 976
5.4.20.3.1 Introduction 976
5.4.20.3.2 Prerequisites 977
5.4.20.3.3 Environment 977
5.4.20.3.4 Architecture 978
5.4.20.3.5 Device Support 983
5.4.20.3.6 PCIe Device 983
5.4.20.3.7 DOCA DevEmu PCI Generic 987
5.4.20.3.8 DOCA DevEmu PCI TLP 1015
5.4.20.4 DOCA DevEmu Virtio 1035
5.4.20.4.1 Introduction 1035
5.4.20.4.2 Prerequisites 1036
5.4.20.4.3 Environment 1036
5.4.20.4.4 Architecture 1036
5.4.20.4.5 DOCA DevEmu Virtio-fs 1039
5.4.20.4.6 DOCA DevEmu Virtio-net 1048
5.4.20.4.7 DOCA DevEmu Virtio-blk 1063
5.4.21 DOCA STA 1078
5.4.21.1 Introduction 1078
5.4.21.2 Prerequisites 1079

5.4.21.3 Changes From Previous Release 1079
5.4.21.4 Environment 1080
5.4.21.5 Architecture 1080
5.4.21.5.1 DOCA STA Contexts 1080
5.4.21.5.2 Key DOCA Libraries 1080
5.4.21.5.3 High-Level Connectivity 1081
5.4.21.5.4 Objects 1082
5.4.21.6 Configuration Phase 1082
5.4.21.6.1 Configurations 1082
5.4.21.7 Execution Phase 1084
5.4.21.7.1 RDMA Connect 1084
5.4.21.7.2 Tasks 1085
5.4.21.8 State Machine 1089
5.4.21.8.1 Idle 1089
5.4.21.8.2 Starting 1090
5.4.21.8.3 Running 1090
5.4.21.8.4 Stopping 1090
5.4.22 DOCA Management 1091
5.4.22.1 Introduction 1091
5.4.22.2 Prerequisites 1091
5.4.22.3 Library Changes From Previous Releases 1091
5.4.22.3.1 Changes in 3.3.0 1091
5.4.22.3.2 Changes in 3.4.0 1091
5.4.22.3.3 Changes in 3.5.0 1092
5.4.22.4 Environment 1092
5.4.22.5 Architecture 1092
5.4.22.5.1 Objects 1093
5.4.22.6 Expected Usage Flow 1096
5.4.22.6.1 Device Context Creation 1096
5.4.22.6.2 Device Representor Context Creation 1097
5.4.22.6.3 ICM Quota Configuration 1098
5.4.22.6.4 Raw APIs 1099
5.4.22.7 DOCA Management Samples 1101
5.4.22.7.1 Running the Samples 1101
5.4.22.7.2 Listing Management Targets 1102
5.4.22.7.3 Samples 1103
5.5 DOCA Utils 1113
5.5.1 DOCA Arg Parser 1114
5.5.1.1 Introduction 1114
5.5.1.2 API 1114
5.5.1.2.1 doca_argp_param 1114

5.5.1.2.2 doca_argp_param_create 1114
5.5.1.2.3 doca_argp_register_param 1115
5.5.1.2.4 doca_argp_cmd 1115
5.5.1.2.5 doca_argp_cmd_create 1115
5.5.1.2.6 doca_argp_cmd_register_param 1115
5.5.1.2.7 doca_argp_cmd_register_cmd 1116
5.5.1.2.8 doca_argp_register_cmd 1116
5.5.1.2.9 doca_argp_set_dpdk_program 1116
5.5.1.2.10 doca_argp_init 1117
5.5.1.2.11 doca_argp_start 1117
5.5.1.2.12 doca_argp_destroy 1117
5.5.1.3 JSON Syntax 1117
5.5.1.4 DOCA General Flags 1117
5.5.1.5 DOCA Program Flags 1118
5.5.1.6 DOCA Program Commands 1118
5.5.1.6.1 Detailed Example of Program Commands 1118
5.5.1.7 Syntax – Device and Device Representor 1119
5.5.1.7.1 JSON Syntax 1120
5.5.1.7.2 PCIe Identifiers 1120
5.5.1.7.3 InfiniBand (IB) Identifiers 1121
5.5.1.7.4 Network Interface Names 1122
5.5.1.7.5 Network Interface Indices 1122
5.5.1.7.6 Auxiliary Device Indices 1122
5.6 DOCA Drivers 1123
5.6.1 DOCA UCX 1123
5.6.1.1 Introduction 1123
5.6.1.2 Prerequisites 1123
5.6.1.3 Architecture 1124
5.6.1.3.1 UCP Objects 1125
5.6.1.4 API 1125
5.6.1.4.1 ucs_status_t 1126
5.6.1.4.2 ucp_init 1126
5.6.1.4.3 ucp_cleanup 1127
5.6.1.4.4 ucp_worker_create 1127
5.6.1.4.5 ucp_worker_destroy 1127
5.6.1.4.6 ucp_listener_create 1128
5.6.1.4.7 ucp_listener_destroy 1128
5.6.1.4.8 ucp_ep_create 1129
5.6.1.4.9 ucs_status_ptr_t 1130
5.6.1.4.10 ucp_ep_close_nbx 1130
5.6.1.4.11 ucp_request_param_t 1131

5.6.1.4.12 ucp_worker_progress ... 1131
5.6.1.4.13 ucp_am_send_nbx ... 1132
5.6.1.4.14 ucp_worker_set_am_recv_handler ... 1133
5.6.1.4.15 ucp_am_recv_data_nbx ... 1134
5.6.1.5 UCX Best Practices ... 1134
5.6.1.5.1 Initialization ... 1134
5.6.1.5.2 Communications ... 1135
5.6.2 MLX Drivers and Technologies ... 1135
5.6.2.1 InfiniBand Network ... 1136
5.6.2.1.1 InfiniBand Interface ... 1136
5.6.2.1.2 NVIDIA SM ... 1136
5.6.2.1.3 InfiniBand QoS ... 1175
5.6.2.1.4 IP Over InfiniBand ... 1178
5.6.2.1.5 Advanced Transport ... 1186
5.6.2.1.6 Optimized Memory Access ... 1188
5.6.2.1.7 NVIDIA PeerDirect ... 1191
5.6.2.1.8 CPU Overhead Distribution ... 1192
5.6.2.1.9 Out-of-order Data Placement ... 1192
5.6.2.1.10 IB Router ... 1192
5.6.2.1.11 MAD Congestion Control ... 1194
5.6.2.2 Ethernet Network ... 1195
5.6.2.2.1 Ethernet Interface ... 1195
5.6.2.2.2 Ethernet QoS ... 1197
5.6.2.2.3 Ethtool ... 1206
5.6.2.2.4 Checksum Offload ... 1209
5.6.2.2.5 Ignore FCS Errors ... 1209
5.6.2.2.6 Flow Control ... 1209
5.6.2.2.7 Explicit Congestion Notification ... 1214
5.6.2.2.8 RSS Support ... 1215
5.6.2.2.9 Time Stamping ... 1216
5.6.2.2.10 Flow Steering ... 1221
5.6.2.2.11 Wake-on-LAN ... 1224
5.6.2.2.12 Q-in-Q Tunneling ... 1224
5.6.2.2.13 VLAN Stripping in Linux Verbs ... 1225
5.6.2.2.14 Offloaded Traffic Sniffer ... 1225
5.6.2.2.15 Dump Configuration ... 1226
5.6.2.2.16 Local Loopback Disable ... 1227
5.6.2.2.17 kTLS Offloads ... 1228
5.6.2.2.18 IPsec Crypto Offload ... 1229
5.6.2.2.19 IPsec Packet Offload ... 1231
5.6.2.2.20 MACsec Full Offload ... 1233

5.6.2.3 RDMA 1236
5.6.2.3.1 RDMA-aware Networks Programming Guide 1236
5.6.2.3.2 RDMA over Converged Ethernet 1358
5.6.2.3.3 Adaptive Retransmission: Parameters Control 1367
5.6.2.3.4 Migration to RDMA-Core 1375
5.6.2.3.5 Congestion Control Infrastructure 1402
5.6.2.4 DPU 1428
5.6.2.4.1 QoS Configuration 1429
5.6.2.4.2 Shared RQ Mode 1431
5.6.2.5 Storage Protocols 1432
5.6.2.5.1 SRP - SCSI RDMA Protocol 1433
5.6.2.5.2 iSER - iSCSI Extensions for RDMA 1441
5.6.2.5.3 Lustre 1443
5.6.2.5.4 NVME-oF - NVM Express over Fabrics 1444
5.6.2.6 Virtualization 1444
5.6.2.6.1 SR-IOV 1444
5.6.2.6.2 Paravirtualization 1460
5.6.2.6.3 VXLAN Hardware Stateless Offloads 1461
5.6.2.6.4 Q-in-Q Encapsulation per VF in Linux 1463
5.6.2.6.5 VLAN Double Tagging 1464
5.6.2.6.6 Scalable Functions 1465
5.6.2.7 Resiliency 1465
5.6.2.7.1 Reset Flow 1465
5.6.2.8 Docker Containers 1467
5.6.2.8.1 Docker Using SR-IOV 1468
5.6.2.8.2 Kubernetes Using SR-IOV 1468
5.6.2.8.3 Kubernetes with Shared HCA 1468
5.6.2.9 HPC-X 1469
5.6.2.10 Fast Driver Unload 1469
6 DOCA Reference Applications 1470
6.1 Introduction 1470
6.1.1 Installation 1470
6.1.2 Prerequisites 1470
6.1.3 Compilation 1471
6.1.4 Developer Configurations 1471
6.2 Application Use of DOCA Libs 1472
6.3 DOCA App Shield Agent Application Guide 1474
6.3.1 Introduction 1474
6.3.2 System Design 1474

6.3.3 Application Architecture 1475
6.3.4 DOCA Libraries 1476
6.3.5 Compiling the Application 1476
6.3.5.1 Compiling All Applications 1476
6.3.5.2 Compiling Only the Current Application 1477
6.3.6 Running the Application 1477
6.3.6.1 Prerequisites 1477
6.3.6.2 Application Execution 1479
6.3.6.3 Command Line Flags 1479
6.3.6.3.1 General Flags 1479
6.3.6.3.2 Program Flags 1480
6.3.6.4 Troubleshooting 1481
6.3.7 Application Code Flow 1481
6.3.8 References 1483
6.4 DOCA Bifurcated Driver Model Reference Application 1483
6.4.1 Introduction 1483
6.4.2 System Design 1483
6.4.3 Application Architecture 1485
6.4.4 DOCA Libraries 1486
6.4.5 Compiling the Application 1486
6.4.5.1 Compiling All Applications 1486
6.4.5.2 Recompiling Only the Current Application 1486
6.4.6 Running the Application 1487
6.4.6.1 Application Execution 1487
6.4.6.2 Command Line Flags 1488
6.4.6.2.1 General Flags 1488
6.4.6.2.2 Program Flags 1489
6.4.6.3 Troubleshooting 1489
6.4.7 Application Code Flow 1489
6.5 DOCA DMA Copy Application Guide 1490
6.5.1 Introduction 1490
6.5.2 System Design 1490
6.5.3 Application Architecture 1491
6.5.4 DOCA Libraries 1492
6.5.5 Compiling the Application 1493
6.5.5.1 Compiling All Applications 1493
6.5.5.2 Compiling Only the Current Application 1493

6.5.6 Running the Application ... 1494
6.5.6.1 Application Execution ... 1494
6.5.6.2 Command Line Flags ... 1495
6.5.6.2.1 General Flags ... 1495
6.5.6.2.2 Program Flags ... 1495
6.5.6.3 Troubleshooting ... 1496
6.5.7 Application Code Flow ... 1496
6.5.8 References ... 1497
6.6 DOCA DPA All-to-all Application Guide ... 1498
6.6.1 Introduction ... 1498
6.6.2 System Design ... 1498
6.6.3 Application Architecture ... 1498
6.6.4 DOCA Libraries ... 1499
6.6.5 Dependencies ... 1499
6.6.6 Compiling the Application ... 1499
6.6.6.1 Compiling All Applications ... 1500
6.6.6.2 Compiling DPA All-to-all Application Only ... 1500
6.6.7 Running the Application ... 1501
6.6.7.1 Prerequisites ... 1501
6.6.7.2 Application Execution ... 1501
6.6.7.3 Command Line Flags ... 1502
6.6.7.3.1 General Flags ... 1502
6.6.7.3.2 Program Flags ... 1503
6.6.7.4 Troubleshooting ... 1503
6.6.8 Application Code Flow ... 1503
6.6.9 References ... 1506
6.7 DOCA East-west Overlay Encryption Application ... 1506
6.7.1 Introduction ... 1507
6.7.2 System Design ... 1508
6.7.3 Application Architecture ... 1509
6.7.4 DOCA Libraries ... 1509
6.7.5 Dependencies ... 1509
6.7.5.1 Compiling and Installing strongSwan ... 1509
6.7.6 Configuration Flow ... 1510
6.7.6.1 Enabling IPsec Packet Offload ... 1511
6.7.6.2 Configuring OVS IPsec ... 1511
6.7.6.2.1 Authentication Methods ... 1514

6.7.6.3 Ensuring IPsec is Configured 1517
6.7.6.4 Configuring OVS IPsec Using strongSwan Manually 1517
6.7.6.5 swanctl.conf Files 1519
6.7.7 Running the Application 1520
6.7.7.1 Installation 1520
6.7.7.2 Application Execution 1520
6.7.7.2.1 Script Parameters 1521
6.7.7.2.2 Using JSON Parameters File 1523
6.7.7.2.3 Passing Parameters on Command Line 1524
6.7.7.3 Troubleshooting 1525
6.7.7.4 Reverting IPsec Configuration 1525
6.7.8 References 1525
6.8 DOCA Eth L2 Forwarding Application Guide 1525
6.8.1 Introduction 1526
6.8.2 System Design 1526
6.8.3 Application Architecture 1527
6.8.4 DOCA Libraries 1528
6.8.5 Compiling the Application 1528
6.8.5.1 Compiling All Applications 1528
6.8.5.2 Compiling Only the Current Application 1528
6.8.6 Running the Application 1529
6.8.6.1 Application Execution 1529
6.8.6.2 Command Line Flags 1530
6.8.6.2.1 General Flags 1530
6.8.6.2.2 Program Flags 1530
6.8.6.3 Troubleshooting 1531
6.8.7 Application Code Flow 1531
6.8.8 References 1532
6.9 DOCA File Compression Application Guide 1532
6.9.1 Introduction 1532
6.9.2 System Design 1532
6.9.3 Application Architecture 1533
6.9.4 DOCA Libraries 1535
6.9.5 Compiling the Application 1535
6.9.5.1 Compiling All Applications 1535
6.9.5.2 Compiling File Compression Application Only 1535
6.9.6 Running the Application 1536
6.9.6.1 Application Execution 1536

6.9.6.2 Command Line Flags 1537
6.9.6.2.1 General Flags 1537
6.9.6.2.2 Program Flags 1538
6.9.6.3 Troubleshooting 1538
6.9.7 Application Code Flow 1538
6.9.8 References 1539
6.10 DOCA File Integrity Application Guide 1539
6.10.1 Introduction 1539
6.10.2 System Design 1539
6.10.3 Application Architecture 1540
6.10.4 DOCA Libraries 1541
6.10.5 Compiling the Application 1542
6.10.5.1 Compiling All Applications 1542
6.10.5.2 Compiling Only the Current Application 1542
6.10.6 Running the Application 1543
6.10.6.1 Application Execution 1543
6.10.6.2 Command Line Flags 1544
6.10.6.2.1 General Flags 1544
6.10.6.2.2 Host Program Flags 1544
6.10.6.3 Troubleshooting 1545
6.10.7 Application Code Flow 1545
6.10.8 References 1546
6.11 DOCA GPU Packet Processing Application Guide 1546
6.11.1 Introduction 1546
6.11.2 System Design 1546
6.11.3 Application Architecture 1547
6.11.3.1 ICMP Network Traffic 1547
6.11.3.2 UDP Network Traffic 1549
6.11.3.2.1 T-Rex Packet Generator 1551
6.11.3.2.2 DPDK testpmd Packet Generator 1551
6.11.3.3 TCP Network Traffic and HTTP Echo Server 1552
6.11.3.3.1 Step 1: TCP Connection Establishment 1552
6.11.3.3.2 Step 2: TCP Data Processing 1552
6.11.3.3.3 Step 3: HTTP Echo Server 1552
6.11.3.3.4 Step 4: TCP Connection Closure 1553
6.11.4 DOCA Libraries 1555
6.11.5 Dependencies 1555
6.11.6 Compiling the Application 1555

6.11.6.1 Compiling All Applications ... 1556
6.11.6.2 Compiling Only the Current Application ... 1556
6.11.7 Running the Application ... 1557
6.11.7.1 Command Line Flags ... 1557
6.11.7.1.1 General Flags ... 1557
6.11.7.1.2 Program Flags ... 1558
6.11.7.2 Troubleshooting ... 1558
6.11.8 Application Code Flow ... 1558
6.11.9 References ... 1560
6.12 NVIDIA DOCA IP Fragmentation Application Guide ... 1560
6.12.1 Introduction ... 1560
6.12.2 System Design ... 1561
6.12.3 Application Architecture ... 1561
6.12.3.1 Operational Workflow ... 1561
6.12.3.2 Performance and Scalability ... 1561
6.12.3.3 Debugging and Monitoring ... 1561
6.12.4 DOCA Libraries ... 1562
6.12.5 Dependencies ... 1562
6.12.6 Compiling the Application ... 1562
6.12.6.1 Compiling All Applications ... 1562
6.12.6.2 Compiling Only the Current Application ... 1562
6.12.7 Running the Application ... 1563
6.12.7.1 Prerequisites ... 1563
6.12.7.2 Application Execution ... 1563
6.12.7.3 Command Line Flags ... 1564
6.12.7.3.1 General Flags ... 1564
6.12.7.3.2 Program Flags ... 1565
6.12.7.4 Troubleshooting ... 1565
6.12.8 References ... 1565
6.13 DOCA IPsec Security Gateway Application Guide ... 1565
6.13.1 Introduction ... 1565
6.13.2 System Design ... 1566
6.13.3 Application Architecture ... 1569
6.13.3.1 Static Configuration ... 1569
6.13.3.2 Dynamic Configuration ... 1570
6.13.3.3 DOCA Flow Modes ... 1570
6.13.3.3.1 VNF Mode ... 1571
6.13.3.3.2 Switch Mode ... 1573

6.13.4 DOCA Libraries ... 1573
6.13.5 Compiling the Application ... 1573
6.13.5.1 Prerequisites ... 1574
6.13.5.2 Compiling All Applications ... 1574
6.13.5.3 Compiling Only the Current Application ... 1574
6.13.6 Running the Application ... 1575
6.13.6.1 Prerequisites ... 1575
6.13.6.2 Application Execution ... 1576
6.13.6.3 Command Line Flags ... 1577
6.13.6.3.1 General Flags ... 1577
6.13.6.3.2 Program Flags ... 1577
6.13.6.4 Static Configuration IPsec Rules ... 1578
6.13.6.5 Dynamic Configuration IPsec Rules ... 1581
6.13.6.6 Troubleshooting ... 1582
6.13.7 Application Code Flow ... 1582
6.13.8 Keying Daemon Integration (StrongSwan) ... 1583
6.13.8.1 End-to-end Architecture ... 1584
6.13.8.2 Building strongSwan ... 1588
6.13.8.3 Running the Solution ... 1589
6.13.9 References ... 1590
6.14 DOCA NVMe Emulation Application Guide ... 1590
6.14.1 Introduction ... 1590
6.14.2 Application Architecture ... 1590
6.14.2.1 Brief Introduction to NVMe ... 1590
6.14.2.1.1 Controller Registers ... 1590
6.14.2.1.2 Completion Queue (CQ) ... 1591
6.14.2.1.3 Submission Queue ... 1592
6.14.2.1.4 Admin Queue Pair (Admin QP) ... 1592
6.14.2.1.5 Input/Output Queue Pair (I/O QP) ... 1595
6.14.2.2 Brief Introduction To SPDK ... 1596
6.14.2.2.1 SPDK Threading Model ... 1596
6.14.2.2.2 Block Device ... 1598
6.14.2.2.3 NVMe-oF Target ... 1598
6.14.2.2.4 RPC Server ... 1600
6.14.2.2.5 RPC Client ... 1601
6.14.2.3 Solution Overview ... 1602
6.14.2.3.1 Integration with SPDK ... 1602
6.14.2.4 Data Structures ... 1606
6.14.2.4.1 Emulation Manager ... 1606
6.14.2.4.2 PCIe Device Admin ... 1607

6.14.2.4.3 DOCA Poll Group ... 1608
6.14.2.4.4 Admin Poll Group ... 1609
6.14.2.4.5 DOCA Transport ... 1609
6.14.2.4.6 PCIe Device Poll Group ... 1610
6.14.2.4.7 Admin QP ... 1612
6.14.2.4.8 DOCA IO ... 1612
6.14.2.4.9 DOCA SQ ... 1616
6.14.2.5 Control Path Flows ... 1620
6.14.2.5.1 DOCA Transport Listen ... 1620
6.14.2.5.2 Controller Register Events ... 1621
6.14.2.5.3 I/O QP Create/Destroy ... 1623
6.14.2.6 Data Path Flows ... 1626
6.14.2.6.1 Retrieve Doorbell ... 1627
6.14.2.6.2 Fetch SQE From Host ... 1627
6.14.2.6.3 Dispatch Command to NVMe-oF Target ... 1628
6.14.2.6.4 Post CQE to Host ... 1628
6.14.2.6.5 Raise RMSI-X ... 1628
6.14.2.7 Limitations ... 1629
6.14.2.7.1 Supported SPDK Versions ... 1629
6.14.2.7.2 Supported Admin Commands ... 1629
6.14.2.7.3 Supported NVM Commands ... 1629
6.14.2.7.4 SPDK Stop Listen Flow ... 1629
6.14.3 DOCA Libraries ... 1630
6.14.4 Dependencies ... 1630
6.14.5 Compiling the Application ... 1630
6.14.5.1 Compiling All Applications ... 1630
6.14.5.2 Compiling With Custom SPDK ... 1631
6.14.6 Running the Application ... 1631
6.14.6.1 Prerequisites ... 1631
6.14.6.2 Application Execution ... 1631
6.14.6.3 Command Line Flags ... 1632
6.14.6.4 Troubleshooting ... 1632
6.14.6.5 References ... 1633
6.15 DOCA PCC Application Guide ... 1633
6.15.1 Introduction ... 1633
6.15.2 System Design ... 1633
6.15.3 Application Architecture ... 1635
6.15.4 DOCA Libraries ... 1635
6.15.5 Dependencies ... 1636

6.15.6 Compiling the Application 1636
6.15.6.1 Compiling All Applications 1636
6.15.6.2 Compiling Only the Current Application 1636
6.15.6.3 Compilation Options 1637
6.15.7 Running the Application 1637
6.15.7.1 Prerequisites 1637
6.15.7.2 Application Execution 1637
6.15.7.3 Command Line Flags 1638
6.15.7.3.1 General Flags 1638
6.15.7.3.2 Program Flags 1639
6.15.7.4 Troubleshooting 1640
6.15.8 Application Code Flow 1640
6.15.9 Port Programmable Congestion Control Register 1642
6.15.9.1 Usage 1642
6.15.9.2 Internal Default Algorithm 1646
6.15.9.3 Counters 1646
6.15.9.3.1 Enabling Counters 1646
6.15.9.3.2 Querying Counters 1646
6.15.9.3.3 Getting Counter Information 1647
6.15.10 References 1647
6.16 DOCA PSP Gateway Application Guide 1647
6.16.1 Introduction 1647
6.16.2 System Design 1648
6.16.3 Application Architecture 1650
6.16.3.1 Startup vs. On-Demand Tunnel Creation 1651
6.16.3.2 Sampling 1653
6.16.3.3 Pipelines 1654
6.16.3.3.1 Host-to-Network Flows 1654
6.16.3.3.2 Network-to-Host Flows 1655
6.16.3.4 DOCA Libraries 1656
6.16.4 Compiling the Application 1656
6.16.4.1 Prerequisites 1656
6.16.4.2 Compiling All Applications 1657
6.16.4.3 Compiling Only the Current Application 1657
6.16.5 Running the Application 1658
6.16.5.1 Prerequisites 1658
6.16.5.2 Application Execution 1658
6.16.5.3 Command Line Flags 1659
6.16.5.3.1 General Flags 1659

6.16.5.3.2 Program Flags 1660
6.16.5.4 Config File JSON Format 1661
6.16.5.5 Troubleshooting 1662
6.16.6 Application Code Flow 1662
6.16.6.1 References 1666
6.17 DOCA Secure Channel Application Guide 1666
6.17.1 Introduction 1666
6.17.2 System Design 1667
6.17.3 Application Architecture 1667
6.17.4 DOCA Libraries 1668
6.17.5 Compiling the Application 1668
6.17.5.1 Compiling All Applications 1669
6.17.5.2 Compiling Only the Current Application 1669
6.17.6 Running the Application 1669
6.17.6.1 Application Execution 1669
6.17.6.2 Command Line Flags 1670
6.17.6.2.1 General Flags 1670
6.17.6.2.2 Program Flags 1671
6.17.6.3 Troubleshooting 1671
6.17.7 Application Code Flow 1672
6.17.8 References 1673
6.18 DOCA Simple Forward VNF Application Guide 1673
6.18.1 Introduction 1673
6.18.2 System Design 1673
6.18.3 Application Architecture 1674
6.18.4 DOCA Libraries 1675
6.18.5 Compiling the Application 1675
6.18.5.1 Compiling All Applications 1676
6.18.5.2 Compiling Simple Forward Application Only 1676
6.18.6 Running the Application 1677
6.18.6.1 Prerequisites 1677
6.18.6.2 Application Execution 1677
6.18.6.3 Command Line Flags 1679
6.18.6.3.1 General Flags 1679
6.18.6.3.2 Host Program Flags 1679
6.18.6.4 Troubleshooting 1680
6.18.7 Application Code Flow 1680

6.18.8 References 1681
6.19 DOCA SNAP Virtio-fs Application Guide 1682
6.19.1 Introduction 1682
6.19.2 System Design 1682
6.19.2.1 Host Side (Virtio-fs Client Driver) 1682
6.19.2.2 BlueField DPU (DOCA SNAP Virtio-fs Application) 1682
6.19.2.3 Network Communication (libnfs) 1683
6.19.2.4 NFS Server 1683
6.19.2.5 End-to-End Flow 1683
6.19.3 Application Architecture 1683
6.19.3.1 Virtio-fs Protocol Components 1683
6.19.3.1.1 FUSE (Filesystem in Userspace) 1683
6.19.3.1.2 Virtio Queues 1684
6.19.3.2 Initialization Process 1684
6.19.3.3 libnfs Integration 1684
6.19.3.4 File Operations Workflow 1684
6.19.3.5 Reset and Shutdown 1684
6.19.4 DOCA Libraries 1684
6.19.5 Dependencies 1685
6.19.5.1 Compiling the Application 1685
6.19.5.2 Compiling All Applications 1685
6.19.5.3 Compiling Only the Current Application 1686
6.19.6 Running the Application 1686
6.19.6.1 Prerequisites 1686
6.19.6.2 Application Execution 1687
6.19.6.2.1 Application Usage 1687
6.19.6.2.2 Example Command Line Execution 1687
6.19.6.2.3 JSON-Based Deployment Mode 1688
6.19.6.2.4 Mounting the Virtio-fs Device (Host/VM Side) 1688
6.19.6.3 Command Line Flags 1689
6.19.6.3.1 General Flags 1689
6.19.6.3.2 Host Program Flags 1689
6.19.6.4 Troubleshooting 1689
6.19.7 Application Code Flow 1690
6.19.7.1 Framework Initialization 1690
6.19.7.1.1 Initialization Flow 1690
6.19.7.1.2 Key Points 1691
6.19.7.2 Resource Management 1691
6.19.7.2.1 Resource Management Flow 1691
6.19.7.2.2 Key Points 1693

6.19.7.3 Device Configuration ... 1693
6.19.7.3.1 Device Configuration Flow ... 1693
6.19.7.3.2 Key Points ... 1695
6.19.7.4 Backend Integration ... 1696
6.19.7.4.1 Backend Integration Flow ... 1696
6.19.7.4.2 Key Points ... 1697
6.19.7.5 Request Handler Registration ... 1698
6.19.7.5.1 Request Handler Registration Flow ... 1698
6.19.7.5.2 Key Points ... 1699
6.19.7.6 Device Lifecycle Management ... 1700
6.19.7.6.1 Device Lifecycle Management Flow ... 1700
6.19.7.6.2 Key Points ... 1702
6.19.7.7 Runtime Operations ... 1702
6.19.7.7.1 Runtime Operations Flow ... 1702
6.19.7.7.2 Key Points ... 1704
6.19.7.8 Cleanup and Shutdown ... 1705
6.19.7.8.1 Cleanup and Shutdown Flow ... 1705
6.19.7.8.2 Key Points ... 1706
6.19.8 FUSE Operations Support Summary ... 1707
6.20 DOCA Storage Applications ... 1709
6.20.1 Introduction ... 1709
6.20.2 Applications ... 1709
6.20.2.1 doca_storage_initiator_comch ... 1710
6.20.2.2 doca_storage_target_rdma ... 1710
6.20.2.3 doca_storage_comch_to_rdma_zero_copy ... 1710
6.20.2.4 doca_storage_comch_to_rdma_gga_offload ... 1710
6.20.2.5 doca_storage_gga_offload_sbc_generator ... 1711
6.20.3 Use Case 1: Zero Copy ... 1711
6.20.3.1 Application Components ... 1711
6.20.3.2 Operational Flow ... 1711
6.20.3.2.1 Setup Phase ... 1711
6.20.3.2.2 Data Path Phase ... 1712
6.20.3.3 Memory Layout ... 1712
6.20.3.4 Data Flow ... 1713
6.20.3.5 Running the Zero Copy Use Case ... 1713
6.20.3.5.1 Run Write Data Validity Test ... 1714
6.20.3.5.2 Run Write Throughput Test ... 1714
6.20.3.5.3 Run Read Data Validity Test ... 1714
6.20.3.5.4 Run Read Throughput Test ... 1715
6.20.4 Use Case 2: GGA Offload ... 1715

6.20.4.1 Architecture and Components 1715
6.20.4.2 Operational Flow 1715
6.20.4.2.1 Setup Phase 1716
6.20.4.2.2 Data Path Phase 1716
6.20.4.3 Memory Layout 1717
6.20.4.4 Normal Read Flow 1718
6.20.4.5 Recovery Read Flow 1718
6.20.4.6 Write Flow 1719
6.20.4.7 Running the GGA Offload Use Case 1719
6.20.4.7.1 Execution Order 1720
6.20.4.7.2 Placeholders 1720
6.20.4.7.3 Generate SBC File 1720
6.20.4.7.4 Run Read Data Validity Test 1720
6.20.4.7.5 Run Read Throughput Test 1721
6.20.4.7.6 Run Write Data Validity Test 1721
6.20.4.7.7 Run Write Throughput Test 1722
6.20.5 DOCA Libraries 1722
6.20.6 Compiling the Application 1722
6.20.6.1 Compiling All Applications 1723
6.20.6.2 Compiling Storage Applications Only 1723
6.20.6.3 Troubleshooting 1723
6.20.7 Running the Applications 1724
6.20.8 DOCA Storage Comch to RDMA GGA Offload Application Guide 1724
6.20.8.1 Introduction 1724
6.20.8.2 System Design 1724
6.20.8.3 Architecture 1724
6.20.8.3.1 Control Phase 1725
6.20.8.3.2 Data Path Phase 1726
6.20.8.4 DOCA Libraries 1735
6.20.8.5 Compiling the Application 1735
6.20.8.6 Running the Application 1735
6.20.8.6.1 Application Execution 1735
6.20.8.6.2 Command-line Flags 1736
6.20.8.6.3 Troubleshooting 1738
6.20.8.7 Application Code Flow 1738
6.20.8.7.1 Main/Control Thread Flow 1739
6.20.8.7.2 Worker/Data Path Thread Flow 1744
6.20.8.8 References 1749
6.20.9 DOCA Storage Comch to RDMA Zero Copy Application Guide 1749
6.20.9.1 Introduction 1749

6.20.9.2 System Design 1749
6.20.9.3 Architecture 1749
6.20.9.3.1 Control Phase 1750
6.20.9.3.2 Data Path Phase 1751
6.20.9.4 DOCA Libraries 1753
6.20.9.5 Compiling the Application 1753
6.20.9.6 Running the Application 1753
6.20.9.6.1 Application Execution 1753
6.20.9.6.2 Command-line Flags 1754
6.20.9.6.3 Troubleshooting 1756
6.20.9.7 Application Code Flow 1756
6.20.9.7.1 Main/Control Thread Flow 1756
6.20.9.7.2 Worker/Data Path Thread Flow 1760
6.20.9.8 References 1762
6.20.10 DOCA Storage GGA Offload SBC Generator Application Guide 1762
6.20.10.1 Introduction 1762
6.20.10.2 System Design 1762
6.20.10.3 Architecture 1762
6.20.10.4 DOCA Libraries 1763
6.20.10.5 Compiling the Application 1763
6.20.10.6 Running the Application 1764
6.20.10.6.1 Application Execution 1764
6.20.10.6.2 Command-line Flags 1764
6.20.10.6.3 Troubleshooting 1766
6.20.10.7 Application Code Flow 1766
6.20.10.7.1 General Application Flow 1766
6.20.10.7.2 Transform Process 1767
6.20.10.8 References 1767
6.20.11 DOCA Storage Initiator ComCh Application Guide 1767
6.20.11.1 Introduction 1767
6.20.11.2 System Design 1767
6.20.11.3 Architecture 1768
6.20.11.3.1 Control Phase 1768
6.20.11.3.2 Data Path Phase 1769
6.20.11.4 DOCA Libraries 1769
6.20.11.5 Compiling the Application 1769
6.20.11.6 Running the Application 1770
6.20.11.6.1 Application Execution 1770
6.20.11.6.2 Command-line Flags 1770
6.20.11.6.3 Troubleshooting 1772
6.20.11.7 Application Code Flow 1772

6.20.11.7.1 Main/Control Thread Flow ... 1773
6.20.11.8 References ... 1776
6.20.12 DOCA Storage Target RDMA Application Guide ... 1776
6.20.12.1 Introduction ... 1776
6.20.12.2 System Design ... 1776
6.20.12.3 Architecture ... 1776
6.20.12.3.1 Control Phase ... 1777
6.20.12.3.2 Data Path Phase ... 1778
6.20.12.4 DOCA Libraries ... 1778
6.20.12.5 Compiling the Application ... 1778
6.20.12.6 Running the Application ... 1778
6.20.12.6.1 Application Execution ... 1778
6.20.12.6.2 Command-line Flags ... 1779
6.20.12.6.3 Troubleshooting ... 1781
6.20.12.7 Application Code Flow ... 1781
6.20.12.7.1 Main/Control Thread Flow ... 1781
6.20.12.7.2 Worker/Data Path Thread Flow ... 1784
6.20.12.8 References ... 1787
6.21 DOCA Stream Receive Performance Application Guide ... 1787
6.21.1 Introduction ... 1788
6.21.1.1 Key Technologies ... 1788
6.21.1.2 Deployment Notes ... 1788
6.21.2 System Design ... 1788
6.21.3 Architecture ... 1788
6.21.3.1 Main Application ... 1789
6.21.3.2 Configuration Management ... 1789
6.21.3.3 Global Resources Management ... 1789
6.21.3.4 Stream Management ... 1789
6.21.3.5 Application Functions and Roles ... 1789
6.21.3.6 Data Structures ... 1790
6.21.3.7 Event Handling ... 1790
6.21.3.8 Flow ... 1790
6.21.4 DOCA Libraries ... 1790
6.21.5 Dependencies ... 1791
6.21.6 Compiling the Application ... 1791
6.21.6.1 Compiling All Applications ... 1791
6.21.6.2 Compiling Only the Current Application ... 1791
6.21.7 Running the Application ... 1792
6.21.7.1 Prerequisites ... 1792
6.21.7.2 Application Execution ... 1792

6.21.7.3 Command Line Flags 1793
6.21.7.3.1 General Flags 1793
6.21.7.3.2 Program Flags 1794
6.21.7.4 Troubleshooting 1795
6.21.8 Application Code Flow 1795
6.21.9 References 1796
6.22 DOCA Switch Application Guide 1797
6.22.1 Introduction 1797
6.22.2 System Design 1797
6.22.3 Application Architecture 1799
6.22.4 DOCA Libraries 1800
6.22.5 Compiling the Application 1801
6.22.5.1 Compiling All Applications 1801
6.22.5.2 Recompiling Only the Current Application 1801
6.22.6 Running the Application 1802
6.22.6.1 Prerequisites 1802
6.22.6.2 Application Execution 1802
6.22.6.3 Command Line Flags 1803
6.22.6.3.1 General Flags 1803
6.22.6.3.2 Program Flags 1803
6.22.6.4 Supported Commands 1804
6.22.6.5 Troubleshooting 1807
6.22.7 Application Code Flow 1807
6.22.8 References 1808
6.23 DOCA Accelerated UPF Application Guide 1808
6.23.1 Terms and Abbreviations 1808
6.23.2 Introduction 1809
6.23.3 System Design 1811
6.23.4 Application Architecture 1811
6.23.4.1 Key Features in UPF-RAP 1812
6.23.4.2 SMF 1813
6.23.4.2.1 SMF Policy Example 1815
6.23.4.3 Flow Processing 1817
6.23.4.4 Test Environment and Setup 1817
6.23.5 DOCA Libraries 1818
6.23.6 Dependencies 1818
6.23.7 Compiling the Application 1818

6.23.7.1 Compiling All Applications 1819
6.23.7.2 Compiling Only the Current Application 1819
6.23.8 Running the Application 1819
6.23.8.1 Runtime Prerequisites 1819
6.23.8.2 Application Execution 1820
6.23.8.3 Command Line Flags 1821
6.23.8.3.1 General Flags 1821
6.23.8.3.2 Program Flags 1821
6.23.8.4 Troubleshooting 1822
6.23.9 References 1822
6.24 DOCA UROM RDMO Application Guide 1822
6.24.1 Introduction 1822
6.24.2 System Design 1823
6.24.2.1 Bootstrap Procedure 1824
6.24.2.2 Memory Management 1824
6.24.2.3 RDMO UROM Worker Operation 1824
6.24.3 Architecture 1825
6.24.3.1 UROM RDMO Worker Component 1826
6.24.3.1.1 Init 1827
6.24.3.1.2 RQ Create 1827
6.24.3.1.3 RQ Destroy 1828
6.24.3.1.4 MR Register 1828
6.24.3.1.5 MR Deregister 1829
6.24.3.2 Command Format 1829
6.24.3.2.1 Append 1830
6.24.3.2.2 Flush 1831
6.24.3.2.3 Scatter 1832
6.24.4 DOCA Libraries 1832
6.24.5 Compiling the Application 1832
6.24.5.1 Compiling All Applications 1833
6.24.5.2 Compiling Only the Current Application 1833
6.24.6 Running the Application 1834
6.24.6.1 Host Application Execution 1834
6.24.6.2 RDMO DPU Plugin Component 1834
6.24.6.3 Command Line Flags 1835
6.24.6.3.1 General Flags 1835
6.24.6.3.2 Program Flags 1835
6.24.6.4 Troubleshooting 1835
6.24.7 Application Code Flow 1836

6.24.8 References ... 1837
6.25 DOCA YARA Inspection Application Guide ... 1837
6.25.1 Introduction ... 1837
6.25.2 System Design ... 1837
6.25.3 Application Architecture ... 1838
6.25.4 DOCA Libraries ... 1839
6.25.5 Limitations ... 1839
6.25.6 Compiling the Application ... 1840
6.25.6.1 Compiling All Applications ... 1840
6.25.6.2 Compiling Only the Current Application ... 1840
6.25.7 Running the Application ... 1841
6.25.7.1 Prerequisites ... 1841
6.25.7.2 Application Execution ... 1842
6.25.7.3 Command Line Flags ... 1843
6.25.7.3.1 General Flags ... 1843
6.25.7.3.2 Program Flags ... 1844
6.25.7.4 Troubleshooting ... 1844
6.25.8 Application Code Flow ... 1844
6.25.9 References ... 1846
6.26 DOCA Pipeline Language Runtime Controller Gateway SHM Application Guide ... 1846
6.26.1 Introduction ... 1846
6.26.2 System Design ... 1846
6.26.3 Architecture ... 1847
6.26.3.1 DPL Program (gateway_shm.p4) ... 1848
6.26.3.1.1 P4 Tables ... 1850
6.26.3.1.2 Direct Counters ... 1850
6.26.3.2 Control Application (gateway.entries.json) ... 1850
6.26.3.3 Pipeline ... 1851
6.26.4 DOCA Libraries ... 1852
6.26.5 Compiling the Application ... 1852
6.26.5.1 Prerequisites ... 1853
6.26.5.2 Compiling the Application ... 1852
6.26.6 Running the Application ... 1854
6.26.6.1 Prerequisites ... 1854
6.26.6.2 Application Execution ... 1854
6.26.7 Application Code Flow ... 1854

6.26.8 References ... 1856
6.27 DOCA DPU GPU Remote Offload Application Guide ... 1856
6.27.1 Introduction ... 1856
6.27.2 Applications ... 1856
6.27.2.1 doca_dpu_gpu_remote_offload_server ... 1857
6.27.2.2 doca_dpu_gpu_remote_offload_orchestrator ... 1857
6.27.2.3 doca_dpu_gpu_remote_offload_client ... 1857
6.27.3 Architecture ... 1857
6.27.3.1 doca_dpu_gpu_remote_offload_server ... 1857
6.27.3.1.1 Control Phase ... 1858
6.27.3.1.2 TCP Server Phase ... 1858
6.27.3.2 doca_dpu_gpu_remote_offload_orchestrator ... 1858
6.27.3.2.1 Control Phase ... 1859
6.27.3.2.2 GPU Processing Phase ... 1859
6.27.3.3 doca_dpu_gpu_remote_offload_client ... 1859
6.27.4 DOCA Libraries ... 1860
6.27.4.1 Server ... 1860
6.27.4.2 Orchestrator ... 1860
6.27.4.3 Client ... 1860
6.27.5 Compiling the Applications ... 1860
6.27.5.1 Compiling All Applications ... 1860
6.27.5.2 Compiling Only DPU GPU Remote Offload Applications ... 1861
6.27.6 Running the Applications ... 1861
6.27.6.1 Server ... 1861
6.27.6.1.1 Application Usage Instructions ... 1861
6.27.6.1.2 CLI Example of the Application ... 1862
6.27.6.1.3 Command Line Flags ... 1862
6.27.6.2 Orchestrator ... 1863
6.27.6.2.1 Application Usage Instructions ... 1863
6.27.6.2.2 CLI Example of the Application ... 1863
6.27.6.2.3 Command Line Flags ... 1864
6.27.6.3 Client ... 1865
6.27.6.3.1 Application Usage Instructions ... 1865
6.27.6.3.2 CLI Example of the Application ... 1865
6.27.6.3.3 Command Line Flags ... 1866
6.28 DOCA Timescale Translation Application Guide ... 1867
6.28.1 Introduction ... 1867
6.28.2 System Design ... 1868
6.28.3 Application Architecture ... 1869

6.28.3.1 Event Sequence ... 1869
6.28.3.2 Clock Correlation ... 1869
6.28.3.2.1 Host and Arm Core ... 1870
6.28.3.2.2 DPA Subsystem ... 1870
6.28.3.3 Output Logging ... 1870
6.28.4 DOCA Libraries ... 1870
6.28.5 Compiling the Application ... 1870
6.28.5.1 Compiling All Applications ... 1871
6.28.5.2 Compiling Only the Current Application ... 1871
6.28.5.2.1 Option 1: Command Line Configuration ... 1871
6.28.5.2.2 Option 2: Configuration File ... 1872
6.28.6 Running the Application ... 1872
6.28.6.1 Application Execution ... 1872
6.28.6.1.1 Running on x86 Host ... 1872
6.28.6.1.2 Running on BlueField DPU ... 1873
6.28.6.2 Command Line Flags ... 1873
6.28.6.2.1 General Flags ... 1873
6.28.6.2.2 Host Program Flags ... 1874
6.28.6.2.3 DPU Program Flags ... 1874
6.28.6.3 Troubleshooting ... 1874
6.28.7 Application Code Flow ... 1875
6.28.7.1 Common ... 1875
6.28.7.2 Host (x86) App ... 1875
6.28.7.3 DPU App ... 1876
6.28.8 References ... 1877
6.29 DOCA Virtio-net Emulation Application Guide ... 1877
6.29.1 1. Introduction ... 1877
6.29.1.1 1.1. System Design ... 1878
6.29.1.1.1 1.1.1. Host Side (Virtio-net Client Driver) ... 1878
6.29.1.1.2 1.1.2. BlueField DPU (DOCA Virtio-net Emulation Application) ... 1878
6.29.1.1.3 1.1.3. Data Path Provider ... 1879
6.29.1.2 1.2. End-to-End Flow ... 1879
6.29.2 2. Application Architecture ... 1880
6.29.2.1 2.1. Core Components ... 1880
6.29.2.1.1 2.1.1. Virtio-net Device Model ... 1880
6.29.2.1.2 2.1.2. Virtio Queues ... 1881
6.29.2.1.3 2.1.3. PCIe TLP Emulation ... 1881
6.29.2.1.4 2.1.4. PCI Switch Topology ... 1882
6.29.2.1.5 2.1.5. SR-IOV ... 1882
6.29.2.1.6 2.1.6. Threading & latency model (core invariant) ... 1882

6.29.2.2 2.2. Initialization Process ... 1883
6.29.2.3 2.3. Reset and Shutdown ... 1883
6.29.3 3. DOCA Libraries ... 1884
6.29.4 4. Dependencies ... 1884
6.29.5 5. Compiling the Application ... 1884
6.29.5.1 5.1. Compiling All Applications ... 1884
6.29.5.2 5.2. Compiling Only the Current Application ... 1884
6.29.5.3 5.3. Build Outputs ... 1885
6.29.6 6. Running the Application ... 1885
6.29.6.1 6.1. Prerequisites ... 1885
6.29.6.2 6.2. Application Usage ... 1885
6.29.6.2.1 6.2.1. Example Command Line Execution ... 1886
6.29.6.2.2 6.2.2. Runtime Commands ... 1886
6.29.6.2.3 6.2.3. Verifying the Device (Host Side) ... 1887
6.29.6.3 6.3. Command Line Flags ... 1887
6.29.6.3.1 6.3.1. General Flags ... 1887
6.29.6.3.2 6.3.2. Program Flags ... 1887
6.29.6.3.3 6.3.3. Optional Features Flags ... 1888
6.29.6.4 6.4. Troubleshooting ... 1888
6.29.7 7. Application Code Flow ... 1888
6.29.7.1 7.1. Framework Initialization ... 1888
6.29.7.1.1 7.1.1. Initialization Flow ... 1888
6.29.7.1.2 7.1.2. Key Points ... 1889
6.29.7.2 7.2. Resource Management ... 1889
6.29.7.2.1 7.2.1. Resource Management Flow ... 1889
6.29.7.2.2 7.2.2. Key Points ... 1890
6.29.7.3 7.3. PCI Switch Topology and Device Configuration ... 1890
6.29.7.3.1 7.3.1. PCI Switch Topology Flow ... 1890
6.29.7.3.2 7.3.2. Key Points ... 1891
6.29.7.4 7.4. Virtio-net Controller ... 1891
6.29.7.4.1 7.4.1. Controller Initialization Flow ... 1891
6.29.7.4.2 7.4.2. Virtio Queue Lifecycle State Machine ... 1892
6.29.7.4.3 7.4.3. Key Points ... 1892
6.29.7.5 7.5. I/O Request Processing ... 1892
6.29.7.5.1 7.5.1. I/O Request Processing Flow ... 1892
6.29.7.5.2 7.5.2. Key Points ... 1893
6.29.7.6 7.6. Device Lifecycle Management ... 1893
6.29.7.6.1 7.6.1. Device Lifecycle State Machine ... 1893
6.29.7.6.2 7.6.2. Hotplug Management Flow ... 1893
6.29.7.6.3 7.6.3. Key Points ... 1894

6.29.7.7 7.7. SR-IOV 1894
6.29.7.7.1 7.7.1. 1 Model & rationale 1894
6.29.7.7.2 7.7.2. 2 Resources & validation 1895
6.29.7.7.3 7.7.3. 3 VF lifecycle 1895
6.29.7.7.4 7.7.4. Per-VF FLR 1895
6.29.7.7.5 7.7.5. SR-IOV PCIe Capability (Host View) 1895
6.29.7.7.6 7.7.6. VF datapath & VM passthrough 1896
6.29.7.7.7 7.7.7. Constraints & expected logs 1896
6.29.7.8 7.8. Cleanup and Shutdown 1897
6.29.7.8.1 7.8.1. Cleanup and Shutdown Flow 1897
6.29.7.8.2 7.8.2. Phase 1 – Worker and Device Cleanup 1897
6.29.7.8.3 7.8.3. Phase 2 – Reverse-Order Resource Cleanup 1898
6.29.7.8.4 7.8.4. Key Points 1898
6.29.7.9 7.9. Live Upgrade (LU) Mode 1898
6.29.7.10 7.10. Crash Recovery 1899
6.29.8 8. Virtio-net Operations Support Summary 1900
6.29.9 9. Virtio Feature Bits Support Summary 1900
6.29.10 10. Developer Notes 1901
6.30 DOCA Virtio-blk Emulation Application Guide 1901
6.30.1 Introduction 1901
6.30.1.1 System Design 1902
6.30.1.1.1 Host Side (Virtio-blk Client Driver) 1902
6.30.1.1.2 BlueField DPU (DOCA Virtio-blk Emulation Application) 1902
6.30.1.1.3 Data Path Provider (DPA or DPU) 1902
6.30.1.2 End-to-End Flow 1903
6.30.2 Application Architecture 1903
6.30.2.1 Core Components 1903
6.30.2.1.1 Virtio-blk Device Model 1903
6.30.2.1.2 Virtio Queues 1903
6.30.2.1.3 PCIe TLP Emulation 1903
6.30.2.1.4 PCI Switch Topology 1904
6.30.2.2 Initialization Process 1904
6.30.2.3 Reset and Shutdown 1904
6.30.3 DOCA Libraries 1905
6.30.4 Dependencies 1905
6.30.5 Compiling the Application 1905
6.30.5.1 Compiling All Applications 1906
6.30.5.2 Compiling Only the Current Application 1906
6.30.5.3 Build Outputs 1906

6.30.6 Running the Application ... 1907
6.30.6.1 Prerequisites ... 1907
6.30.6.2 Application Execution ... 1907
6.30.6.2.1 Application Usage ... 1907
6.30.6.2.2 Example Command Line Execution ... 1908
6.30.6.2.3 Runtime Commands ... 1908
6.30.6.2.4 Verifying the Device (Host Side) ... 1909
6.30.6.3 Command Line Flags ... 1909
6.30.6.3.1 General Flags ... 1909
6.30.6.3.2 Program Flags ... 1909
6.30.6.4 Troubleshooting ... 1910
6.30.7 Application Code Flow ... 1911
6.30.7.1 Framework Initialization ... 1911
6.30.7.1.1 Initialization Flow ... 1911
6.30.7.1.2 Key Points ... 1912
6.30.7.2 Resource Management ... 1913
6.30.7.2.1 Resource Management Flow ... 1913
6.30.7.2.2 Key Points ... 1915
6.30.7.3 PCI Switch Topology and Device Configuration ... 1915
6.30.7.3.1 PCI Switch Topology Flow ... 1915
6.30.7.3.2 Key Points ... 1917
6.30.7.4 Virtio-blk Controller ... 1917
6.30.7.4.1 Controller Initialization Flow ... 1917
6.30.7.4.2 Virtio Queue Lifecycle State Machine ... 1919
6.30.7.4.3 Key Points ... 1919
6.30.7.5 I/O Request Processing ... 1920
6.30.7.5.1 I/O Request Processing Flow ... 1920
6.30.7.5.2 Key Points ... 1921
6.30.7.6 Device Lifecycle Management ... 1921
6.30.7.6.1 Device Lifecycle State Machine ... 1921
6.30.7.6.2 Hotplug Management Flow ... 1922
6.30.7.6.3 Key Points ... 1923
6.30.7.7 Cleanup and Shutdown ... 1923
6.30.7.7.1 Cleanup and Shutdown Flow ... 1923
6.30.7.7.2 Key Points ... 1925
6.30.7.8 SRIOV mode ... 1925
6.30.7.8.1 Resource management ... 1925
6.30.7.8.2 Init Flow ... 1926
6.30.7.8.3 runtime model ... 1928
6.30.7.8.4 Final teardown ... 1929
6.30.8 Virtio-blk Operations Support Summary ... 1929

6.30.8.1 Virtio Feature Bits Support Summary 1929
6.31 DOCA Telemetry Applications 1930
6.31.1 DOCA Telemetry OpenTelemetry Application 1930
6.31.1.1 Introduction 1930
6.31.1.2 System Design 1930
6.31.1.3 Application Architecture 1931
6.31.1.3.1 Supported Diag Data IDs 1931
6.31.1.3.2 OpenTelemetry Labelling 1933
6.31.1.4 Libraries 1933
6.31.1.4.1 DOCA Libraries 1933
6.31.1.4.2 OpenTelemetry C++ SDK 1933
6.31.1.5 Compiling the Application 1934
6.31.1.5.1 Compiling All Applications 1934
6.31.1.5.2 Compiling Only the Current Application 1934
6.31.1.6 Running the Application 1935
6.31.1.6.1 Application Execution 1935
6.31.1.6.2 Command Line Flags 1936
6.31.1.6.3 Troubleshooting 1937
6.31.1.7 Application Code Flow 1937
6.31.1.7.1 Phase 1: Argument Parsing 1937
6.31.1.7.2 Phase 2: OpenTelemetry Initialization 1937
6.31.1.7.3 Phase 3: Main Execution Loop 1938
6.31.1.7.4 Phase 4: Teardown and Cleanup 1938
6.31.1.8 References 1938
6.31.2 DOCA Telemetry Traceback Application 1939
6.31.2.1 Introduction 1939
6.31.2.2 System Design 1939
6.31.2.3 Application Architecture 1939
6.31.2.3.1 Traceback Configuration File 1940
6.31.2.3.2 Rule Definitions 1941
6.31.2.4 DOCA Libraries 1942
6.31.2.5 Compiling the Application 1943
6.31.2.5.1 Compiling All Applications 1943
6.31.2.5.2 Compiling Only the Current Application 1943
6.31.2.6 Running the Application 1944
6.31.2.6.1 Application Execution 1944
6.31.2.6.2 Command Line Flags 1944
6.31.2.6.3 Troubleshooting 1945
6.31.2.7 Application Code Flow 1946
6.31.2.7.1 Phase 1: Argument Parsing 1946
6.31.2.7.2 Phase 2: Component Initialization 1946

6.31.2.7.3 Phase 3: Main Execution Loop 1947
6.31.2.8 References 1948
7 DOCA Tools 1949
7.1 DOCA Bench 1949
7.1.1 Introduction 1949
7.1.2 Feature Overview 1949
7.1.3 Installation 1950
7.1.3.1 Prerequisites 1950
7.1.3.2 Granular Build Support 1950
7.1.4 Operating Modes 1950
7.1.5 Throughput Measurements 1950
7.1.5.1 Latency Measurements 1950
7.1.5.1.1 Bulk Latency 1951
7.1.5.1.2 Precision Latency 1952
7.1.6 Core Principles 1952
7.1.6.1 Host or BlueField Arm Execution 1952
7.1.6.2 Pipelines 1952
7.1.6.3 Warm-up Period 1952
7.1.6.4 Defaults 1953
7.1.6.5 Optimizing Performance 1953
7.1.7 Supported BlueField Feature Matrix 1953
7.1.8 Remote Operations 1954
7.1.9 CPU Core and Thread Selection 1955
7.1.10 Device Selection 1956
7.1.10.1 BlueField Devices 1956
7.1.10.2 GPU Devices 1956
7.1.11 Input Data Selection and Sizing of Jobs 1956
7.1.11.1 Input Data Selection 1956
7.1.11.1.1 File 1956
7.1.11.1.2 File Sets 1956
7.1.11.1.3 Random Data 1957
7.1.11.2 Job Sizing 1957
7.1.12 Controlling Test Duration 1957
7.1.12.1 Limit to Specific Number of Seconds 1957
7.1.12.2 Limited Through Total Number of Jobs 1957
7.1.13 GGA-specific Attributes 1957
7.1.14 Command-line Parameters 1958
7.1.14.1 CPU Core and Thread Count Configuration 1958

7.1.14.1.1 --core-mask ... 1958
7.1.14.1.2 --core-list ... 1959
7.1.14.1.3 --core-count ... 1959
7.1.14.1.4 --threads-per-core -t ... 1959
7.1.14.2 Device Configuration ... 1959
7.1.14.2.1 --device -A ... 1959
7.1.14.2.2 --representor -R ... 1959
7.1.14.3 Input Data and Buffer Size Configuration ... 1959
7.1.14.3.1 --data-provider -I ... 1960
7.1.14.3.2 --data-provider-job-count ... 1961
7.1.14.3.3 --data-provider-input-file ... 1961
7.1.14.3.4 --uniform-job-size ... 1961
7.1.14.3.5 --job-output-buffer-size ... 1961
7.1.14.3.6 --input-cwd -i ... 1962
7.1.14.4 Test Execution Control ... 1963
7.1.14.4.1 --mode ... 1963
7.1.14.4.2 --latency-bucket-range ... 1963
7.1.14.5 Blocking Mode ... 1964
7.1.14.5.1 --use-blocking-mode ... 1964
7.1.14.5.2 --record-cpu-usage ... 1964
7.1.14.6 Execution Limits ... 1965
7.1.14.6.1 --run-limit-seconds -s ... 1965
7.1.14.6.2 --run-limit-jobs -J ... 1965
7.1.14.6.3 --run-limit-bytes -b ... 1965
7.1.14.7 Batching Support ... 1965
7.1.14.7.1 --batch-mode ... 1965
7.1.14.7.2 --batch-size ... 1966
7.1.14.8 Gather/Scatter Support ... 1966
7.1.14.8.1 --gather-value ... 1966
7.1.14.8.2 --scatter-value ... 1966
7.1.14.9 Stats Output ... 1966
7.1.14.9.1 --rt-stats-interval ... 1966
7.1.14.9.2 --csv-output-file ... 1967
7.1.14.9.3 --csv-stats ... 1967
7.1.14.9.4 --csv-append-mode ... 1967
7.1.14.9.5 --csv-separate-dynamic-values ... 1967
7.1.14.9.6 --enable-environment-information ... 1968
7.1.14.10 Remote Memory Testing ... 1968
7.1.14.10.1 --use-remote-input-buffers ... 1968
7.1.14.10.2 --use-remote-output-buffers ... 1968
7.1.15 Network Options ... 1968

7.1.15.1 --mtu-size ... 1968
7.1.15.2 --receive-queue-size ... 1968
7.1.15.3 --send-queue-size ... 1969
7.1.15.4 DOCA Lib Configuration Options ... 1969
7.1.15.4.1 --task-pool-size ... 1969
7.1.15.5 Pipeline Configuration ... 1969
7.1.15.5.1 --pipeline-steps ... 1969
7.1.15.5.2 --attribute ... 1970
7.1.15.5.3 --warm-up-jobs ... 1971
7.1.15.6 Companion Configuration ... 1971
7.1.15.6.1 --companion-connection-string ... 1971
7.1.15.6.2 --companion-core-list ... 1972
7.1.15.6.3 --companion-core-mask ... 1972
7.1.15.7 Sweep Tests ... 1972
7.1.15.7.1 --sweep ... 1972
7.1.15.8 Queries ... 1973
7.1.15.8.1 Device Capabilities ... 1973
7.1.15.8.2 Supported Sweep Attributes ... 1974
7.1.16 Test Memory Footprint ... 1974
7.1.17 GPUNetIO Benchmarks ... 1975
7.1.17.1 Controlling CUDA Threads ... 1975
7.1.17.2 GPUNetIO Eth ... 1976
7.1.18 DOCA Bench Sample Invocations ... 1976
7.1.18.1 Overview ... 1976
7.1.18.2 DOCA Eth Receive Sample ... 1976
7.1.18.2.1 Command Line ... 1977
7.1.18.2.2 Results Output ... 1977
7.1.18.2.3 Results Overview ... 1978
7.1.18.3 DOCA Eth Send Sample ... 1978
7.1.18.3.1 Command Line ... 1978
7.1.18.3.2 Results Output ... 1978
7.1.18.3.3 Results Overview ... 1979
7.1.18.4 Host-side AES-GCM Decrypt Sample ... 1979
7.1.18.4.1 Command Line ... 1980
7.1.18.4.2 Results Output ... 1980
7.1.18.4.3 Results Overview ... 1980
7.1.18.5 BlueField-side AES-GCM Encrypt Sample ... 1980
7.1.18.5.1 Command Line ... 1981
7.1.18.5.2 Results Output ... 1981
7.1.18.5.3 Results Overview ... 1981
7.1.18.6 Host-side AES-GCM Encrypt and Decrypt Sample ... 1982

7.1.18.6.1 Command Line ... 1982
7.1.18.6.2 Results Output ... 1982
7.1.18.6.3 Results Overview ... 1982
7.1.18.7 Host-side SHA with CSV Output File Sample ... 1982
7.1.18.7.1 Command Line ... 1982
7.1.18.7.2 Results Output ... 1983
7.1.18.7.3 Results Overview ... 1983
7.1.18.8 Host-side SHA with CSV Appended Output File Sample ... 1983
7.1.18.8.1 Command Line ... 1984
7.1.18.8.2 Results Output ... 1984
7.1.18.8.3 Results Overview ... 1985
7.1.18.9 BlueField-side SHA with Transient Statistics Sample ... 1985
7.1.18.9.1 Command Line ... 1986
7.1.18.9.2 Results Output ... 1986
7.1.18.9.3 Results Overview ... 1987
7.1.18.10 Host-side Local DMA with Core Sweep Sample ... 1987
7.1.18.10.1 Command Line ... 1987
7.1.18.10.2 Results Overview ... 1987
7.1.18.10.3 Results Overview ... 1988
7.1.18.11 Host-side Local DMA with Job Size Sweep Sample ... 1989
7.1.18.11.1 Command Line ... 1989
7.1.18.11.2 Results Overview ... 1989
7.1.18.11.3 Results Overview ... 1990
7.1.18.12 BlueField-side Remote DMA Sample ... 1990
7.1.18.12.1 Command Line ... 1990
7.1.18.12.2 Results Overview ... 1991
7.1.18.12.3 Results Overview ... 1991
7.1.18.13 Compress BlueField-side Sample ... 1991
7.1.18.13.1 Command Line ... 1991
7.1.18.13.2 Result Output ... 1991
7.1.18.13.3 Results Overview ... 1991
7.1.18.14 BlueField-side Decompress LZ4 Sample ... 1992
7.1.18.14.1 Command Line ... 1992
7.1.18.14.2 Results Output ... 1992
7.1.18.14.3 Results Comment ... 1992
7.1.18.15 Host-side EC Creation in Bulk Latency Mode Sample ... 1992
7.1.18.15.1 Command Line ... 1992
7.1.18.15.2 Results Output ... 1993
7.1.18.15.3 Results Comment ... 1993
7.1.18.16 BlueField-side EC Creation in Precision Latency Mode Sample ... 1993
7.1.18.16.1 Command Line ... 1993

7.1.18.16.2 Results Output ... 1993
7.1.18.16.3 Results Comment ... 1993
7.1.18.17 Comch Consumer from Host Side Sample ... 1993
7.1.18.17.1 Command Line ... 1993
7.1.18.17.2 Results Output ... 1994
7.1.18.17.3 Results Comment ... 1994
7.1.18.18 Host-side Comch Producer Sample ... 1994
7.1.18.18.1 Command Line ... 1994
7.1.18.18.2 Results Overview ... 1994
7.1.18.18.3 Results Comment ... 1994
7.1.18.19 Host-side RDMA Send Sample ... 1994
7.1.18.19.1 Command Line ... 1995
7.1.18.19.2 Results Output ... 1995
7.1.18.19.3 Results Comment ... 1995
7.1.18.20 Host-side RDMA Receive Sample ... 1995
7.1.18.20.1 Command Line ... 1995
7.1.18.20.2 Results Output ... 1995
7.1.18.20.3 Results Overview ... 1996
7.2 DOCA Capabilities Print Tool ... 1996
7.2.1 Introduction ... 1996
7.2.2 Prerequisites ... 1996
7.2.3 Description ... 1996
7.2.4 Execution ... 1996
7.3 DOCA Comm Channel Admin Tool ... 2005
7.3.1 Introduction ... 2005
7.3.2 Prerequisites ... 2005
7.3.3 Description and Execution ... 2005
7.3.3.1 Sample Output from BlueField Arm ... 2006
7.3.3.2 Sample Output from x86 ... 2006
7.4 DPA Tools ... 2007
7.4.1 Introduction ... 2007
7.4.2 List of Tools ... 2008
7.4.3 DOCA DPACC Compiler ... 2008
7.4.3.1 Introduction ... 2008
7.4.3.1.1 Offloading Work on DPA ... 2009
7.4.3.1.2 DPACC Predefined Macros ... 2009
7.4.3.1.3 Writing DPA Applications ... 2009
7.4.3.2 Prerequisites ... 2011
7.4.3.3 Description ... 2011

7.4.3.3.1 DPACC Inputs and Outputs ... 2011
7.4.3.3.2 DPACC Trajectory ... 2014
7.4.3.3.3 Modes of Operation ... 2014
7.4.3.4 Execution ... 2015
7.4.3.4.1 Mandatory Arguments ... 2015
7.4.3.4.2 Commonly Used Arguments ... 2016
7.4.3.4.3 DPA Hardware Architectures ... 2017
7.4.3.4.4 Architecture Macros ... 2018
7.4.3.4.5 LTO Usage Guidelines ... 2019
7.4.3.4.6 Optimized memcpy ... 2019
7.4.3.4.7 Deprecated Features ... 2020
7.4.3.4.8 Examples ... 2020
7.4.3.4.9 DPA Compiler Usage ... 2023
7.4.3.5 Release Notes ... 2025
7.4.3.5.1 Changes in 2.3.0 Release ... 2025
7.4.4 Single Point of Resource Distribution ... 2025
7.4.4.1 Introduction ... 2025
7.4.4.1.1 Key Characteristics ... 2026
7.4.4.1.2 Integration Requirements ... 2026
7.4.4.1.3 Glossary ... 2026
7.4.4.2 Preparation and Description ... 2027
7.4.4.2.1 System Requirements ... 2027
7.4.4.2.2 EUs and Cores ... 2027
7.4.4.2.3 Partition Concept ... 2027
7.4.4.2.4 Group Concept ... 2028
7.4.4.3 SPRD ... 2028
7.4.4.3.1 dpa-resource-mgmt Utility ... 2028
7.4.4.3.2 Format of Input YAML File ... 2032
7.4.4.3.3 Output YAML files ... 2036
7.4.4.4 SPRD Configuration and Management Flow ... 2037
7.4.4.4.1 Gather Application Requirements ... 2037
7.4.4.4.2 Map Devices to VHCAs ... 2037
7.4.4.4.3 Inspect Core and EU Layout ... 2038
7.4.4.4.4 Create Partition Configuration ... 2038
7.4.4.4.5 Define Application Assignments ... 2038
7.4.4.4.6 Validate Configuration ... 2038
7.4.4.4.7 Handle Conflicts (If Any) ... 2038
7.4.4.4.8 Review the Resulting Setup ... 2038
7.4.4.4.9 Iterate (Optional) ... 2039
7.4.4.4.10 Distribute Output Files ... 2039
7.4.4.4.11 Revise as Needed ... 2039

7.4.4.5 Flex IO SDK API for SPRD Support 2039
7.4.4.5.1 Core API Functions 2039
7.4.4.5.2 Example of Using EU Resources 2040
7.4.4.6 Back-End Tool - dpaeumgmt 2041
7.4.4.6.1 Execution Unit Objects 2042
7.4.4.6.2 dpaeumgmt CLI Reference 2042
7.4.4.6.3 vHCAs and Partitions 2043
7.4.4.6.4 Known Limitations of dpaeumgmt 2044
7.4.5 DOCA DPA GDB Server Tool 2045
7.4.5.1 Introduction 2045
7.4.5.1.1 Glossary 2045
7.4.5.1.2 Package Contents 2045
7.4.5.1.3 Known Limitations 2046
7.4.5.2 DPA-specific Notes 2046
7.4.5.2.1 Token 2046
7.4.5.2.2 Connection on Application Launch 2046
7.4.5.2.3 Dummy Thread Concept 2046
7.4.5.2.4 Watchdog Issues 2046
7.4.5.3 Tool TCP Port and Execution Unit (EU) 2047
7.4.5.4 Debugging 2047
7.4.5.4.1 Preparation for Debug 2047
7.4.5.4.2 Start Debugging 2047
7.4.5.4.3 DPA-specific Debugging Techniques 2048
7.4.5.5 Error Reporting 2050
7.4.5.5.1 Tool Log Directory 2050
7.4.5.5.2 Verbosity Level of gdbserver 2050
7.4.5.6 Useful Info Regarding Work with GDB 2051
7.4.5.6.1 Command "directory" 2051
7.4.5.6.2 Core Dump Usage 2052
7.4.5.6.3 Debug of Optimized Code 2053
7.4.5.6.4 Disassembly of Advanced RISC-V Commands 2053
7.4.6 DOCA DPA PS Tool 2054
7.4.6.1 Introduction 2054
7.4.6.2 Command Flags and Arguments 2054
7.4.6.3 Example 2054
7.4.6.4 Known Limitations 2055
7.4.7 DOCA DPA Statistics Tool 2055
7.4.7.1 Introduction 2055
7.4.7.2 Collecting Performance Statistics Data 2055
7.4.7.3 Presenting Statistics List 2056
7.4.7.3.1 Examples 2057

7.4.7.4 Known Limitations 2057
7.4.8 DPA Application Security Attributes to Blob 2057
7.4.8.1 Overview 2057
7.4.8.2 Features 2058
7.4.8.3 Usage 2058
7.4.8.3.1 Examples 2058
7.4.8.4 Attributes YAML Format 2059
7.4.8.4.1 Example Attributes YAML 2059
7.4.8.4.2 Generated YAML Output 2059
7.4.8.5 Error Handling 2059
7.4.8.6 Binary Blob Format 2060
7.5 DOCA Flow Tune Tool 2060
7.5.1 Introduction 2060
7.5.1.1 Monitor Mode Overview 2061
7.5.1.2 Analyze Mode Overview 2061
7.5.1.3 Visualize Mode Overview 2062
7.5.1.3.1 Mermaid Graph 2062
7.5.2 Dependencies 2063
7.5.3 Prerequisites 2063
7.5.4 Execution 2063
7.5.4.1 Monitor Command 2064
7.5.4.1.1 CLI Examples 2065
7.5.4.2 Analyze Command 2065
7.5.4.2.1 CLI Examples 2066
7.5.4.3 Visualize Command 2066
7.5.4.3.1 CLI Examples 2067
7.5.5 Configuration 2068
7.5.5.1 Config File Default Values 2068
7.5.5.2 Custom Config File 2068
7.5.5.3 Overriding Config Values from CLI 2068
7.5.5.4 Common Configuration Values 2069
7.5.5.4.1 Output Directory 2069
7.5.5.4.2 Connection to DOCA Flow Tune Server 2069
7.5.6 Monitor Mode 2070
7.5.6.1 Hardware Counters 2070
7.5.6.2 Software Key Performance Indicators 2071
7.5.6.2.1 Action Resources Counters 2071
7.5.6.3 Configuration 2072
7.5.6.3.1 Counters Dumper 2072
7.5.6.3.2 Configuration File 2074

7.5.7 Analyze Mode 2077
7.5.7.1 Pipeline Export 2077
7.5.7.2 Packet Trace 2077
7.5.7.2.1 Requirements: 2077
7.5.7.2.2 Output data: 2077
7.5.7.2.3 Example: 2077
7.5.7.2.4 Know Issues: 2078
7.5.8 Visualize Mode 2078
7.5.8.1 Supported Pipe Types 2079
7.5.8.2 Viewing the Pipeline 2079
7.5.8.3 Visualization Layers 2079
7.5.8.3.1 Base Layer 2079
7.5.8.3.2 Resources Layer 2080
7.5.8.3.3 Cost Layer 2081
7.5.8.3.4 Critical Path Layer 2082
7.5.8.3.5 Distribution Layer 2083
7.5.8.3.6 Packet Trace Level 2084
7.5.8.3.7 Recommendations Layer 2086
7.5.9 Use Cases 2087
7.5.9.1 Actions Memory Calculation 2087
7.5.9.1.1 Measured Strategy 2087
7.5.9.1.2 Calculated Strategy 2087
7.5.9.2 Counter Observation 2087
7.5.9.3 Program Pipeline Visualization 2088
7.5.9.4 Identifying Bottle Neck Pipes 2088
7.5.10 Troubleshooting 2088
7.5.10.1 Telemetry fwctl driver is not loaded 2088
7.5.10.1.1 Error 2088
7.5.10.1.2 Solution 2088
7.5.10.2 Mermaid visualization in Visual Studio Code 2089
7.5.10.3 Limited Feature Set – Could Not Detect Running DOCA Flow Program 2089
7.5.10.3.1 Error 2089
7.5.10.3.2 Solution 2090
7.5.10.4 visualize --cost or --critical-path Not Working 2090
7.5.10.4.1 Error 2090
7.5.10.4.2 Solution 2090
7.5.11 Appendix – Configuration File Example 2091
7.6 DOCA Ngauge 2092
7.6.1 Introduction 2092

7.6.2 Prerequisites ... 2093
7.6.2.1 Installing Ngauge ... 2093
7.6.3 Description ... 2093
7.6.3.1 Parsing Output ... 2094
7.6.4 Execution ... 2096
7.7 DOCA Perftest ... 2097
7.7.1 Introduction ... 2098
7.7.2 Comparison with Legacy Perftest ... 2098
7.7.3 Setup and Dependencies ... 2099
7.7.3.1 RDMA Core (libibverbs) ... 2099
7.7.3.2 MPI (Multi-Node Scenarios) ... 2099
7.7.3.3 Optional DOCA SDK Components ... 2099
7.7.4 Point-To-Point Benchmarks ... 2099
7.7.5 Scenario-Based Benchmarks ... 2100
7.7.5.1 CLI Scenario Mode ... 2101
7.7.5.2 JSON Scenario Mode ... 2102
7.7.5.3 Results and Output ... 2103
7.7.6 Benchmark Results ... 2104
7.7.6.1 Bandwidth ... 2104
7.7.6.2 Latency ... 2104
7.7.6.2.1 Latency Modes ... 2104
7.7.6.2.2 Verb Compatibility ... 2105
7.7.6.2.3 Metrics Collected ... 2105
7.7.6.2.4 Measurement Notes ... 2106
7.7.7 Common Arguments and Use-Cases ... 2106
7.7.7.1 Traffic Directions ... 2106
7.7.7.1.1 Unidirectional Traffic (Default) ... 2107
7.7.7.1.2 Bidirectional Traffic ... 2107
7.7.7.1.3 Dual Unidirectional Traffic ... 2107
7.7.7.2 Traffic Patterns ... 2107
7.7.7.2.1 One-to-One (O2O) ... 2108
7.7.7.2.2 One-to-Many (O2M) ... 2108
7.7.7.2.3 Many-to-One (M2O) ... 2108
7.7.7.2.4 All-to-All (A2A) ... 2108
7.7.7.2.5 Bisection (B) ... 2111
7.7.7.2.6 Ring (R) ... 2111
7.7.7.2.7 Traffic Partitions ... 2111
7.7.7.3 Per-iteration-sync Flow (Lock-step Benchmarking) ... 2112
7.7.7.3.1 Logic and Implementation ... 2113

7.7.7.3.2 Configuration ... 2113
7.7.7.3.3 Data Validation Integration ... 2114
7.7.7.3.4 Bandwidth Measurement ... 2114
7.7.7.3.5 Limitations ... 2114
7.7.7.4 Peer Shifting ... 2114
7.7.7.5 Hostname, Device Name, and CUDA Device ID Ranged Selection ... 2115
7.7.7.5.1 Supported Syntax ... 2115
7.7.7.5.2 Expansion Logic ... 2115
7.7.7.5.3 CUDA Device ID Ranged Selection ... 2115
7.7.7.5.4 Duplicate Connections ... 2116
7.7.7.6 Multiprocess (Cores) ... 2116
7.7.7.7 Working with GPUs – Device Selection ... 2116
7.7.7.8 Working with GPUs – Memory Types ... 2117
7.7.7.8.1 Auto-Detection Mode (cuda_auto_detect) ... 2117
7.7.7.8.2 Standard CUDA Memory (cuda_peermem) ... 2117
7.7.7.8.3 DMA-BUF Memory (cuda_dmabuf) ... 2117
7.7.7.8.4 Data Direct Memory (cuda_data_direct) ... 2118
7.7.7.9 Memory Types ... 2118
7.7.7.9.1 Host Memory (host) ... 2118
7.7.7.9.2 Null Memory Region (nullmr) ... 2118
7.7.7.9.3 Device Memory (device) ... 2118
7.7.7.10 RDMA Drivers ... 2118
7.7.7.10.1 Atomic Verbs ... 2119
7.7.7.11 Auto-launching Remote Server ... 2119
7.7.7.12 QP Histogram ... 2120
7.7.7.13 Start Packet Sequence Number ... 2121
7.7.7.14 Data Validation ... 2121
7.7.7.14.1 Prerequisites and Constraints ... 2122
7.7.7.14.2 Output and Reporting ... 2122
7.7.7.15 Enhanced Connection Establishment ... 2122
7.7.7.15.1 High-Level Flow ... 2122
7.7.7.15.2 ECE Configuration ... 2123
7.7.7.15.3 Limitations and Constraints ... 2123
7.7.7.16 QP Hints ... 2123
7.7.7.16.1 Configuring QP Hints ... 2123
7.7.7.17 TPH ... 2123
7.7.7.18 Warmup Configuration ... 2124
7.7.7.19 End-margin ... 2124
7.7.7.20 Live Bandwidth Reporting ... 2125
7.7.7.21 QP Transport Timeout ... 2125
7.7.7.22 Multi-verb Operations ... 2125

7.7.8 Automation and CI Integration 2125
7.7.8.1 Performance Thresholds 2126
7.7.8.2 Non-interactive Mode 2126
7.7.8.3 Structured JSON Output to stdout 2126
7.7.8.4 Connection Matrix Report 2127
7.7.9 Running DOCA Perftest on BlueField-3 Devices 2127
7.7.9.1 MPI Network Configuration 2127
7.7.9.2 Traffic Originating from x86 Host (Server) 2127
7.7.9.3 Traffic Originating from BlueField (Arm Cores) 2127
7.7.10 SLURM 2128
7.7.11 GPUNetIO Engine (GPU-initiated RDMA) 2128
7.8 DOCA Pipeline Language Developer Tools 2129
7.8.1 Introduction 2129
7.8.2 Prerequisites 2129
7.8.3 Setup 2130
7.8.4 Support and Feedback 2130
7.8.5 DPL Nspect 2130
7.8.5.1 1. Introduction 2130
7.8.5.2 2. Release Notes 2130
7.8.5.2.1 2.1. New Features 2130
7.8.5.2.2 2.2. Fixes 2131
7.8.5.2.3 2.3. Other 2131
7.8.5.3 3. DPL Nspect Execution 2131
7.8.5.3.1 3.1. Prerequisites and Configuration 2131
7.8.5.3.2 3.2. Command Usage 2132
7.8.5.4 4. DPL Nspect Commands and Flags 2132
7.8.5.4.1 4.1. system-info 2133
7.8.5.4.2 4.2. devices 2133
7.8.5.4.3 4.3. tables 2134
7.8.5.4.4 4.4. query 2135
7.8.5.4.5 4.5. externs 2136
7.8.5.4.6 4.6. graph 2138
7.8.5.4.7 4.7. debug 2140
7.8.5.4.8 4.8. perf 2140
7.8.5.4.9 4.9. hello_packet.p4 2142
7.8.6 DPL Debugger 2143
7.8.6.1 1. Introduction 2144
7.8.6.2 2. Launching the Debugger 2144
7.8.6.2.1 2.1. Option A: Live Debug Session 2144

7.8.6.2.2 2.2. Option B: Pre-recorded Debug Session ... 2145
7.8.6.3 3. Setting Source Directories ... 2146
7.8.6.4 4. Debug Workflow ... 2146
7.8.6.4.1 4.1. Enabling Debug Points ... 2146
7.8.6.4.2 4.2. Inspecting and Editing Packets ... 2147
7.8.6.4.3 4.3. Advancing the Pipeline ... 2148
7.8.6.4.4 4.4. Example Debugging Scenario ... 2149
7.8.6.5 5. UI Reference: Widgets ... 2151
7.8.6.5.1 5.1. Packet List ... 2151
7.8.6.5.2 5.2. Packet Trace ... 2152
7.8.6.5.3 5.3. Filter ... 2152
7.8.6.5.4 5.4. Variables ... 2153
7.8.6.5.5 5.5. Packet State ... 2153
7.8.6.5.6 5.6. Packet Dissector ... 2154
7.8.6.5.7 5.7. Hex Dump ... 2155
7.8.6.5.8 5.8. Source Code ... 2156
7.8.6.5.9 5.9. Pipeline ... 2158
7.8.6.5.10 5.10. Parser ... 2159
7.8.6.5.11 5.11. Log ... 2160
7.8.6.6 6. Release Notes ... 2161
7.8.6.6.1 6.1. New Features ... 2161
7.8.6.6.2 6.2. Bug Fixes ... 2161
7.8.7 DPL Admin ... 2161
7.8.7.1 Introduction ... 2161
7.8.7.2 Options and Commands ... 2162
7.8.7.2.1 config ... 2163
7.8.7.2.2 set-counter-cache-timeout ... 2164
7.8.7.2.3 set-log-level ... 2165
7.8.7.2.4 get-log-full ... 2165
7.8.7.2.5 get-log-recent ... 2165
7.8.7.2.6 hws-dump ... 2166
7.8.7.2.7 hca-capabilities ... 2167
7.8.7.2.8 nv-config ... 2167
7.8.7.2.9 set-idle-timeout-polling-interval ... 2167
7.8.8 P4 Runtime Controller ... 2168
7.8.8.1 Introduction ... 2168
7.8.8.2 Basic Information and Inspection ... 2168
7.8.8.3 Working with P4 Table Entries ... 2169
7.8.8.3.1 Specifying Match Keys ... 2169
7.8.8.3.2 Adding Entries ... 2169
7.8.8.3.3 Reading Entries ... 2169

7.8.8.3.4 Deleting Entries ... 2170
7.8.8.3.5 Modifying Entries ... 2170
7.8.8.3.6 Managing Default Entries ... 2171
7.8.8.4 Working with Counters ... 2171
7.8.8.4.1 Direct Counters ... 2171
7.8.8.4.2 Indirect Counters ... 2171
7.8.8.5 Working with Meters ... 2172
7.8.8.5.1 Direct Meters ... 2172
7.8.8.5.2 Indirect Meters ... 2172
7.8.8.6 Packet I/O and Notifications ... 2172
7.8.8.6.1 Packet In (Sniffing) ... 2172
7.8.8.6.2 Packet Out (Sending) ... 2172
7.8.8.6.3 Idle Timeout Notifications ... 2173
7.8.9 DPL Runtime Controller SDK ... 2173
7.8.9.1 Introduction ... 2173
7.8.9.1.1 Control Protocols ... 2173
7.8.9.1.2 Table Types ... 2174
7.8.9.1.3 SDK Libraries Overview ... 2174
7.8.9.2 Installation and System Setup ... 2179
7.8.9.2.1 SDK Packages and Requirements ... 2179
7.8.9.2.2 Installation Methods ... 2179
7.8.9.2.3 Hugepages Configuration ... 2180
7.8.9.3 DPL P4Runtime Controller Library ... 2181
7.8.9.4 DPL Runtime Controller Library ... 2183
7.8.9.5 Idle Timeout Entries ... 2191
7.8.9.5.1 Limitations ... 2191
7.8.9.5.2 Usage ... 2191
7.8.9.5.3 Handling of Stale Entries ... 2191
7.8.9.6 IPsec ... 2192
7.8.9.6.1 P4 Programming ... 2192
7.8.9.6.2 Usage ... 2192
7.8.9.7 DPL Runtime Controller Samples ... 2193
7.8.9.7.1 Running the Sample ... 2193
7.8.9.7.2 Samples ... 2194
7.9 DOCA PCC Counter Tool ... 2200
7.9.1 Introduction ... 2200
7.9.2 Prerequisites ... 2200
7.9.3 Description ... 2201
7.9.4 Execution ... 2201
7.10 DOCA Socket Relay ... 2202

7.10.1 Introduction ... 2202
7.10.2 Prerequisites ... 2204
7.10.3 Dependencies ... 2204
7.10.4 Execution ... 2204
7.10.5 Arg Parser DOCA Flags ... 2204
7.10.5.1 General Flags ... 2204
7.10.5.2 Program Flags ... 2205
7.11 DOCA Telemetry Utils ... 2206
7.11.1 Introduction ... 2206
7.11.2 Prerequisites ... 2206
7.11.2.1 Verifying the fwctl Driver ... 2206
7.11.2.2 Manually Loading the Driver ... 2206
7.11.2.3 Reinstalling the DOCA Host Package ... 2207
7.11.2.4 Installing DOCA Telemetry Utils ... 2207
7.11.3 Description ... 2207
7.11.3.1 Usage with Counter Name ... 2208
7.11.3.2 Usage with Data ID ... 2208
7.11.3.3 Checking Counter Support on a Device ... 2208
7.11.4 Execution ... 2209
7.12 DOCA doca-hugepages Tool ... 2209
7.12.1 Introduction ... 2209
7.12.2 Execution ... 2210
7.12.2.1 config ... 2210
7.12.2.2 reload ... 2210
7.12.2.3 remove ... 2210
7.12.2.4 show ... 2210
7.12.2.5 original-config ... 2211
7.12.2.6 Important Notes ... 2211
7.12.3 Incompatibility with GRUB-Based Hugepages Management ... 2211
7.12.4 Integration with Packages ... 2211
8 DOCA Services ... 2213
8.1 Introduction ... 2213
8.1.1 DOCA Services Summary and Quality Levels ... 2213
8.2 Development Lifecycle ... 2214
8.2.1 Development ... 2214
8.2.2 Containerization ... 2214
8.2.3 Profiling ... 2215

8.3 DOCA Container Deployment Guide ... 2216
8.3.1 1. Introduction ... 2216
8.3.2 2. Prerequisites ... 2216
8.3.3 3. Container Deployment ... 2217
8.3.3.1 3.1. Pull Container YAML Configurations ... 2217
8.3.3.2 3.2. Container-specific Instructions ... 2218
8.3.3.3 3.3. Structure of NGC Resource ... 2218
8.3.3.4 3.4. Spawn Container ... 2218
8.3.3.5 3.5. Review Container Deployment ... 2219
8.3.3.6 3.6. Stop Container ... 2220
8.3.4 4. Troubleshooting Common Errors ... 2220
8.3.4.1 4.1. Yaml Syntax ... 2220
8.3.4.2 4.2. Huge Pages ... 2221
8.3.5 5. Advanced Troubleshooting ... 2221
8.3.5.1 5.1. Manual Execution from Within Container - Debugging ... 2221
8.3.6 6. Air-gapped Container Deployment ... 2222
8.3.6.1 6.1. Pulling Container for Offline Deployment ... 2222
8.3.6.2 6.2. Importing Container Image ... 2222
8.3.6.3 6.3. Built-in Infrastructure Support ... 2223
8.3.7 7. DOCA Services for Host ... 2223
8.3.7.1 7.1. Docker Deployment ... 2223
8.3.8 8. Container Deployment Using DMS ... 2224
8.4 DOCA Platform Framework (DPF) ... 2225
8.4.1 Getting Started with DPF ... 2225
8.4.2 Why DPF ... 2226
8.4.3 Architecture ... 2226
8.4.3.1 Where DPF Fits ... 2227
8.4.4 Deployment Models ... 2227
8.4.5 Platform Support ... 2229
8.4.6 DPU Services Supported by DPF ... 2229
8.4.7 Service Function Chaining ... 2230
8.4.8 DPF Validated Reference Deployment Guides ... 2231
8.4.8.1 Host Trusted ... 2231
8.4.8.2 Zero Trust ... 2231
8.4.8.3 Operations and Monitoring ... 2232
8.4.9 NVIDIA Platform Integrations ... 2232
8.4.10 Next Steps ... 2232

8.5 DOCA Argus Service Guide 2232
8.5.1 Introduction 2233
8.5.2 Operation 2233
8.5.2.1 Monitored Activity and Relationships 2233
8.5.2.2 Events and Alerts 2234
8.5.3 Use Cases 2234
8.5.3.1 Runtime Workload Threat Detection 2234
8.5.3.2 AI Workload Discovery 2234
8.5.3.3 Runtime Observability 2235
8.5.4 Requirements 2235
8.5.5 Limitations 2235
8.5.6 Prerequisites 2235
8.5.7 Service Deployment 2236
8.5.8 Service Configuration 2236
8.5.8.1 Service 2236
8.5.8.2 DOCA Argus Configuration 2237
8.5.8.3 Per-system Configurations 2237
8.5.8.4 Collection 2237
8.5.8.5 Output 2237
8.5.9 Log Types 2238
8.5.9.1 Standard Output 2238
8.5.9.2 Debug Log Output 2238
8.5.9.3 Event Log Output 2238
8.5.9.4 Telemetry Output 2238
8.5.9.4.1 Fluent Bit 2238
8.5.9.4.2 Vector 2239
8.5.10 Reported Activities 2240
8.5.10.1 Naming Conventions 2240
8.5.10.2 Message Type 2240
8.5.10.3 Message Structure 2240
8.5.10.3.1 Schema 2240
8.5.10.3.2 Example JSON Message 2242
8.5.11 Alerts, Events, and System Activity Messages 2243
8.5.11.1 Events and Alerts 2243
8.5.11.1.1 Bash Shell 2243
8.5.11.1.2 Containers 2244
8.5.11.1.3 File Descriptors 2244
8.5.11.1.4 GPU Telemetry 2244
8.5.11.1.5 Kernel Modules 2245

8.5.11.1.6 Network Connections ... 2245
8.5.11.1.7 Process Memory ... 2246
8.5.11.1.8 Processes ... 2246
8.5.11.1.9 Runtime Manifest ... 2247
8.5.11.1.10 Threads ... 2247
8.5.11.2 System Activity Messages ... 2247
8.5.11.2.1 Service Lifecycle ... 2247
8.5.11.2.2 Host Initialization and OS Discovery ... 2248
8.5.11.2.3 OS Profile Processing ... 2248
8.5.11.2.4 Resource Limits ... 2249
8.5.11.2.5 Configuration and Data Collection ... 2249
8.5.12 Activity Data Reference ... 2249
8.5.12.1 Bash Shell Command Data ... 2249
8.5.12.2 Container Context ... 2250
8.5.12.3 File Descriptor Data ... 2250
8.5.12.4 GPU Telemetry Data ... 2251
8.5.12.5 Kernel Module Data ... 2251
8.5.12.6 Network Connection Data ... 2251
8.5.12.7 Process Data ... 2252
8.5.12.8 Process Memory Data ... 2253
8.5.12.9 Runtime Manifest Data ... 2253
8.5.12.9.1 Expected Behavior Data ... 2254
8.5.12.10 Thread Data ... 2254
8.6 DOCA BlueMan Service Guide ... 2255
8.6.1 Introduction ... 2255
8.6.2 Requirements ... 2255
8.6.2.1 Verifying DTS Status ... 2255
8.6.2.2 Verifying DPE Status ... 2256
8.6.3 Service Deployment ... 2256
8.6.3.1 DOCA Service on NGC ... 2256
8.6.3.2 Default Deployment – BlueField BSP ... 2256
8.6.3.2.1 Enabling BlueMan Service ... 2256
8.6.3.3 Verifying Deployment Success ... 2257
8.6.4 Collected Data ... 2257
8.6.5 Connecting to BlueMan Web Interface ... 2258
8.6.6 Troubleshooting ... 2259
8.7 DOCA Firefly Service Guide ... 2259
8.7.1 Introduction ... 2259
8.7.2 Requirements ... 2260

8.7.2.1 Firmware Version 2260
8.7.2.2 BlueField BSP Version 2261
8.7.2.3 Embedded Mode 2261
8.7.2.3.1 Configuring Firmware Settings on DPU for Embedded Mode 2261
8.7.2.3.2 Ensuring OVS Hardware Offload 2261
8.7.2.3.3 Helper Scripts 2261
8.7.2.3.4 Setting Up Network Interfaces for DPU Mode 2263
8.7.2.4 Separated Mode 2264
8.7.2.4.1 Configuring Firmware Settings on DPU for Separated Mode 2264
8.7.2.4.2 Setting Up Network Interfaces for Separated Mode 2264
8.7.2.5 Host-based Deployment 2265
8.7.3 Service Deployment 2265
8.7.3.1 DPU Deployment 2265
8.7.3.2 Host Deployment 2265
8.7.4 Configuration 2265
8.7.4.1 Built-In Config File 2265
8.7.4.2 Custom Config File 2266
8.7.4.3 Overriding Specific Config File Parameters 2266
8.7.4.4 Ensuring and Debugging Correctness of Config Files 2267
8.7.5 Description 2268
8.7.5.1 Providers 2268
8.7.5.2 Profiles 2269
8.7.5.3 Outputs 2270
8.7.5.3.1 Container Output 2270
8.7.5.3.2 Firefly Output 2271
8.7.5.3.3 ptp4l Output 2271
8.7.5.3.4 phc2sys Output 2272
8.7.5.3.5 SyncE Output 2272
8.7.5.3.6 Firefly Servo Output 2272
8.7.5.4 Tx Timestamping Support on DPU Mode 2273
8.7.5.4.1 Troubleshooting Tx Timestamp Issues 2273
8.7.5.5 PTP 2274
8.7.5.6 PHC2SYS 2274
8.7.5.7 SYNCE 2275
8.7.5.8 PTP Monitor 2276
8.7.5.8.1 Configuration 2277
8.7.5.8.2 Time Representations (PTP Time vs System Time) 2277
8.7.5.8.3 Monitor Server 2278
8.7.5.8.4 Monitor Client 2280
8.7.5.8.5 Telemetry Export 2280
8.7.5.9 Firefly Servo 2282

8.7.5.9.1 Firefly Servo Configuration 2283
8.7.5.9.2 Dynamic Packet Rate Support 2284
8.7.5.10 VLAN Tagging 2284
8.7.5.10.1 Separated Mode 2264
8.7.5.10.2 Embedded Mode 2284
8.7.5.11 Multiple Interfaces 2285
8.7.6 Troubleshooting 2285
8.7.6.1 Pod is Marked as "Ready" and No Container is Listed 2285
8.7.6.1.1 Error 2285
8.7.6.1.2 Solution 2286
8.7.6.2 Custom Config File is Not Found 2286
8.7.6.2.1 Error 2286
8.7.6.2.2 Solution 2286
8.7.6.3 Profile is Not Supported 2286
8.7.6.3.1 Error 2286
8.7.6.3.2 Solution 2287
8.7.6.4 PPS Capability is Missing 2287
8.7.6.4.1 Error 2287
8.7.6.4.2 Solution 2287
8.7.6.5 Timed Out While Polling for Tx Timestamp 2287
8.7.6.5.1 Error 2287
8.7.6.5.2 Solution 2288
8.7.6.6 Warning – Time Jumped Backwards 2288
8.7.6.6.1 Error 2288
8.7.6.6.2 Solution 2288
8.7.7 PTP Profile Default Config Files 2288
8.7.7.1 Media Profile 2288
8.7.7.2 Default Profile 2288
8.7.7.3 Telco (L2) Profile 2289
8.7.8 Firefly Modules Configuration Options 2289
8.7.8.1 PTP Monitor 2289
8.7.8.1.1 monitor-default.conf 2289
8.7.8.1.2 Configuration Options 2289
8.7.8.2 Firefly Servo 2290
8.7.8.2.1 servo-default.conf 2290
8.7.8.2.2 Configuration Options 2290
8.8 DOCA Flow Inspector Service Guide 2291
8.8.1 Introduction 2291
8.8.1.1 Service Flow 2292
8.8.2 Requirements 2294

8.8.3 Service Deployment 2295
8.8.4 Configuration 2295
8.8.4.1 JSON Input 2295
8.8.4.1.1 Export Unit Attributes 2295
8.8.4.2 Yaml File 2297
8.8.4.3 Verifying Output 2297
8.8.5 Troubleshooting 2298
8.8.5.1 Pod is Marked as "Ready" and No Container is Listed 2298
8.8.5.1.1 Error 2298
8.8.5.1.2 Solution 2299
8.8.5.2 Pod is Not Listed 2299
8.8.5.2.1 Error 2299
8.8.5.2.2 Solution 2299
8.9 DOCA HBN Service Guide 2299
8.9.1 Introduction 2300
8.9.2 Service Function Chaining 2301
8.9.3 HBN Service Release Notes 2302
8.9.3.1 Changes and New Features 2302
8.9.3.2 Supported Platforms and Interoperability 2302
8.9.3.2.1 Supported BlueField Networking Platforms 2302
8.9.3.2.2 Supported BlueField OS 2303
8.9.3.2.3 Verified Scalability Limits 2303
8.9.3.2.4 Performance Limitations 2305
8.9.3.3 Known Issues 2305
8.9.3.4 Bug Fixes 2312
8.9.4 HBN Service Deployment 2317
8.9.4.1 1. HBN Service Requirements 2317
8.9.4.1.1 1.1. Enabling BlueField DPU Mode 2317
8.9.4.1.2 1.2. Enabling SFC 2318
8.9.4.2 2. Scaling Profiles 2323
8.9.4.2.1 2.1. Scaling Profiles and Resource Limits 2323
8.9.4.2.2 2.2. Profile Configuration 2324
8.9.4.2.3 2.3. Changing Profiles 2324
8.9.4.2.4 2.4. System Behavior and Services 2325
8.9.4.3 3. Launching HBN Service 2325
8.9.4.3.1 3.1. HBN Service Container Deployment 2325
8.9.4.3.2 3.2. HBN Deployment Configuration 2328
8.9.4.3.3 3.3. HBN Deployment Considerations 2332
8.9.5 HBN Service Configuration 2338
8.9.5.1 General Network Configuration 2338

8.9.5.1.1 Flat Files Configuration ... 2338
8.9.5.2 NVUE Configuration ... 2339
8.9.5.2.1 NVUE Service ... 2339
8.9.5.2.2 NVUE REST API ... 2339
8.9.5.2.3 NVUE REST API in DOCA Platform Framework ... 2340
8.9.5.2.4 NVUE CLI ... 2342
8.9.5.2.5 NVUE Startup Configuration File ... 2342
8.9.5.3 HBN Configuration Examples ... 2342
8.9.5.3.1 HBN Default Configuration ... 2342
8.9.5.3.2 Layer-3 Routing ... 2343
8.9.5.3.3 Ethernet Virtual Private Network – EVPN ... 2350
8.9.5.3.4 Access Control Lists ... 2381
8.9.5.3.5 Network Address Translation ... 2386
8.9.5.3.6 Bidirectional Forwarding Detection (BFD) for BGP ... 2391
8.9.5.3.7 Control plane policing (CoPP) ... 2402
8.9.5.3.8 DHCP Relay on HBN ... 2403
8.9.6 HBN Service Troubleshooting ... 2406
8.9.6.1 HBN Container Stuck in init-sfs ... 2406
8.9.6.2 Host-side PF/VF Down After BlueField Reboot ... 2407
8.9.6.3 BGP Session not Establishing ... 2407
8.9.6.4 Generating Support Information ... 2407
8.9.6.5 SFC Troubleshooting ... 2408
8.9.6.6 General nl2doca Troubleshooting ... 2408
8.9.6.7 nl2doca Offload Troubleshooting ... 2409
8.9.6.8 NVUE Troubleshooting ... 2411
8.10 DOCA Management Service Guide ... 2411
8.10.1 1. Introduction ... 2411
8.10.1.1 1.1. Goals and Use Cases ... 2411
8.10.1.2 1.2. Key Concepts ... 2412
8.10.1.2.1 1.2.1. Yang - the configuration map ... 2412
8.10.1.2.2 1.2.2. XPath - your address in the map ... 2412
8.10.1.2.3 1.2.3. Target - which device you are talking to ... 2413
8.10.2 2. System Components ... 2413
8.10.2.1 2.1. dms-cli ... 2413
8.10.2.2 2.2. Yang Models ... 2413
8.10.2.3 2.3. OS Tools ... 2414
8.10.3 3. YANG Models ... 2414
8.10.3.1 3.1. Networking & Interfaces ... 2415
8.10.3.2 3.2. Device & Platform ... 2415
8.10.3.3 3.3. Lifecycle & Operations ... 2416

8.10.4 4. DMS-CLI Flags ... 2416
8.10.4.1 4.1. Common flags ... 2417
8.10.4.2 4.2. Introspection flags ... 2417
8.10.5 5. Operations ... 2417
8.10.5.1 5.1. Get ... 2418
8.10.5.2 5.2. Set ... 2418
8.10.5.3 5.3. Actions ... 2419
8.10.5.4 5.4. Batch operations & target scoping ... 2419
8.10.6 6. Introspection ... 2420
8.10.6.1 6.1. tree ... 2420
8.10.6.2 6.2. describe ... 2420
8.10.6.3 6.3. list-xpaths ... 2421
8.10.6.4 6.4. list-targets ... 2421
8.10.7 7. Provisioning and Reset ... 2421
8.10.7.1 7.1. Lifecycle Workflow ... 2422
8.10.7.1.1 7.1.1. Supported Hardware Flows ... 2423
8.10.7.2 7.2. Artifact Management ... 2423
8.10.7.2.1 7.2.1. Stage an Artifact ... 2424
8.10.7.2.2 7.2.2. List Artifacts ... 2425
8.10.7.2.3 7.2.3. Get Artifact Information ... 2426
8.10.7.2.4 7.2.4. Delete an Artifact ... 2426
8.10.7.3 7.3. Provisioning Install ... 2427
8.10.7.3.1 7.3.1. Install Inputs and Source Rules ... 2427
8.10.7.3.2 7.3.2. Asynchronous Start Behavior ... 2428
8.10.7.3.3 7.3.3. Install ConnectX Firmware ... 2428
8.10.7.3.4 7.3.4. Install a BFB on BlueField-3 ... 2428
8.10.7.3.5 7.3.5. Install on BlueField-4 ... 2429
8.10.7.4 7.4. Version Queries ... 2431
8.10.7.4.1 7.4.1. Get the Running ConnectX Firmware Version ... 2431
8.10.7.4.2 7.4.2. Get BlueField-3 Image Versions ... 2432
8.10.7.4.3 7.4.3. Get BlueField-4 OOB Inventory ... 2432
8.10.7.4.4 7.4.4. Compare a BlueField-3 BFB with the Running Image ... 2432
8.10.7.5 7.5. Reset ... 2433
8.10.7.5.1 7.5.1. Check Whether Reset Is Required ... 2433
8.10.7.5.2 7.5.2. Read the Reset Message ... 2433
8.10.7.5.3 7.5.3. Get the Suggested Reset ... 2434
8.10.7.5.4 7.5.4. Get Reset Capabilities ... 2434
8.10.7.5.5 7.5.5. Execute a Reset ... 2434
8.10.7.6 7.6. Monitor Operations ... 2435
8.10.7.7 7.7. Troubleshooting ... 2436
8.10.7.8 7.8. Path Reference ... 2437

8.10.8 8. Prerequisites and Dependencies ... 2438
8.11 DOCA Pipeline Language Services Guide ... 2438
8.11.1 Introduction ... 2439
8.11.1.1 Key Features of P4 ... 2439
8.11.1.2 P4 Compiler ... 2439
8.11.1.3 Focus on NVIDIA's DOCA Pipeline Language ... 2439
8.11.1.4 DPL Highlights ... 2439
8.11.1.4.1 Key Features of DPL ... 2440
8.11.1.4.2 Developer Resources ... 2440
8.11.2 Prerequisites ... 2440
8.11.2.1 Development Environment Requirements ... 2440
8.11.2.2 Suggested Workflow ... 2440
8.11.3 DPL Programming Model ... 2442
8.11.3.1 Parsing ... 2442
8.11.3.1.1 Flex Parsing ... 2442
8.11.3.1.2 Operational Mode ... 2442
8.11.3.2 Match-Action Processing (Steering) ... 2443
8.11.3.3 Forwarding Database ... 2443
8.11.3.4 BlueField DPU and ConnectX SuperNIC Pipeline Behavior ... 2444
8.11.3.5 DPL Services ... 2444
8.11.4 DPL Release Notes ... 2444
8.11.4.1 1. Changes and New Features ... 2444
8.11.4.2 2. Related Documentation ... 2444
8.11.4.3 3. Capabilities and Limitations ... 2445
8.11.4.3.1 3.1. General Constraints ... 2445
8.11.4.3.2 3.2. Supported RPC Messages ... 2445
8.11.4.3.3 3.3. Entity & Operation Support Matrix (Write/Read RPC) ... 2445
8.11.4.3.4 3.4. Metering Limits ... 2445
8.11.4.3.5 3.5. Unsupported Features ... 2446
8.11.4.4 4. Bug Fixes ... 2446
8.11.4.5 5. DPL Runtime Daemon Known Issues ... 2446
8.11.4.6 6. DPL Compiler Known Issues ... 2447
8.11.5 DPL System ... 2447
8.11.5.1 Introduction ... 2448
8.11.5.2 BlueField Network Configuration ... 2448
8.11.5.2.1 DOCA Device Types ... 2448
8.11.5.2.2 Representor Naming ... 2448
8.11.5.3 DPL Architecture ... 2449
8.11.5.4 DPL Components ... 2450
8.11.5.4.1 DPL Compiler ... 2450

8.11.5.4.2 P4Runtime Controller ... 2451
8.11.5.4.3 DPL Nspect ... 2451
8.11.5.4.4 DPL Runtime Service ... 2451
8.11.5.4.5 NV HWS Driver ... 2451
8.11.5.4.6 eSwitch Management ... 2452
8.11.6 DPL Runtime Service ... 2452
8.11.6.1 Introduction ... 2452
8.11.6.2 Requirements ... 2453
8.11.6.2.1 Supported Platforms ... 2453
8.11.6.2.2 Hardware Prerequisites ... 2453
8.11.6.2.3 Software Prerequisites ... 2453
8.11.6.2.4 Firmware Settings ... 2454
8.11.6.3 Supported Port Forwarding ... 2454
8.11.6.4 Deploying the DPL Runtime Service ... 2455
8.11.6.4.1 Prerequisites ... 2455
8.11.6.4.2 Installation ... 2455
8.11.6.5 DPL Dev Container ... 2455
8.11.6.6 Proceeding with DPL ... 2456
8.11.6.7 Implementing Your Own P4Runtime Controller ... 2456
8.11.6.8 DPL Container Deployment ... 2456
8.11.6.8.1 1. Introduction ... 2456
8.11.6.8.2 2. BlueField Preparations ... 2456
8.11.6.8.3 3. Device Configuration ... 2457
8.11.6.8.4 4. Installing the DPL Runtime Service ... 2459
8.11.6.8.5 5. Troubleshooting ... 2464
8.11.6.9 DPL Service Configuration ... 2465
8.11.6.9.1 1. Introduction ... 2465
8.11.6.9.2 2. DPL Device Configuration ... 2465
8.11.6.9.3 3. Service Daemon Configuration ... 2467
8.11.6.9.4 4. System Configuration ... 2470
8.11.7 DPL Development Container ... 2471
8.11.7.1 DOCA Target Architecture ... 2472
8.11.7.1.1 1. Introduction ... 2472
8.11.7.1.2 2. ConnectX-9 Target Architecture ... 2472
8.11.7.1.3 3. Annotations ... 2473
8.11.7.1.4 4. DOCA Pipeline Model ... 2473
8.11.7.1.5 5. DOCA Core Library ... 2475
8.11.7.1.6 6. DOCA Packet Parser ... 2477
8.11.7.1.7 7. DOCA Extern Functions ... 2486
8.11.7.1.8 8. DOCA Extern Objects ... 2512
8.11.7.1.9 9. DPL Expressions ... 2541

8.11.7.1.10 10. Tables ... 2549
8.11.7.1.11 11. Packet IO ... 2573
8.11.7.1.12 12. P4Runtime Extern Entities ... 2573
8.11.7.2 P4 Language Support in DPL ... 2574
8.11.7.2.1 Introduction ... 2574
8.11.7.2.2 P4 Language Features ... 2574
8.11.7.2.3 P4Runtime Features ... 2579
8.11.7.3 DPL Installation Guide ... 2580
8.11.7.3.1 Introduction ... 2580
8.11.7.3.2 Installation ... 2580
8.11.7.3.3 Uninstallation ... 2581
8.11.7.3.4 Working with dpl ... 2581
8.11.7.3.5 Next Steps ... 2582
8.11.7.4 Compiling DPL Applications ... 2582
8.11.7.4.1 Introduction ... 2582
8.11.7.4.2 Installation ... 2582
8.11.7.4.3 Dependencies ... 2582
8.11.7.4.4 Compiling Applications ... 2583
8.11.7.4.5 Compiling Applications in Debug Mode ... 2584
8.11.7.5 Loading DPL Applications ... 2584
8.11.7.5.1 Introduction ... 2584
8.11.7.5.2 Prerequisites ... 2585
8.11.7.5.3 Loading the Application ... 2585
8.11.7.5.4 P4Runtime Optimizations ... 2586
8.11.7.5.5 Troubleshooting ... 2586
8.11.7.6 Sample DPL Applications ... 2587
8.11.7.6.1 Sample Prerequisites ... 2587
8.11.7.6.2 Running the Sample ... 2587
8.11.7.6.3 Sample Applications ... 2588
8.11.7.6.4 Hello Packet Example ... 2588
8.11.7.6.5 GTP Parsing Example ... 2590
8.11.7.6.6 GTP Tunnel Encapsulation Example ... 2593
8.11.7.6.7 GTP Full Tunnel Application ... 2597
8.11.7.6.8 Geneve TLV Parsing Example ... 2601
8.11.7.6.9 VXLAN Tunnel Gateway Example ... 2606
8.11.7.6.10 Connection Tracking Example ... 2610
8.11.7.6.11 TCP State Tracking ... 2616
8.11.7.6.12 Host-based Networking Example ... 2620
8.11.7.6.13 Equal-Cost Multi-Path Routing Example ... 2630
8.11.7.6.14 Entropy in Hash-Based Load Balancing Example ... 2632
8.11.7.6.15 NAT64 Example ... 2634

8.11.7.6.16 Add on Miss ... 2639
8.11.7.6.17 PSP Encryption ... 2640
8.11.7.7 Inspecting DPL Programs ... 2643
8.11.7.7.1 1. Introduction ... 2644
8.11.7.7.2 2. Release Notes ... 2644
8.11.7.7.3 3. DPL Nspect Execution ... 2645
8.11.7.7.4 4. DPL Nspect Commands and Flags ... 2646
8.11.7.8 Debugging DPL Programs ... 2656
8.11.7.8.1 1. Introduction ... 2656
8.11.7.8.2 2. Launching the Debugger ... 2657
8.11.7.8.3 3. Setting Source Directories ... 2659
8.11.7.8.4 4. Debug Workflow ... 2659
8.11.7.8.5 5. UI Reference: Widgets ... 2664
8.11.7.8.6 6. Release Notes ... 2674
8.11.7.9 DPL Program Metrics ... 2674
8.11.7.9.1 Introduction ... 2674
8.11.7.9.2 Configuration and Output ... 2675
8.11.7.9.3 Report Details (program_name.metrics.log) ... 2675
8.11.7.9.4 Optimization Guidelines ... 2677
8.11.7.9.5 Disclaimers ... 2677
8.12 DOCA SNAP Services ... 2677
8.12.1 DOCA SNAP Virtio-fs Service Guide ... 2678
8.12.1.1 1. Introduction ... 2678
8.12.1.1.1 1.1. DOCA SNAP Virtio-fs as Container ... 2679
8.12.1.2 2. DOCA SNAP Virtio-fs Deployment ... 2680
8.12.1.2.1 2.1. Installing Full DOCA Image on BlueField ... 2680
8.12.1.2.2 2.2. Firmware Installation ... 2681
8.12.1.2.3 2.3. Firmware Configuration ... 2681
8.12.1.2.4 2.4. DPA Core Mask ... 2683
8.12.1.2.5 2.5. Virtio-fs DOCA Host Drive Installation (Optional) ... 2684
8.12.1.2.6 2.6. DOCA SNAP Virtio-fs Container Deployment ... 2686
8.12.1.2.7 2.8. DOCA SNAP Virtio-fs with SNAP Support ... 2690
8.12.1.3 3. RPC Commands ... 2690
8.12.1.3.1 3.1. Using JSON-based RPC Protocol ... 2691
8.12.1.3.2 3.2. PCIe Function Management ... 2692
8.12.1.3.3 3.3. Hot-pluggable PCIe Functions Management ... 2693
8.12.1.3.4 3.5. SPDK Fsdev Management ... 2695
8.12.1.3.5 3.6. Virtio-fs Emulation Management ... 2700
8.12.1.3.6 3.7. Configuration Example ... 2710
8.12.1.4 4. RPC Log History ... 2713
8.12.1.5 5. Saving and Loading Configuration ... 2714

8.12.1.6 6. Disabling/enabling SPDK Subsystem 2715
8.12.1.7 7. SR-IOV 2716
8.12.1.7.1 7.1. General Setup Procedure 2716
8.12.1.7.2 7.2. Dynamic MSI-X and Queue Management 2716
8.12.1.8 8. NFS/TCP XLIO 2718
8.12.1.8.1 8.1. Configuration and Resource Management 2718
8.12.1.9 9. Advanced Configuration 2719
8.12.1.9.1 9.1. NFS/RDMA 2719
8.12.1.10 10. Appendix – NVIDIA GPUDirect Storage (GDS) Support in VirtioFS 2720
8.12.1.10.1 10.1. Prerequisites 2720
8.12.1.10.2 10.2. Enabling GDS Support in Virtio-fs 2720
8.12.1.11 11. Appendix – BlueField Firmware Configuration 2721
8.12.1.11.1 11.1. System Configuration Parameters 2722
8.12.1.11.2 11.2. RDMA/RoCE Configuration 2722
8.12.1.11.3 11.3. Virtio-fs Configuration 2723
8.12.1.12 12. Appendix – Host OS Configuration 2723
8.12.1.12.1 12.1. Intel Server Performance Optimizations 2724
8.12.1.12.2 12.2. AMD Server Performance Optimizations 2724
8.12.1.13 13. Appendix – Custom Fsdev Modules 2724
8.12.1.13.1 13.1. Building a New Production Container 2724
8.12.1.13.2 13.2. Manual Build and Run in a Development Container 2725
8.12.1.14 14. Appendix – Custom Bdev Modules 2727
8.12.1.15 15. Appendix – Linux NFS Target 2727
8.12.1.16 16. References 2728
8.12.1.17 SNAP Virtio-fs Service Release Notes 2729
8.12.1.17.1 Key Features in Version 1.6.0-doca3.3.0 2729
8.12.1.17.2 Limitations 2729
8.12.1.17.3 Known Issues 2729
8.12.2 DOCA SNAP-4 Service Guide 2733
8.12.2.1 Introduction 2734
8.12.2.1.1 SNAP as Container 2735
8.12.2.1.2 SNAP as a Package 2735
8.12.2.2 SNAP-4 Service Release Notes 2738
8.12.2.2.1 Changes and New Features in NVIDIA SNAP 4.12.0 2738
8.12.2.2.2 Bug Fixes 2738
8.12.2.2.3 Known Issues 2739
8.12.2.3 SNAP-4 Service Deployment 2747
8.12.2.3.1 1. Installing Full DOCA Image on DPU 2747
8.12.2.3.2 2. Firmware Installation 2747
8.12.2.3.3 3. Firmware Configuration 2748
8.12.2.3.4 4. DPU Configurations 2751

8.12.2.3.5 5. Downloading YAML Configuration 2753
8.12.2.3.6 7. SNAP Source Package Deployment 2758
8.12.2.3.7 8. SNAP-4 Deployment for BlueField-4 2762
8.12.2.4 SNAP-4 Service Environment Variables 2764
8.12.2.4.1 Supported Environment Variables 2764
8.12.2.4.2 YAML Configuration 2764
8.12.2.4.3 Source Package Configuration 2765
8.12.2.5 SNAP-4 Service RPC Commands 2765
8.12.2.5.1 Using JSON-based RPC Protocol 2765
8.12.2.5.2 Log Management 2766
8.12.2.5.3 PCIe Function Management 2766
8.12.2.5.4 Hot-pluggable PCIe Functions Management 2769
8.12.2.5.5 SPDK Bdev Management 2778
8.12.2.5.6 Virtio-blk Emulation Management 2778
8.12.2.5.7 NVMe Emulation Management 2789
8.12.2.5.8 Environment Variable Management 2802
8.12.2.6 SNAP-4 Service Advanced Features 2802
8.12.2.6.1 RPC Log History 2802
8.12.2.6.2 SR-IOV 2803
8.12.2.6.3 Hot-pluggable PCIe Functions 2804
8.12.2.6.4 Zero Copy (SNAP-direct) 2806
8.12.2.6.5 NVMe/TCP XLIO Zero Copy 2806
8.12.2.6.6 Encryption 2808
8.12.2.6.7 Virtio-blk Live Migration 2811
8.12.2.6.8 SNAP Container Live Upgrade 2811
8.12.2.6.9 SR-IOV Dynamic MSI-X and Queues Management 2819
8.12.2.6.10 Recovery 2822
8.12.2.6.11 Watchdog and Heartbeat Monitoring 2826
8.12.2.6.12 IO/Core Multiplexer 2827
8.12.2.6.13 DPA Execution Unit Management 2827
8.12.2.6.14 SNAP ML Optimizer Online 2830
8.12.2.6.15 SNAP ML Optimizer Offline (DEPRECATED) 2832
8.12.2.6.16 Plugins 2834
8.12.2.6.17 Single Protocol Mode 2836
8.12.2.6.18 UEFI Boot Configuration Walkthrough 2838
8.12.2.7 SNAP-4 Service Appendixes 2840
8.12.2.7.1 Appendix – DPU Firmware Configuration 2840
8.12.2.7.2 Appendix – Configure Persistent Network Interfaces 2843
8.12.2.7.3 Appendix – Building SNAP Container with Custom SPDK 2844
8.12.2.7.4 Appendix – Deploying Container on Setups Without Internet Connectivity 2845
8.12.2.7.5 Appendix – Install Legacy SPDK 2846

8.12.2.7.6 Appendix – PCIe BDF to VUID Translation ... 2847
8.12.2.7.7 Appendix – SNAP Memory Consumption ... 2847
8.12.2.7.8 Appendix – Host OS Configuration ... 2848
8.12.2.7.9 Appendix – SPDK Configuration ... 2849
8.12.2.7.10 Appendix – Custom Bdev Modules ... 2851
8.12.3 DOCA SNAP-3 User Guide ... 2854
8.12.3.1 Introduction ... 2854
8.12.3.1.1 SNAP ... 2854
8.12.3.1.2 Libsnap ... 2855
8.12.3.2 SNAP Installation ... 2855
8.12.3.2.1 SNAP Installation Process ... 2855
8.12.3.2.2 SNAP Service Control (systemd) ... 2857
8.12.3.2.3 Logging ... 2857
8.12.3.3 SNAP Commands ... 2857
8.12.3.3.1 RPC Command Interface ... 2857
8.12.3.3.2 PCIe Function Management Commands ... 2858
8.12.3.3.3 NVMe Emulation Management Commands ... 2859
8.12.3.3.4 Virtio-blk Emulation Management Commands ... 2862
8.12.3.3.5 Debug and Statistics ... 2865
8.12.3.3.6 Initialization Scripts ... 2866
8.12.3.4 Advanced Features ... 2866
8.12.3.4.1 Performance Optimization ... 2866
8.12.3.4.2 NVMe-RDMA Full Offload Mode ... 2867
8.12.3.4.3 SR-IOV ... 2869
8.12.3.4.4 Zero Copy (SNAP-direct) ... 2870
8.12.3.4.5 NVMe/TCP Zero Copy ... 2870
8.12.3.4.6 Robustness and Recovery ... 2871
8.12.3.4.7 Mem-pool ... 2872
8.12.3.4.8 Virtio-blk Transitional Device (0.95) ... 2873
8.12.3.4.9 Virtio-blk Live Migration ... 2874
8.12.3.5 Appendix – Live Upgrade ... 2874
8.12.3.5.1 Passing virtio-blk Controller's Management Between SNAP Processes ... 2874
8.12.3.5.2 Full "Live Upgrade" Procedure ... 2875
8.12.3.6 Appendix – Host OS Configuration ... 2875
8.12.3.7 Appendix – SPDK Configuration ... 2876
8.12.3.8 Appendix – Firmware Configuration ... 2876
8.12.3.8.1 Basic Configuration ... 2876
8.12.3.8.2 System Configuration Parameters ... 2877
8.12.3.8.3 NVMe Configuration ... 2877
8.12.3.8.4 Virtio-blk Configuration ... 2878
8.12.3.9 Appendix – JSON File Format ... 2878

8.12.3.9.1 Configuration File Examples ... 2879
8.12.3.9.2 Configuration Parameters ... 2880
8.12.3.10 Appendix – Frequently Asked Questions ... 2882
8.12.3.10.1 How do I enable SNAP? ... 2882
8.12.3.10.2 How do I configure SNAP to support VirtIO-blk? ... 2882
8.12.3.10.3 How do I configure SNAP to work with both ports (for the same or for multiple targets)? ... 2882
8.12.3.10.4 How do I configure offload mode? Which protocols are supported? ... 2883
8.12.3.10.5 How do I configure Firmware for SNAP? ... 2884
8.12.3.10.6 Can I work with custom SPDK on Arm? ... 2884
8.12.3.10.7 Can I replace my backend storage at runtime? ... 2884
8.12.3.10.8 I'm suffering from low performance after updating to latest mlnx-snap. How can I fix it? ... 2885
8.12.3.11 SNAP-3 Release Notes ... 2885
8.12.3.11.1 Changes and New Features ... 2885
8.12.3.11.2 Known Issues ... 2885
8.13 DOCA Telemetry Service Guide ... 2891
8.13.1 1. Introduction ... 2891
8.13.1.1 1.1. Data Export Options ... 2891
8.13.2 2. Service Deployment ... 2891
8.13.2.1 2.1. Built-in DOCA Service Image (BlueField) ... 2891
8.13.2.2 2.2. DOCA Telemetry Exporter Integration ... 2892
8.13.2.3 2.3. DOCA Service on NGC ... 2892
8.13.2.4 2.4. DPU Deployment ... 2892
8.13.2.5 2.5. Host Deployment ... 2893
8.13.2.6 2.6. Deploying with Grafana Monitoring ... 2893
8.13.2.6.1 2.6.1. Grafana Deployment Prerequisites ... 2893
8.13.2.6.2 2.6.2. Grafana Deployment Configuration ... 2893
8.13.2.6.3 2.6.3. Exploring Telemetry Data ... 2895
8.13.3 3. DTS Configuration ... 2895
8.13.3.1 3.1. Initialization Scripts (BlueField Deployment) ... 2896
8.13.3.2 3.2. Enabling Fluent Bit Forwarding ... 2896
8.13.3.3 3.3. Generating Configuration ... 2896
8.13.3.4 3.4. Resetting Configuration ... 2896
8.13.3.5 3.5. Sampling Interval ... 2896
8.13.3.6 3.6. Enabling Providers ... 2897
8.13.3.6.1 3.6.1. Remote Collection (BlueField Only) ... 2897
8.13.3.7 3.7. Enabling Data Output ... 2897
8.13.3.8 3.8. Enabling IPC with External Applications ... 2897
8.13.3.9 3.9. Level Labels ... 2897
8.13.3.9.1 3.9.1. Sample level_labels.ini ... 2898

8.13.3.9.2 3.9.2. Enabling Level Labels ... 2898
8.13.3.9.3 3.9.3. Monitoring and Reloading ... 2898
8.13.3.10 3.10. Device Labels ... 2899
8.13.3.10.1 3.10.1. PCI Label ... 2899
8.13.3.10.2 3.10.2. NUMA Node Label ... 2899
8.13.3.10.3 3.10.3. Example ... 2899
8.13.4 4. Telemetry Providers ... 2900
8.13.4.1 4.1. Hardware & Network Providers ... 2900
8.13.4.1.1 4.1.1. Sysfs ... 2900
8.13.4.1.2 4.1.2. Ethtool Counters ... 2907
8.13.4.1.3 4.1.3. Traffic Control Info ... 2923
8.13.4.1.4 4.1.4. Network Interfaces ... 2923
8.13.4.1.5 4.1.5. vNIC Provider ... 2924
8.13.4.1.6 4.1.6. PPCC_ETH Provider ... 2924
8.13.4.2 4.2. GPU & Performance Providers ... 2925
8.13.4.2.1 4.2.1. NVIDIA System Management Interface ... 2925
8.13.4.2.2 4.2.2. NVIDIA Data Center GPU Manager ... 2925
8.13.4.2.3 4.2.3. BlueField Performance ... 2926
8.13.4.3 4.3. Infrastructure Providers ... 2926
8.13.4.3.1 4.3.1. Storage Provider ... 2926
8.13.4.3.2 4.3.2. Amber Provider ... 2928
8.13.4.3.3 4.3.3. Diagnostic Data ... 2928
8.13.4.3.4 4.3.4. RDMA Notifications ... 2939
8.13.4.3.5 4.3.5. Adaptive Retx Provider ... 2942
8.13.4.3.6 4.3.6. Phy Provider ... 2943
8.13.4.3.7 4.3.7. PCI Provider ... 2943
8.13.4.4 4.4. Aggregation Providers ... 2945
8.13.4.4.1 4.4.1. Fluent Aggregator ... 2945
8.13.4.4.2 4.4.2. Prometheus Aggregator ... 2945
8.13.5 5. Data Exporters ... 2946
8.13.5.1 5.1. Data Writer ... 2946
8.13.5.2 5.2. Prometheus ... 2947
8.13.5.2.1 5.2.1. Data Retrieval & Formatting ... 2947
8.13.5.2.2 5.2.2. Retrieving Schema Metadata ... 2947
8.13.5.2.3 5.2.3. Configuration Details (dts_config.ini) ... 2947
8.13.5.2.4 5.2.4. Prometheus Aggregator Exporter ... 2948
8.13.5.3 5.3. Fluent Bit ... 2948
8.13.5.3.1 5.3.1. Export File Configuration Details ... 2949
8.13.5.3.2 5.3.2. Msgpack Data Layout ... 2949
8.13.5.3.3 5.3.3. Cset/Fset Filtering ... 2950
8.13.5.4 5.4. NetFlow Exporter ... 2951

8.13.5.5 5.5. Open Telemetry Exporter 2951
8.13.5.5.1 5.5.1. Transport & Security Configurations 2951
8.13.5.5.2 5.5.2. Advanced Data & Filtering Configurations 2952
8.13.5.5.3 5.5.3. Serialization Overrides 2953
8.13.5.5.4 5.5.4. Open Telemetry Metrics Receiver Sample Configuration 2953
8.13.5.5.5 5.5.5. Running the Open Telemetry Instance 2954
8.13.5.6 5.6. Prometheus Remote Write Exporter 2954
8.13.5.7 5.7. Load Balancer Exporter 2954
8.13.5.8 5.8. Distributed Real Time Analysis Exporter 2955
8.13.5.8.1 5.8.1. Additional Configuration Parameters 2956
8.13.5.8.2 5.8.2. Lua Script Requirements 2957
8.13.5.8.3 5.8.3. Lua Scripts Workflow 2957
8.13.5.8.4 5.8.4. Lua Script Examples 2957
8.13.5.8.5 5.8.5. Lua Scripts API 2958
8.13.5.8.6 5.8.6. HTTP REST API 2964
8.13.6 6. Counterset and Fieldset Files 2968
8.13.6.1 6.1. Difference Between .cset and .fset Files 2968
8.13.6.2 6.2. Counterset (.cset) File Format 2969
8.13.6.3 6.3. Fieldset (.fset) File Format 2969
8.13.6.4 6.4. Best Practices 2969
8.13.6.5 6.5. Common Issues 2970
8.13.7 7. DOCA Privileged Executor 2970
8.13.7.1 7.1. DPE Usage 2970
8.13.7.2 7.2. DPE Configuration File 2971
8.13.8 8. Troubleshooting 2971
8.13.9 9. Release Notes 2972
8.13.9.1 9.1. New Features 2972
8.13.9.2 9.2. Bug Fixes 2972
8.13.9.3 9.3. Known issues 2972
8.14 DOCA UROM Service Guide 2973
8.14.1 Introduction 2973
8.14.2 Requirements 2974
8.14.3 Service Deployment 2974
8.14.4 Description 2974
8.14.4.1 Plugin Discovery and Reporting 2974
8.14.4.2 Loading Plugin in Worker 2975
8.14.4.3 Yaml File 2975
8.14.5 Troubleshooting 2975
8.14.5.1 Pod is Marked as "Ready" and No Container is Listed 2976

8.14.5.1.1 Error ... 2976
8.14.5.1.2 Solution ... 2976
8.14.5.2 Pod is Not Listed ... 2976
8.14.5.2.1 Error ... 2976
8.14.5.2.2 Solution ... 2977
8.15 DOCA Virtio-net Service Guide ... 2977
8.15.1 Introduction ... 2977
8.15.2 Virtio-net Controller SystemD Service ... 2978
8.15.3 Virtio-net Deployment ... 2979
8.15.3.1 Updating OS Image on BlueField ... 2979
8.15.3.2 Updating NIC Firmware ... 2979
8.15.3.3 Configuring NIC Firmware ... 2979
8.15.3.3.1 Static PF ... 2981
8.15.3.3.2 Hotplug PF ... 2982
8.15.3.3.3 SR-IOV VF ... 2982
8.15.3.3.4 PF/VF Combinations ... 2983
8.15.3.4 System Configuration ... 2983
8.15.3.4.1 Host System Configuration ... 2983
8.15.3.4.2 BlueField System Configuration ... 2984
8.15.3.5 Usage ... 2985
8.15.3.5.1 Hotplug PF Devices ... 2986
8.15.3.5.2 SR-IOV VF Devices ... 2989
8.15.3.6 Assigning Virtio-net Device to VM ... 2992
8.15.3.6.1 Locating Virtio-net Devices ... 2992
8.15.3.6.2 Using virt-manager ... 2992
8.15.3.6.3 Using virsh Command ... 2993
8.15.4 Configuration File ... 2994
8.15.4.1 Configuration File Options ... 2994
8.15.4.2 Configuration File Examples ... 3001
8.15.4.2.1 Configuring LAG on Dual Port BlueField ... 3001
8.15.4.2.2 Configuring Static PF on Dual Port BlueField ... 3001
8.15.4.2.3 Configuring VF Specific Options ... 3001
8.15.4.2.4 Virtio Live Migration Settings ... 3001
8.15.4.2.5 DPA Configuration (SPRD) ... 3002
8.15.5 Virtnet CLI Commands ... 3004
8.15.5.1 User Front End CLI ... 3004
8.15.5.2 Hotplug ... 3004
8.15.5.2.1 Syntax ... 3004
8.15.5.2.2 Output ... 3007
8.15.5.2.3 Example ... 3007

8.15.5.3 Unplug 3007
8.15.5.3.1 Syntax 3007
8.15.5.3.2 Output 3010
8.15.5.3.3 Example 3007
8.15.5.4 List 3010
8.15.5.4.1 Syntax 3010
8.15.5.4.2 Output 3011
8.15.5.4.3 Example 3015
8.15.5.5 Query 3016
8.15.5.5.1 Syntax 3016
8.15.5.5.2 Output 3017
8.15.5.5.3 Example 3021
8.15.5.6 Stats 3022
8.15.5.6.1 Syntax 3022
8.15.5.6.2 Output 3023
8.15.5.6.3 Example 3025
8.15.5.7 Modify Device 3025
8.15.5.7.1 Syntax 3026
8.15.5.7.2 Output 3033
8.15.5.7.3 Example 3033
8.15.5.8 Log 3033
8.15.5.8.1 Syntax 3033
8.15.5.8.2 Output 3034
8.15.5.8.3 Example 3034
8.15.5.9 Validate 3034
8.15.5.9.1 Syntax 3034
8.15.5.9.2 Output 3034
8.15.5.9.3 Example 3034
8.15.5.10 Version 3035
8.15.5.10.1 Syntax 3035
8.15.5.10.2 Output 3035
8.15.5.10.3 Example 3035
8.15.5.11 Update 3035
8.15.5.11.1 Syntax 3035
8.15.5.11.2 Output 3036
8.15.5.11.3 Example 3036
8.15.5.12 Restart 3036
8.15.5.12.1 Syntax 3036
8.15.5.12.2 Output 3037
8.15.5.12.3 Example 3037
8.15.5.13 Health 3037

8.15.5.13.1 Syntax ... 3037
8.15.5.13.2 Output ... 3038
8.15.5.13.3 Example ... 3039
8.15.5.14 Error Code ... 3040
8.15.5.15 Debug ... 3044
8.15.6 Feature Guidance ... 3045
8.15.6.1 Counters ... 3045
8.15.6.1.1 Packet Statistics ... 3045
8.15.6.1.2 RQ Drop Counter ... 3049
8.15.6.1.3 Packet Counter ... 3054
8.15.6.1.4 Health Statistics ... 3055
8.15.6.2 Dynamic Interruption Moderation ... 3056
8.15.6.2.1 Configuring DIM ... 3057
8.15.6.3 High Availability ... 3057
8.15.6.4 Jumbo MTU ... 3058
8.15.6.5 Link Aggregation ... 3059
8.15.6.6 Live Migration ... 3060
8.15.6.6.1 Live Migration Using vHost Acceleration Software Stack ... 3060
8.15.6.6.2 Live Migration Using VFIO With Full Emulation ... 3065
8.15.6.7 Live Update ... 3070
8.15.6.7.1 Requirements ... 3070
8.15.6.7.2 Check Versions ... 3070
8.15.6.7.3 Start Updating ... 3070
8.15.6.7.4 Check Status ... 3070
8.15.6.8 Mergeable Rx Buffer ... 3071
8.15.6.8.1 Configuration ... 3071
8.15.6.8.2 Limitations ... 3071
8.15.6.9 Performance Tuning ... 3072
8.15.6.9.1 Number of Queues and MSIX ... 3072
8.15.6.9.2 Queue Depth ... 3073
8.15.6.9.3 MTU ... 3073
8.15.6.10 Device State Recovery ... 3073
8.15.6.10.1 Recovery File Structure ... 3073
8.15.6.10.2 Example Recovery File ... 3075
8.15.6.10.3 Use Cases ... 3075
8.15.6.11 Transitional Device ... 3076
8.15.6.11.1 Transitional Virtio-net VF Device ... 3076
8.15.6.12 VF Dynamic MSIX ... 3077
8.15.6.12.1 Firmware Configuration ... 3078
8.15.6.12.2 MSIX ... 3078
8.15.6.12.3 Queue Pairs ... 3080

8.15.6.13 Virt Queue Types ... 3082
8.15.6.13.1 Split VQ ... 3082
8.15.6.13.2 Packed VQ ... 3082
8.15.6.14 Virtio-net Feature Bits ... 3083
8.15.6.15 Virtio-net Event Notifications ... 3084
8.15.6.15.1 Supported Event Types ... 3084
8.15.6.15.2 Prerequisites ... 3084
8.15.6.15.3 NATS Subject Scheme ... 3086
8.15.6.15.4 JSON Event Schema (v1) ... 3087
8.15.6.15.5 Consuming Events ... 3088
8.15.6.15.6 Troubleshooting ... 3091
8.15.7 Virtio-net Service Guide Release Notes ... 3092
8.15.7.1 Changes and New Features in This Release ... 3092
8.15.7.2 Bug Fixes ... 3092
8.15.7.3 Known Issues ... 3094
8.16 OpenvSwitch Acceleration - OVS in DOCA ... 3098
8.16.1 Introduction ... 3098
8.16.2 OVS and Virtualized Devices ... 3099
8.16.3 OVS-Kernel Hardware Acceleration ... 3099
8.16.3.1 Switchdev Configuration ... 3100
8.16.3.2 Switchdev Performance Tuning ... 3101
8.16.3.2.1 Steering Mode ... 3101
8.16.3.2.2 Troubleshooting SMFS ... 3101
8.16.3.2.3 vPort Match Mode ... 3102
8.16.3.2.4 Flow Table Large Group Number ... 3102
8.16.3.3 Open vSwitch Configuration ... 3103
8.16.3.4 OVS Performance Tuning ... 3104
8.16.3.4.1 Flow Aging ... 3104
8.16.3.4.2 TC Policy ... 3104
8.16.3.4.3 max-revalidator ... 3105
8.16.3.4.4 n-handler-threads ... 3105
8.16.3.4.5 n-revalidator-threads ... 3105
8.16.3.5 Basic TC Rules Configuration ... 3105
8.16.3.6 SR-IOV VF LAG ... 3106
8.16.3.6.1 SR-IOV VF LAG Configuration on ASAP2 ... 3106
8.16.3.6.2 Scalable/Sub-Function (SF) LAG Configuration on ASAP2 ... 3107
8.16.3.6.3 Using TC with VF LAG ... 3107
8.16.3.7 Classification Fields (Matches) ... 3108
8.16.3.7.1 Ethernet Layer 2 ... 3108
8.16.3.7.2 IPv4/IPv6 ... 3108

8.16.3.7.3 TCP/UDP Source and Destination Ports and TCP Flags 3109
8.16.3.7.4 VLAN 3109
8.16.3.7.5 Tunnel 3110
8.16.3.8 Supported Actions 3110
8.16.3.8.1 Forward 3110
8.16.3.8.2 Drop 3111
8.16.3.8.3 Statistics 3111
8.16.3.8.4 Tunnels: Encapsulation/Decapsulation 3112
8.16.3.8.5 VLAN Push/Pop 3113
8.16.3.8.6 Header Rewrite 3114
8.16.3.8.7 Ethernet Layer 2 3114
8.16.3.8.8 IPv4/IPv6 3114
8.16.3.8.9 Connection Tracking 3116
8.16.3.8.10 Forward to Chain (TC Only) 3117
8.16.3.9 Port Mirroring: Flow-based VF Traffic Mirroring for ASAP² 3117
8.16.3.10 Forward to Multiple Destinations 3119
8.16.3.11 sFlow 3119
8.16.3.12 Rate Limit 3120
8.16.3.13 Kernel Requirements 3121
8.16.3.14 VF Metering 3121
8.16.3.15 Representor Metering 3122
8.16.3.16 OVS Metering 3122
8.16.3.17 Multiport eSwitch Mode 3123
8.16.4 OVS-DPDK Hardware Acceleration 3123
8.16.5 OVS-DOCA Hardware Acceleration 3124
8.16.5.1 1. Configuring OVS-DOCA 3124
8.16.5.1.1 1.1. Setting Default Datapath 3126
8.16.5.2 2. OVS-DOCA Design Considerations 3126
8.16.5.2.1 2.1. Eswitch Dependency 3126
8.16.5.2.2 2.2. Pre-allocated Offload Tables 3126
8.16.5.2.3 2.3. Unsupported CT-CT-NAT 3127
8.16.5.3 3. OVS-DOCA Specific vSwitch Configuration 3127
8.16.5.3.1 3.1. other_config 3127
8.16.5.4 4. Offloading VXLAN Encapsulation/Decapsulation Actions 3129
8.16.5.4.1 4.1. VXLAN GBP Extension 3129
8.16.5.4.2 4.2. VF-Tunnel Configuration 3130
8.16.5.5 5. Offloading Connection Tracking 3130
8.16.5.6 6. SR-IOV VF LAG 3131
8.16.5.7 7. Multiport eSwitch Mode 3132
8.16.5.8 8. Offloading Geneve Encapsulation/Decapsulation 3133
8.16.5.9 9. GRE Tunnel Offloads 3134

8.16.5.10 10. Slow Path Rate Limiting/SW-Meter 3134
8.16.5.11 11. Hairpin 3135
8.16.5.12 12. OpenFlow Meters 3135
8.16.5.13 13. DP-HASH Offloads 3136
8.16.5.14 14. sFlow 3136
8.16.5.15 15. Mirroring 3137
8.16.5.16 16. Guaranteed Packet Rate 3137
8.16.5.17 17. Configuring OVS-DOCA Pre-Miss Rules (Kernel Punting) 3138
8.16.5.17.1 17.1. Constraints and Default Behavior 3138
8.16.5.17.2 17.2. Configuration Commands 3138
8.16.5.18 18. OVS-DOCA Known Limitations 3139
8.16.5.19 19. OVS-DOCA Debugging 3139
8.16.5.20 20. Scaling Megaflows 3140
8.16.5.21 21. Software Datapath Packet Capture (ovs-doca-tcpdump) 3140
8.16.5.21.1 21.1. Prerequisites 3141
8.16.5.21.2 21.2. Tool Capabilities 3141
8.16.5.21.3 21.3. Example Usage 3141
8.16.5.22 Multi-PMD Configuration 3142
8.16.5.22.1 Configuring PMD CPU Mask 3142
8.16.5.22.2 Configuring Multiple RX Queues 3142
8.16.5.22.3 Scheduling Queues on PMD Threads 3142
8.16.5.22.4 Verifying PMD Thread Scheduling 3143
8.16.5.22.5 Limitations 3143
8.16.6 OVS Metrics 3143
8.16.6.1 Introduction 3143
8.16.6.2 Prometheus Metrics 3144
8.16.6.2.1 Running ovs-exporter 3144
8.16.6.2.2 Integration with DOCA Telemetry Services 3145
8.16.6.3 OVS CLI Metrics 3145
8.16.6.4 OVS-metrics Dashboard 3145
8.16.6.5 OVS-Per Flow Counters 3146
8.16.6.5.1 Management Commands 3146
8.16.6.5.2 Parameters 3146
8.16.6.5.3 Operational Behavior 3147
8.16.7 OVS Inside BlueField 3147
8.16.7.1 Verifying Host Connection on Linux 3147
8.16.7.2 Verifying Connection from Host to BlueField 3147
8.16.7.3 Verifying Host Connection on Windows 3148
8.16.8 Virtual Switch on BlueField 3148
8.16.8.1 Verifying Host Connection on Linux 3149
8.16.8.2 Verifying Connection from Host to BlueField 3150

8.16.8.3 Verifying Host Connection on Windows 3151
8.16.8.4 Enabling OVS HW Offloading 3151
8.16.8.5 Enabling OVS-DPDK Hardware Offload 3152
8.16.8.6 Configuring DPDK and Running TestPMD 3153
8.16.8.7 Flow Statistics and Aging 3153
8.16.8.8 Connection Tracking Offload 3153
8.16.8.8.1 Configuring Connection Tracking Offload 3154
8.16.8.8.2 Connection Tracking With NAT 3154
8.16.8.8.3 Querying Connection Tracking Offload Status 3155
8.16.8.8.4 Performance Tune Based on Traffic Pattern 3155
8.16.8.8.5 Connection Tracking Aging 3156
8.16.8.8.6 Maximum Tracked Connections 3156
8.16.8.9 Offloading VLANs 3156
8.16.8.10 VXLAN Tunneling Offload 3157
8.16.8.10.1 Configuring VXLAN Tunnel 3157
8.16.8.10.2 Querying OVS VXLAN hw_offload Rules 3157
8.16.8.11 GRE Tunneling Offload 3158
8.16.8.11.1 Configuring GRE Tunnel 3158
8.16.8.11.2 Querying OVS GRE hw_offload Rules 3158
8.16.8.12 GENEVE Tunneling Offload 3159
8.16.8.12.1 Configuring GENEVE Tunnel 3159
8.16.8.13 Using TC Interface to Configure Offload Rules 3160
8.16.8.13.1 L2 Rules Example 3160
8.16.8.13.2 VLAN Rules Example 3160
8.16.8.13.3 VXLAN Encap/Decap Example 3160
8.16.8.14 Virtio Acceleration Through Hardware vDPA 3161
8.16.9 OVS Live Upgrade 3161
8.16.9.1 Introduction 3161
8.16.9.2 NDU Operational Model 3161
8.16.9.3 Prerequisites 3162
8.16.9.4 Upgrade Procedure 3162
8.16.9.4.1 Initiating the Upgrade 3162
8.16.9.4.2 Post-Upgrade Validation 3162
8.16.9.4.3 Error Handling and Rollback 3162
8.17 DOCA XPlane Service Guide 3163
8.17.1 1. Introduction 3163
8.17.2 2. Requirements 3163
8.17.2.1 2.1. Configuring Firmware Settings on the SuperNIC 3163
8.17.2.2 2.2. Setting devlink Parameters 3163
8.17.2.3 2.3. Creating SR-IOV Virtual Functions 3164
8.17.2.4 2.4. Configuring hugepages 3164

8.17.3 3. Service Deployment ... 3164
8.17.3.1 3.1. XPlane service ... 3164
8.17.3.2 3.2. CLI client ... 3165
8.17.4 4. Configuration File ... 3165
8.17.5 5. Maintenance mode ... 3166
8.17.6 6. Troubleshooting ... 3166
8.17.6.1 6.1. Checking System Status ... 3166
8.17.6.2 6.2. Service Logs ... 3167
8.17.6.3 6.3. Using doca-xplane-client ... 3167
8.17.6.3.1 6.3.1. Retrieve Overall System Status ... 3167
8.17.6.3.2 6.3.2. Print Topology as Read by DOCA XPlane Agent ... 3167
8.17.6.3.3 6.3.3. Get Summary of Configured Planes ... 3168
8.17.6.3.4 6.3.4. Get Local Failures (e.g., link down) for Specific Plane ... 3168
8.17.6.3.5 6.3.5. Get Remote Failures for Specific Plane ... 3168
8.17.6.3.6 6.3.6. Get Traffic Summary for Specific Plane ... 3169
8.17.6.3.7 6.3.7. Get Statistics for Traffic Diverted from One Plane to Another ... 3169
9 API References ... 3171
10 DOCA Troubleshooting ... 3172
10.1 NVIDIA BlueField Reset and Reboot Procedures ... 3172
10.1.1 BlueField System Reboot ... 3172
10.1.2 BlueField System-level Reset ... 3172
10.1.2.1 System-level Reset for BlueField in DPU Mode ... 3172
10.1.2.2 System-level Reset for BlueField in NIC Mode ... 3174
10.1.2.3 System-level Reset for Host with Separate Power Control ... 3174
10.1.3 BlueField DPU Reset Using a BMC Platform ... 3176
10.1.4 Shutdown BlueField DPU OS (Command=0x12, Parameter=0x1A) ... 3177
10.1.4.1 Shutdown BlueField DPU OS Response ... 3177
11 Miscellaneous (Runtime) ... 3179
11.1 DOCA Glossary ... 3179
11.2 DOCA Crypto Acceleration ... 3184
11.3 DOCA Services Fluent Logger ... 3185
11.3.1 Introduction ... 3185
11.3.2 Deployment ... 3185
11.3.3 Configuration ... 3186
11.3.4 Troubleshooting ... 3187
11.4 DOCA DPU CLI ... 3187

11.4.1 Introduction ... 3187
11.4.2 General Commands ... 3187
11.4.3 DPU/DOCA Commands ... 3188
11.5 DOCA Switching ... 3190
11.5.1 Introduction ... 3190
11.5.2 Switch Device Mode ... 3190
11.5.2.1 ConnectX and BlueField NIC Mode ... 3191
11.5.2.2 BlueField DPU Mode ... 3191
11.5.3 E-Switch Port ... 3192
11.5.3.1 Netdev Interface ... 3192
11.5.3.2 RDMA Core Interface ... 3193
11.5.4 Representor Port ... 3193
11.5.4.1 Netdev Interface ... 3193
11.5.4.2 RDMA Core Interface ... 3193
11.5.5 Configuring Switchdev Mode ... 3193
11.5.6 DPU Kernel Representors Model ... 3194
11.5.7 Virtio Acceleration through Hardware vDPA ... 3196
11.5.7.1 Hardware vDPA Installation ... 3196
11.5.7.2 Hardware vDPA Configuration ... 3197
11.5.7.3 Running Hardware vDPA ... 3198
11.5.8 Bridge Offload ... 3198
11.5.8.1 Basic Configuration ... 3199
11.5.8.2 Configuring VLAN ... 3199
11.5.8.3 VF LAG Support ... 3199
11.5.9 Link Aggregation ... 3200
11.5.9.1 LAG Modes ... 3200
11.5.9.1.1 Queue Affinity Mode ... 3201
11.5.9.1.2 Hash Mode ... 3201
11.5.9.2 Prerequisites ... 3201
11.5.9.3 LAG Configuration ... 3202
11.5.9.4 Removing LAG Configuration ... 3203
11.5.9.5 LAG on Multi-host ... 3204
11.5.9.5.1 LAG Multi-host Prerequisites ... 3204
11.5.9.5.2 LAG Configuration on Multi-host ... 3204
11.5.9.5.3 Removing LAG Configuration on Multi-host ... 3205
11.5.10 Controlling Host PF and VF Parameters ... 3205
11.5.10.1 Setting Host PF and VF Default MAC Address ... 3205
11.5.10.2 Setting Host PF and VF Link State ... 3205
11.5.10.3 Querying Configuration ... 3205

11.5.10.4 Disabling Host Networking PFs 3206
11.5.10.5 Configuring Uplink MTU 3206
11.5.11 DPDK on BlueField 3206
11.6 DOCA with OpenSSL 3207
11.6.1 Introduction 3207
11.6.2 Prerequisites 3207
11.6.3 Architecture 3207
11.6.4 Capabilities and Limitations 3208
11.6.5 OpenSSL Command Line Verification 3208
11.6.6 OpenSSL Throughput Test 3209
11.6.7 Using DOCA SHA Offload Engine in OpenSSL Application 3209
11.7 BlueField Scalable Functions 3210
11.7.1 Overview 3210
11.7.2 Before You Begin 3210
11.7.2.1 Requirements 3210
11.7.2.2 Choose the Target Controller 3211
11.7.2.3 Understand the SF Identifiers 3211
11.7.3 Configure SF Resources 3212
11.7.4 Create and Deploy an SF 3212
11.7.4.1 Create the SF Port 3213
11.7.4.2 Configure and Activate the SF 3214
11.7.4.3 Identify the Auxiliary Device 3214
11.7.4.4 Bind the SF Driver 3215
11.7.4.5 Verify the SF 3215
11.7.5 Use an SF 3215
11.7.6 Example: Run an Application Using Two SFs and OVS 3216
11.7.6.1 Prepare SFs 4 and 5 3216
11.7.6.2 Remove an Existing Bridge, if Necessary 3216
11.7.6.3 Create the OVS Bridges 3216
11.7.6.4 Add the SF Representors 3217
11.7.6.5 Verify the OVS Configuration 3217
11.7.6.6 Run the Application 3217
11.7.7 Manage SFs 3217
11.7.7.1 List and Inspect SFs 3218
11.7.7.2 Remove an SF 3218
11.7.8 Use devlink on an Upstream Kernel 3218
11.7.9 Related Documentation 3218

11.8 DOCA TLS Offload ... 3219
11.8.1 1. Introduction ... 3219
11.8.1.1 1.1. TLS Handshake ... 3219
11.8.1.2 1.2. kTLS ... 3220
11.8.1.3 1.3. HW-offloading kTLS ... 3220
11.8.1.3.1 1.3.1. kTLS Offload Flow in High Level ... 3220
11.8.1.3.2 1.3.2. Resync and Error Handling ... 3221
11.8.2 2. Prerequisites ... 3221
11.8.2.1 2.1. Checking Hardware Support for Crypto Acceleration ... 3221
11.8.2.2 2.2. Kernel Requirements ... 3222
11.8.3 3. Configurations and Useful Commands ... 3223
11.8.3.1 3.1. TLS Setup ... 3223
11.8.3.2 3.2. Finding NVIDIA Interfaces ... 3223
11.8.3.3 3.3. Configuring TLS Offload ... 3223
11.8.3.4 3.4. Configuring OVS Bridge on BlueField ... 3224
11.8.4 4. Common Use Cases ... 3224
11.8.4.1 4.1. OpenSSL ... 3224
11.8.4.2 4.2. Nginx ... 3225
11.8.4.2.1 4.2.1. Prerequisites ... 3226
11.8.4.2.2 4.2.2. Configuration ... 3226
11.8.4.2.3 4.2.3. Stopping Nginx ... 3227
11.8.4.2.4 4.2.4. Wrk – Client ... 3227
11.8.4.2.5 4.2.5. Using Wrk ... 3227
11.8.5 5. Testing Offload via OpenSSL ... 3227
11.8.5.1 5.1. TLS Testing Setup ... 3227
11.8.5.2 5.2. Adding Certificate and Key ... 3228
11.8.5.3 5.3. Running Server Side ... 3228
11.8.5.4 5.4. Running Client Side ... 3229
11.8.5.5 5.5. Testing kTLS ... 3229
11.8.6 6. Optimizations over kTLS ... 3230
11.8.6.1 6.1. XLIO ... 3230
11.8.7 7. Performance Tuning Options ... 3230
11.8.8 8. Additional Reading ... 3232
11.9 DOCA Virtual Functions ... 3232
11.9.1 Introduction ... 3232
11.9.2 Prerequisites ... 3232
11.9.3 VF Creation ... 3233
11.9.4 Running DOCA Application on Host ... 3234

11.9.5 Topology Example ... 3235
11.9.6 VF Creation on Adapter Card ... 3236
11.10 BlueField SR-IOV ... 3236
11.10.1 Introduction ... 3237
11.10.2 Enabling BlueField SR-IOV ... 3237
11.10.3 Configuring VF Limits ... 3237
11.10.4 Applying Configuration ... 3237
11.10.5 Instantiating VFs ... 3238
11.10.6 Limitations and Legacy Support ... 3238
11.11 fTPM over OP-TEE ... 3238
11.11.1 Introduction ... 3238
11.11.2 Architecture and Secure Storage ... 3239
11.11.3 Enabling OP-TEE on BlueField-3 ... 3241
11.11.4 Verifying BlueField-3 is Running OP-TEE ... 3241
12 Archives ... 3243

1 DOCA Framework

This page provides an overview of the NVIDIA DOCA Framework including details on its architecture, installation packages, and software components.

1.1 Introduction

The DOCA Framework provides a comprehensive software suite that includes drivers, libraries, tools, and low-level software for interacting with hardware devices and the default operating system installed with BlueField.

Rather than being a single installation package, the DOCA Framework is a collection of DOCA-related software components, distributed through two primary packages: DOCA-Host and BlueField-Bundle.

The framework consists of two main parts:

DOCA SDK	Enables developers to quickly build applications and services on NVIDIA® BlueField® networking platforms, leveraging industry-standard APIs and the power of NVIDIA BlueField DPUs and SuperNICs. DOCA SDK documentation is available here.
BlueField Platform Software	Provides the essential components for booting and configuring BlueField devices, including bootloaders, firmware, and the official Ubuntu-based reference Linux distribution. It contains the ATF/UEFI, BMC firmware, and BlueField NIC firmware. BlueField platform software documentation is available here.

1.2 DOCA Packages

To ensure an optimal installation of DOCA on your host and device, DOCA provides the installation packages presented in the following subsections.

1.2.1 DOCA-Host

DOCA-Host is the software package installed on the host server to support NVIDIA devices, including BlueField and ConnectX.

DOCA-Host offers multiple installation profiles, each tailored to specific use cases:

- doca-all
- doca-networking
- doca-ofed

See details on installation profiles and supported host operating systems here.

1.2.2 BlueField-Bundle (BF-Bundle)

The BlueField software bundle (BF-Bundle) is installed on the BlueField Arm cores to provide a complete DOCA experience on the BlueField networking platform. BlueField devices are shipped with BF-Bundle preinstalled.

BF-Bundle includes:

- DOCA SDK libraries, drivers, and tools
- BlueField Platform Software
- The default operating system for BlueField Arm cores (Ubuntu 22.04)

1.2.3 BlueField-Firmware Bundle (BF-FWBundle)

The BlueField Firmware Bundle (BF-FWBundle) is a minimal software package installed on the BlueField Arm cores. It provides only the management interface required for external provisioning and software loading.

BF-FWBundle includes:

- ATF
- UEFI
- BlueField NIC Firmware
- BMC Firmware
- eROT Firmware

BF-FWBundle does not include:

- DOCA SDK
- An operating system for BlueField Arm cores

1.3 BlueField Devices

BlueField user manuals provide detailed instructions on installation, supported interfaces, specifications, and troubleshooting.

2 DOCA SDK

This section contains the following pages:

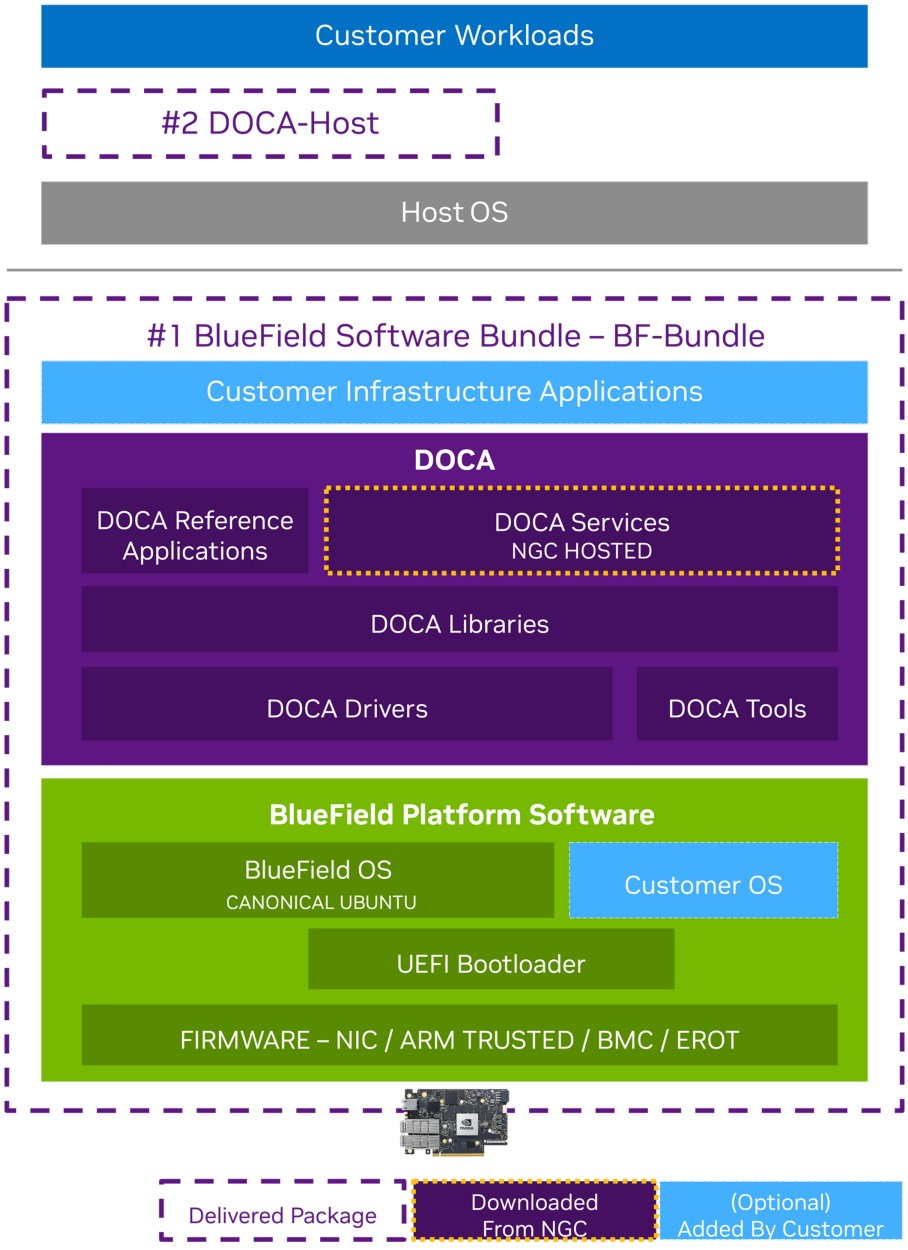
- DOCA Overview
- DOCA Release Notes
- BlueField and DOCA User Types
- DOCA EULA

2.1 DOCA Overview

This is an overview of the structure of NVIDIA DOCA documentation. It walks you through DOCA's developer zone portal which contains all the information about the DOCA toolkit from NVIDIA, providing all you need to develop NVIDIA® BlueField®-accelerated applications and the drivers for the host.

2.1.1 Introduction

The NVIDIA DOCA™ Framework enables rapidly creating and managing applications and services on top of the BlueField networking platform, leveraging industry-standard APIs. With DOCA, developers can deliver breakthrough networking, security, and storage performance by harnessing the power of NVIDIA's BlueField data-processing units (DPUs) and SuperNICs. Installing DOCA on your host provides all the necessary drivers and tools to manage NVIDIA® BlueField® and NVIDIA® ConnectX® devices.



DOCA Framework includes the DOCA-Host package and the BlueField Software Bundle for BlueField Arm:

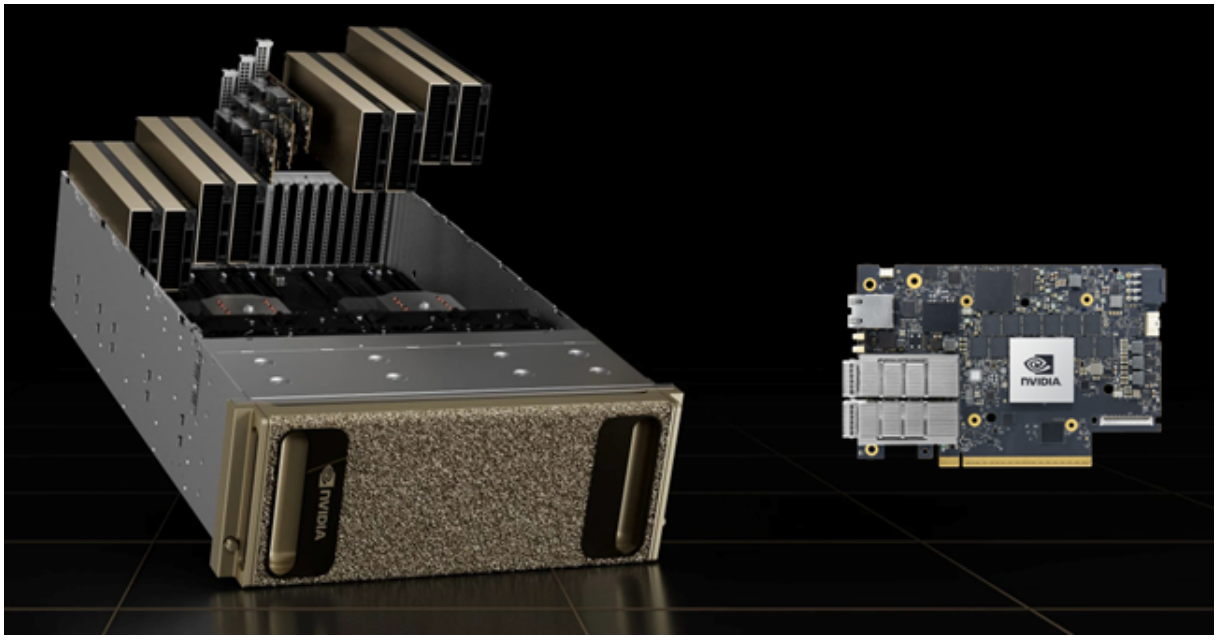
- BlueField Software Bundle (BF-Bundle) is the software package installed on the BlueField Arm cores
- DOCA-Host is the software package installed on the host server which includes different DOCA installation profiles

The BlueField Software Bundle includes:

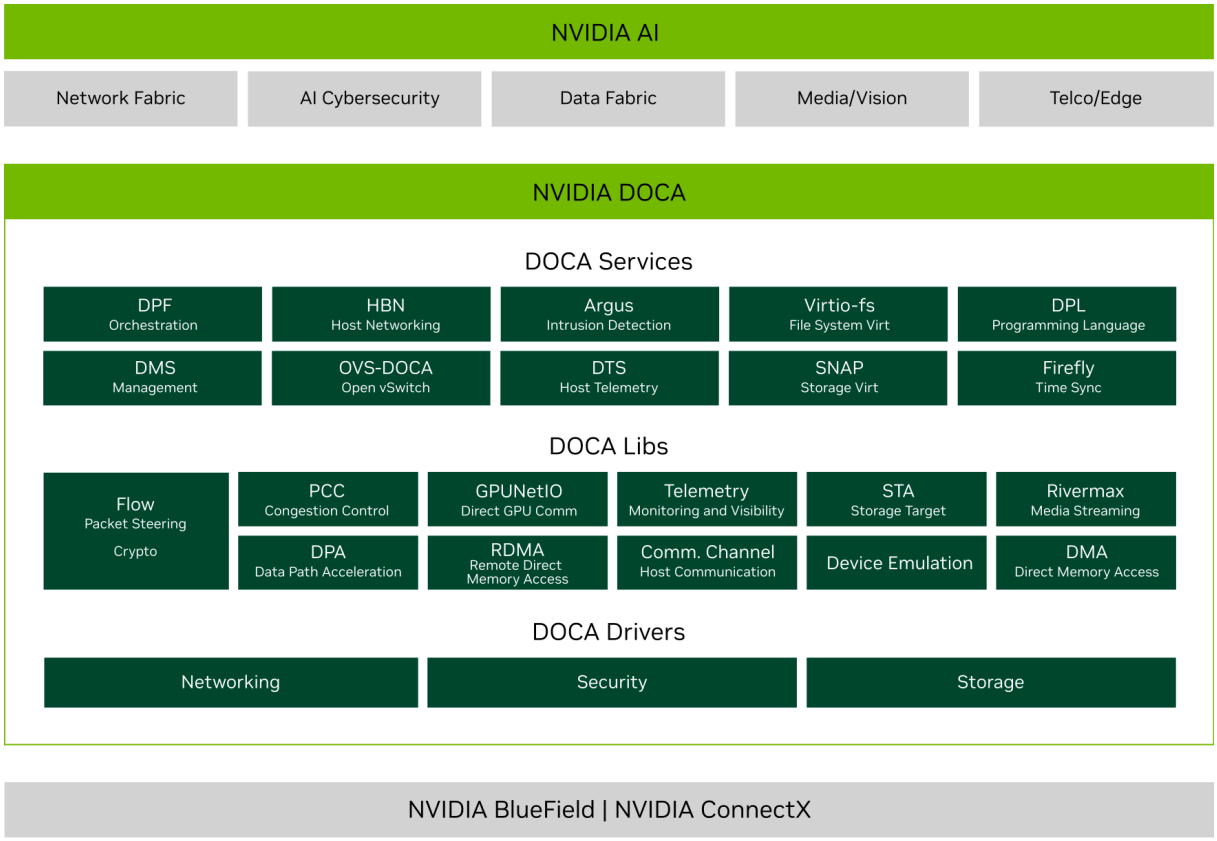
- The DOCA runtime drivers and libs installed on top of the BlueField Platform
- The OS installed on the BlueField Platform

- The BlueField Platform Software (i.e., firmware and UEFI bootloader)

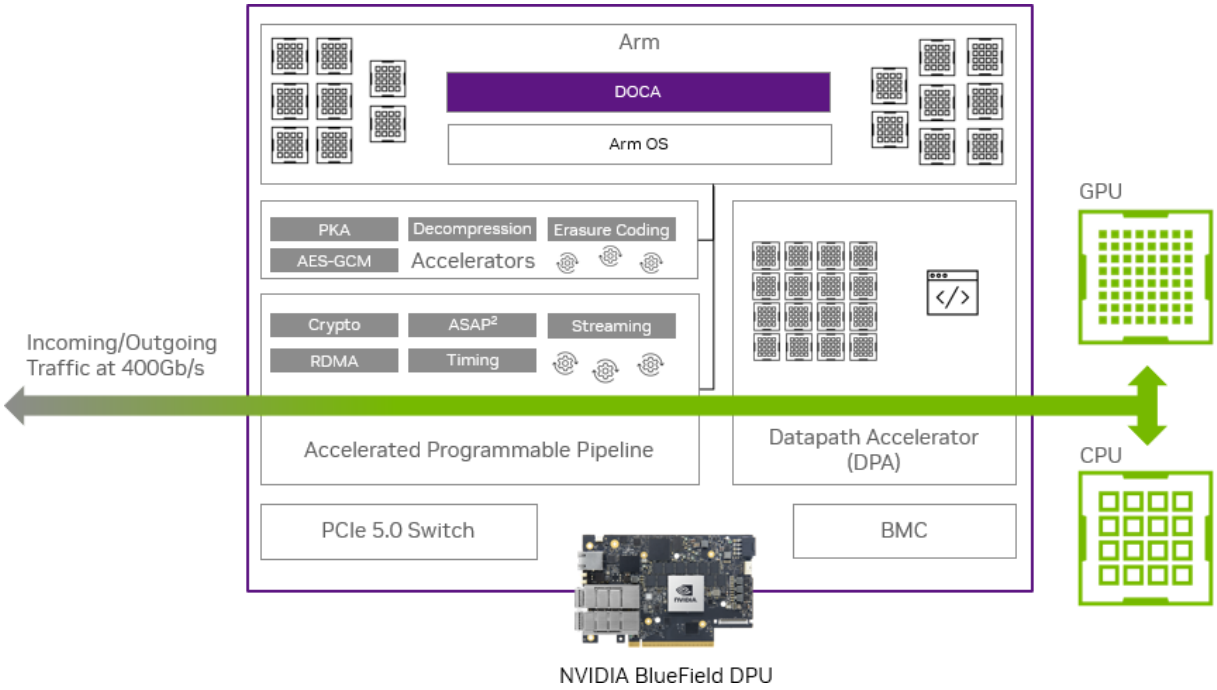
DOCA provides all the required libraries and drivers for hosts that include NVIDIA Networking platforms (i.e., BlueField and ConnectX) with a dedicated DOCA-Host package installation.



DOCA contains a runtime and development environment, including libraries and drivers for device management and programmability, for the host and as part of a BlueField Platform Software.



DOCA is the software infrastructure for BlueField's main hardware entities:



2.1.2 Installation

Installation instructions for both host and BlueField image can be found in the DOCA Installation Guide for Linux.

Whether DOCA has been installed on the host or on the BlueField networking platform, one can find the different DOCA components under the **/opt/mellanox/doca** directory. These include the traditional SDK-related components (libraries, header files, etc.) as well as the DOCA samples, applications, tools and more, as described in this document.

2.1.3 API

The DOCA SDK is built around the different DOCA libraries designed to leverage the capabilities of BlueField. Under the Programming Guide section, one can find a detailed description of each DOCA library, its goals, and API. These guides document DOCA's API, aiming to help develop DOCA-based programs.

The API References section holds the Doxygen-generated documentation of DOCA's official API.

2.1.4 Programming Guides

DOCA programming guides provide the full picture of DOCA libraries and their APIs. Each guide includes an introduction, architecture, API overview, and other library-specific information.

Each library's programming guide includes code snippets for achieving basic DOCA-based tasks. It is recommended to review these samples while going over the programming guide of the relevant

DOCA library to learn about its API. The samples provide an implementation example of a single feature of a given DOCA library.

For a more detailed reference of full DOCA-based programs that make use of multiple DOCA libraries, please refer to the Reference Applications.

2.1.5 Applications

Applications are a higher-level reference code than the samples and demonstrate how a full DOCA-based program can be built. In addition to the supplied source code and compilation definitions, the applications are also shipped in their compiled binary form. This is to allow users an out-of-the-box interaction with DOCA-based programs without the hassle of a developer-oriented compilation process.

Many DOCA applications combine the functionality of more than one DOCA library and offer an example implementation for common scenarios of interest to users such as application recognition according to incoming/outgoing traffic, scanning files using the hardware RegEx acceleration, and much more.

For more information about DOCA applications, refer to DOCA Reference Applications.

2.1.6 Tools

Some of the DOCA libraries are shipped alongside helper tools for both runtime and development. These tools are often an extension to the library's own API and bridge the gap between the library's expected input format and the input available to the users.

For more information about DOCA tools, refer to DOCA Tools.

2.1.7 Services

DOCA services are containerized DOCA-based programs that provide an end-to-end solution for a given use case. DOCA services are accessible as part of NVIDIA's container catalog (NGC) from which they can be easily deployed directly to BlueField, and sometimes also to the host.

For more information about container-based deployment to the BlueField Platform, refer to the DOCA Container Deployment Guide.

For more information about DOCA services, refer to the DOCA Services.

For questions, comments, and feedback, please contact us at doca-feedback@nvidia.com.

2.2 DOCA Release Notes

DOCA v3.5.0 is tested as part of Spectrum-X reference architecture release version 2.3. For a list of approved releases, refer to the NVIDIA Spectrum-X Validated Solution Stack.

NVIDIA DOCA™ SDK release notes contain information on new features, software interoperability, and known issues.

2.2.1 Introduction

The DOCA release notes contain the following subpages:

- Changes and New Features
- General Support
- Bug Fixes in This Version
- Known Issues
- BlueField Firmware Cables

2.2.2 Installation Notes

NVIDIA® BlueField®-3 devices are not supported with MLNX_OFED as the host driver. They must use DOCA-Host.

BlueField DPUs with the following SKUs require an 8-pin ATX power supply cable connection when powering up. Without this connection to the power supply cable, the device will not complete the power-on procedure and will not function properly.

- B3220 DPUs – 900-9D3B6-00CV-AA0 and 900-9D3B6-00SV-AA0
- B3240 DPUs – 900-9D3B6-00CN-AB0 and 900-9D3B6-00SN-AB0
- B3210 DPUs – 900-9D3B6-00CC-AA0 and 900-9D3B6-00SC-AA0
- B3210E DPUs – 900-9D3B6-00CC-EA0 and 900-9D3B6-00SC-EA0

Refer to the (3.4.0) DOCA Installation Guide for Linux for information on:

- Setting up the DOCA SDK on your BlueField networking platform or SmartNIC
- Supported BlueField platforms

DOCA Runtime and DOCA Devel (SDK)

By default, installing DOCA profiles with standard Linux tools (yum, apt) installs both **doca-runtime** and **doca-devel** (previously **doca-sdk**).

- **doca-runtime** includes all components, libraries, drivers, and tools used in a production environment by a DOCA admin
- **doca-devel** includes all components, libraries, drivers, and tools used for development, including reference applications, compilers, etc.

2.2.3 Supported Device Speeds

Uplink/Adapter Card	Driver Name	Uplink Speed
BlueField-3	mlx5	• InfiniBand: SDR, FDR, EDR, HDR100, HDR, NDR200, NDR • Ethernet: 1GbE, 10GbE, 25GbE, 40GbE, 50GbE, 100GbE, 200GbE, 400GbE

Uplink/Adapter Card	Driver Name	Uplink Speed
BlueField-2		• InfiniBand: SDR, FDR, EDR, HDR • Ethernet: 1GbE, 10GbE, 25GbE, 40GbE, 50GbE[1], 100GbE[1]
BlueField		• InfiniBand: SDR, QDR, FDR, FDR10, EDR • Ethernet: 1GbE, 10GbE, 25GbE, 40GbE, 50GbE, 100GbE
ConnectX-9		• InfiniBand: NDR, XDR • Ethernet: 100GbE, 200GbE, 400GbE, 800GbE
ConnectX-8		• InfiniBand: SDR, FDR, EDR, HDR100, HDR, NDR200, NDR, XDR • Ethernet: 1GbE, 10GbE, 25GbE, 40GbE, 50GbE, 100GbE, 200GbE, 400GbE
ConnectX-7		• InfiniBand: EDR, HDR100, HDR, NDR200, NDR • Ethernet: 1GbE, 10GbE, 25GbE, 40GbE, 50GbE[1], 100GbE[1], 200GbE[2], 400GbE
ConnectX-6 Lx		• Ethernet: 1GbE, 10GbE, 25GbE, 40GbE, 50GbE[1]
ConnectX-6 Dx		• Ethernet: 10GbE, 25GbE, 40GbE, 50GbE[1], 100GbE[1], 200GbE[1]
ConnectX-6		• InfiniBand: SDR, FDR, EDR, HDR • Ethernet: 10GbE, 25GbE, 40GbE, 50GbE[1], 100GbE[1], 200GbE[1]
ConnectX-5/ ConnectX-5 Ex		• InfiniBand: SDR, QDR, FDR, FDR10, EDR • Ethernet: 1GbE, 10GbE, 25GbE, 40GbE, 50GbE, 100GbE
ConnectX-4 Lx		• Ethernet: 1GbE, 10GbE, 25GbE, 40GbE, 50GbE
ConnectX-4		• InfiniBand: SDR, QDR, FDR, FDR10, EDR • Ethernet: 1GbE, 10GbE, 25GbE, 40GbE, 50GbE, 56GbE[3], 100GbE

[1] Speed that supports both NRZ and PAM4 modes in Force mode and Auto-Negotiation mode.

[2] Speed that supports PAM4 mode only.

[3] 56GbE is an NVIDIA proprietary link speed and can be achieved while connecting an NVIDIA adapter card to NVIDIA SX10XX switch series or when connecting an NVIDIA adapter card to another NVIDIA adapter card.

2.2.4 Technical Support

Customers who purchased NVIDIA products directly from NVIDIA can contact us through the following methods:

- E-mail: enterprisesupport@nvidia.com
- Enterprise Support Services: https://www.nvidia.com/en-us/support/enterprise

If you purchased NVIDIA M-1 Global Support Services, refer to your contract for technical support details.

Customers who purchased NVIDIA products through an NVIDIA-approved reseller should first seek assistance through their reseller.

2.2.5 Changes and New Features

If you are using BlueField FW Bundle/DOCA-Host as part of the Spectrum-X reference architecture, please refer to the NVIDIA Spectrum-X Validated Solution Stack page for the list of approved product releases and versions.

2.2.5.1 1. DOCA Framework New Features and Updates

Upcoming device support change: DOCA 3.6.0 (October 2026) will be the final DOCA release to support ConnectX-4 Lx and ConnectX-5 devices. As a Long-Term Support (LTS) release, DOCA 3.6.0 will receive bug fixes and security updates addressing CVEs for three years. Starting with DOCA 4.0, scheduled for release in January 2027, these devices will no longer be supported.

- DOCA Argus
  - Now provides key GPU metrics without a host tool/agent.
  - Now supports providing different “runtime manifests” for different containers.
  - Can send out data using vector observability tool.
- DOCA Bridge
  - Removes DOCA dependency on DPDK (for customers who do not need DPDK).
- DOCA Ethernet
  - Adds DOCA Ethernet header data-split support.
  - Adds DOCA Ethernet support for inlining data to WQE, improving performance.
- DOCA Management Service (DMS) *[TBC*]*
  - Adds a DMS-facing DOCA Core API to query **native_port_num** .*
  - Adds **doca_mgmt** API support to resolve **vhca_id** through **libdms** and CLI.*
  - Updates DMS provisioning flows to use native DMS APIs instead of gNOI.*
- OVS-DOCA
  - OVS-DOCA now supports IPv6 CT.
  - Adds an OVS-DOCA API to disable reset logic in DOCA-CT aging flows.
  - Adds OVS-DOCA support for the B3220SH Storage Controller.
  - Supports DOCA Ethernet for OVS-DOCA datapath.
  - Adds OVS-DOCA API support for configuring floating CPU behavior for PMD threads.
- DOCA Telemetry
  - DOCA Telemetry lib now exposes RTT counters (hardware congestion control counters).
- DOCA VERBs
  - New DOCA Multi-Rail API in DOCA VERBs to support NIC resiliency for VR [2:1] and LB for non-NVIDIA comm libs.

2.2.5.2 2. DOCA-Host and DOCA Drivers New Features and Updates

NVIDIA® BlueField®-3 networking platforms must use DOCA-Host as the host driver. MLNX_OFED does not support BlueField-3 devices.

- Adaptive-ReTransmission Related Q-Counters - Adaptive-ReTransmission-related Q-counters are now exposed to all function types, rather than being limited to Physical Functions (PFs) only. This allows any function to view and monitor these counters for improved visibility and diagnostics.
- Fast Registration Memory Region (FRMR) Pools to Replace MR Cache - Replaced the MR cache mechanism with the new generic Fast Registration Memory Region (FRMR) pools feature, providing improved control and visibility into driver optimizations for resource creation during memory registration. This enables users to control the pools, including their sizes and aging time, through the user-space RDMA tool in iproute2.

2.2.5.3 3. BlueField-3 Firmware Changes and New Features

- PCIe In-Band SPDM Attestation - Added capabilities that allow the host to use in-band channels over PCIe to reach the PSC and perform SPDM attestation.

2.2.5.4 4. BlueField-2 Firmware Changes and New Features

N/A

2.2.5.5 5. API Changes

The following table details the API changes that were introduced in this version of DOCA:

Library	Change Description
TBD	TBD

To view API changes for previous DOCA versions, refer to the release notes of those archived versions.

2.2.5.6 Customer Affecting Changes

2.2.5.6.1 Changes in This Release

Category	Description	Customer Impact and Recommendation
DOCA Apps 3.5.0	DPDK queue-count initialization in `dpdk_queues_and_ports_init()` is now strictly enforced; the configured queue count is no longer implicitly overridden by available lcores.	Apps/samples that set `reserve_main_thread=true` with `nb_queues=1` will now fail to initialize. Either set `reserve_main_thread=false` or use `nb_queues >= 2`.
DOCA Flow 3.5.0	Deprecation of system-wide resource APIs (stable): The following system-wide resource sizing and binding APIs are deprecated in DOCA 3.5: `doca_flow_cfg_set_nr_counters()`, `doca_flow_cfg_set_nr_meters()`, `doca_flow_cfg_set_nr_shared_resource()`, and `doca_flow_shared_resources_bind()`. These APIs configure a global resource pool. Applications should migrate to the per-port resource model.	Compiling source code that uses the deprecated APIs now produces deprecation warnings. ABI compatibility is unchanged in this release, and the corresponding symbols remain exported, so existing binaries continue to link and run without modification. Migrate to the following per-port APIs: • Use `doca_flow_port_cfg_set_nr_resources()` to configure resource capacity, such as `DOCA_FLOW_RESOURCE_COUNTER` and `DOCA_FLOW_RESOURCE_METER`. • Use `doca_flow_port_shared_resource_get()` and `doca_flow_port_shared_resource_put()` to allocate and release shared resources. Before calling `doca_flow_init()`, enable the per-port resource model: `doca_flow_cfg_set_resource_mode(cfg, DOCA_FLOW_RESOURCE_MODE_PORT)` In DOCA 3.5, the replacement APIs are experimental and produce an experimental-API compilation warning. Define `DOCA_ALLOW_EXPERIMENTAL_API` to acknowledge and suppress this warning. These APIs may change until they are promoted to stable. During migration, define `DOCA_ALLOW_DEPRECATED_API` to suppress warnings from the legacy APIs temporarily.

2.2.5.6.2 Changes Planned for Future Releases

Planned for Version	Description	Customer Impact and Recommendation

DOCA 3.6 (Oct 2026)	Removing **ofed_scripts** Package in DOCA-Host	the **ofed_scripts** package in DOCA-Host is formally deprecated and will be removed from all DOCA-Host profiles/packs.
		This removal impacts the following components:
		• **ofed_info** / **ofed_rpm_info**
		• **ofed_uninstall.sh**
		• **sysinfo-snapshot.py**
		• **install.pl**
		• **vendor_pre_uninstall.sh** / **vendor_post_uninstall.sh**
		ℹ The **openibd** script is not a part of **ofed_scripts** and is not affected by this change.
		Migration path:
		• Instead of **ofed_info**, use native Linux package manager metadata to query DOCA-Host meta-packages:
		  • Ubuntu/Debian:
		  `apt info doca-networking`
		  • RHEL/Rocky:
		  `yum info doca-networking`
		• Instead of **sysinfo-snapshot.py**, migrate to **doca-sosreport**.
		• Regarding **ofed_uninstall**: The legacy script is preserved in older versions for backward compatibility during initial installations, but it is not used for upgrades between DOCA-Host versions (DOCA-Host 2.6 and higher).

Planned for Version	Description	Customer Impact and Recommendation

	Manual Enablement of SF Auxiliary Devices:
	With the new implementation, mlxdevm Sub-Functions are no longer probed automatically with all auxiliary sub-devices, such as Ethernet and RDMA.
	Previously, when an SF was created and activated, all relevant auxiliary sub-devices were probed by default. Now, the SF is initially probed without auxiliary sub-devices, and each required auxiliary device must be enabled manually.
		To manually enable each required auxiliary device, the user must:
		1. Create and activate the SF.
		2. Enable the Ethernet auxiliary device.
		3. Reload the auxiliary device to fully probe the SF after enabling the required auxiliary device.
		For example:
		1. Create and activate the SF.
		`/opt/mellanox/iproute2/sbin/mlxdevm port add pci/0000:08:00.0 flavour pcisf pfnum 0 sfnum 11`
		`/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:08:00.0/32768 state active`
		2. Enable the Ethernet auxiliary device.
		`/opt/mellanox/iproute2/sbin/mlxdevm dev param set auxiliary/mlx5_core.sf.1 name enable_eth value true cmode driverinit`
		3. Reload the auxiliary device to fully probe the SF after enabling the required auxiliary device.
		`/opt/mellanox/iproute2/sbin/devlink dev reload auxiliary/mlx5_core.sf.1`

	RoCE Configuration Interface Update: The sysfs interface previously used to configure the RoCE state (/sys/bus/pci/devices/0000:00:08.0/roce_enable) will no longer be supported.	Users should now use mlxdevm to enable or disable RoCE on a port. Example: `mlxdevm dev param set pci/0000:08:00.0 roce <enable|disable>`

Planned for Version	Description	Customer Impact and Recommendation

	MAC Address Configuration Interface Update: MAC address configuration through sysfs is being deprecated. Users should use devlink or mlxdevm to configure the MAC address for the ECPF, PF vport, or VF vport.	To configure a MAC address using **devlink** or **mlxdevm**:
		1. Discover the relevant port index.
		`devlink port show`
		Example:
		`pci/0000:03:00.0/65536: flavour pci_pf controller 1 pf 0 external`
		`pci/0000:03:00.0/65537: flavour pci_vf controller 1 pf 0 vf 0 external`
		`pci/0000:03:00.0/65538: flavour pci_vf controller 1 pf 0 vf 1 external`
		2. Set the MAC address using the relevant port index.
		`devlink port function set pci/0000:03:00.0/ <port_index> hw_addr <mac>`
		Example mappings:
		`Host PF vport 0 pci/0000:03:00.0/65536`
		`Host VF 0 vport 1 pci/0000:03:00.0/65537`
		`Host VF 1 vport 2 pci/0000:03:00.0/65538`

	Support for App Shield, BlueMan, and Flow Inspector will be discontinued.	Support for the DOCA Flow Inspector service, DOCA BlueMan service, and DOCA AppShield library will be discontinued in DOCA 3.6. DOCA 3.5.0 is the last release that supports these components.
	DOCA 3.6 (Oct 2026 release) will be the last DOCA version to support ConnectX-4 Lx and ConnectX-5. DOCA 3.6 will be an LTS version supported for 3 years for bug fixes and CVE updates. DOCA 4.0 (Jan 2027 release) will not support these devices.	
	DOCA 3.5 July 2026 release will be the last to support BFB-Install tool. in DOCA 3.6 Oct 2026 DOCA-Installer will replace the fun functionality currently provided by BFB-Install	
DOCA 4.0 (Jan 2027)	DOCA-Flow - Support of Crypto and NAT64 actions will be removed from Basic pipe.	For Crypto and NAT64 functionality customers should use **Ordered_List** pipe.

Planned for Version	Description	Customer Impact and Recommendation
	Removal of system-wide resource APIs: In DOCA 4.0, the declarations for the following system-wide shared-resource APIs are removed from `doca_flow.h`: • `doca_flow_cfg_set_nr_counters()` • `doca_flow_cfg_set_nr_meters()` • `doca_flow_cfg_set_nr_shared_resource()` • `doca_flow_shared_resources_bind()` • `doca_flow_shared_resource_set_cfg()` • `doca_flow_shared_resources_query()` The declaration for `doca_flow_cfg_set_resource_mode()` and the definition of `enum doca_flow_resource_mode` are also removed. The corresponding symbols remain in the library to preserve binary compatibility for applications built against earlier headers. For applications compiled against the DOCA 4.0 headers, per-port shared resources are the default and only supported resource model.	Existing binaries are unaffected. Applications built against pre-4.0 headers continue to link and run against DOCA 4.0 without relinking or behavior changes. The legacy symbols remain exported, and the system-wide resource behavior is preserved for these binaries. Source migration is required. Code rebuilt against the DOCA 4.0 headers no longer compiles if it calls the removed APIs or references `enum doca_flow_resource_mode` or `DOCA_FLOW_RESOURCE_MODE_*`. Migrate the source code to the per-port APIs before rebuilding it against DOCA 4.0. Per-port resource management is the default and only supported model in DOCA 4.0, so remove calls to `doca_flow_cfg_set_resource_mode()` and all related resource-mode references. Use the following replacements: • `doca_flow_cfg_set_nr_counters()`, `doca_flow_cfg_set_nr_meters()`, and `doca_flow_cfg_set_nr_shared_resource()` → `doca_flow_port_cfg_set_nr_resources()` • `doca_flow_shared_resources_bind()` → `doca_flow_port_shared_resource_get()` and `doca_flow_port_shared_resource_put()` • `doca_flow_shared_resource_set_cfg()` → `doca_flow_port_shared_resource_set_cfg()` • `doca_flow_shared_resources_query()` → `doca_flow_port_shared_resources_query()` • `doca_flow_cfg_set_resource_mode()` and `enum doca_flow_resource_mode` → no replacement; remove the call and any `DOCA_FLOW_RESOURCE_MODE_*` references Migration requirements: • Configure resource capacity on the port configuration before calling `doca_flow_port_start()`. • After the port starts, allocate resource IDs with `doca_flow_port_shared_resource_get()`, configure and query the resources through the corresponding per-port APIs, and release the IDs with `doca_flow_port_shared_resource_put()`.

Planned for Version	Description	Customer Impact and Recommendation
		• Translate resource types by name rather than by numeric value. `enum doca_flow_resource` and `enum doca_flow_shared_resource_type` are not numerically aligned; `DOCA_FLOW_RESOURCE_COUNTER_CT` shifts the values that follow `IPSEC_SA`. • Resource IDs are scoped by port and resource type. They start at `0`, may be reused after release, and are not guaranteed to increase monotonically. `doca_flow_port_shared_resource_get()` returns `DOCA_ERROR_EMPTY` when the configured capacity for that resource type is exhausted. • In DOCA 3.5.0, the per-port APIs require calling `doca_flow_cfg_set_resource_mode(cfg, DOCA_FLOW_RESOURCE_MODE_PORT)` before `doca_flow_init()`. This call is not required in DOCA 4.0. Isolate it if the source must support both releases. For implementation examples, see `samples/doca_flow/flow_shared_counter` and `samples/doca_flow/flow_shared_meter`.
DOCA 4.0 (Jan 2027)	DOCA-Host introduction of new Host Packs to replace current DOCA-Host Profiles. DOCA-Host-Packs allows better resolution of DOCA installation, where each Pack includes a smaller part of DOCA libs. Packs can be installed alone, or more than one on a host. DOCA-Host-Packs will include: • DOCA-Host-Base • DOCA-Networking-Core • DOCA-GPU-Net • DOCA-Storage • DOCA-Scientific-AI • DOCA-All • DOCA-Tools	Customer automation and installation flows must be updates with the new DOCA-Host-Packs. To align with existing DOCA-Host Profiles, users should install: • DOCA-OFED – install DOCA-Host-Base + DOCA-Tools • DOCA-Networking – install DOCA-Networking-Core + DOCA Tools • DOCA-All – install DOCA-All
October 2027	ConnectX-6 Adapters and BlueField-2 DPU Adapters End-of-Life	October 2027 DOCA release will be the last to support ConnectX-6 and BlueField-2. This version will be an LTS version supported for 3 years. The following release (January 2028) will no longer support these devices.

2.2.5.6.3 Changes from Earlier Releases

This section provides a list of changes that took place throughout the past two major releases that broke compatibility/interface or discontinued support for features or OS versions.

> For older changes, consult the DOCA documentation archive.

Introduced in Version	Description	Customer Impact and Recommendation
DOCA 3.4 (Apr 2026)	Virtio-net and virtio-blk support for BlueField-2 devices will be removed from DOCA GA release. BlueField-2 virtio-net will continue to be available in the current LTS branches (DOCA 2.9.x and DOCA 3.2.x).	BlueField-2 customers interested in using virtio-net should use one of the LTS branches.
DOCA-Host 3.3.0 (Jan 2026)	The GPG key used to sign the DOCA-Host Linux repository has been rotated to a new, stronger key. The key identity has also changed from "Mellanox Technologies" to "NVIDIA DOCA Host."	During installation or upgrade, users may be prompted to confirm and import the new repository key; once accepted, the process will proceed normally.
	DOCA-Host 3.3.0 no longer supports the end-of-life Ubuntu 20.04 , Ubuntu 25.04, Azure Linux 2.0, OpenEuler 20.03, Debian 10.13, Xenserver 8.2, BCLinux 21.10 SP2.	
	The SHARP package has been removed from all DOCA-Host deliverables.	
	The SYSFS interface for MR cache configuration will be deprecated and replaced by the `rdma` resource command.	Any users or scripts relying on this SYSFS interface must be updated to use the new `rdma` command API.
	The knem module has been deprecated and replaced by xpmem.	

Introduced in Version	Description	Customer Impact and Recommendation
DOCA-Host 3.3.0 (Jan 2026)	Transition to 2023 Microsoft UEFI Certificate Authority To align with updated Microsoft UEFI Secure Boot requirements and the upcoming end-of-life of the 2011 Certificate Authority (CA), NVIDIA is transitioning to the 2023 CA. To ensure successful loading of the Expansion ROM (ExpROM) during the UEFI Secure Boot process, system BIOS and operating system trust stores must be updated to include the 2023 CA.	When performing a firmware update of ConnectX and BlueField devices the new certificate is required for Secure Boot. To continue supporting Secure Boot, systems must be updated to recognize the "Microsoft Option ROM UEFI CA 2023."
DPA 3.2.0	DPA Application Attributes: To enhance security, DPA application attributes are being introduced. Developers must now create a dpa-app-attributes file for each application, which declares the app's permissions and must be compiled with the application.	When updating the BF-Bundle (or ConnectX firmware), users must also update the corresponding DOCA Host components. As part of this update, each DPA application must be rebuilt with its `dpa-app-attributes` file. ⚠ If a file is not supplied, a default manifest enabling all permissions will be applied automatically.
	Undocumented and previously deprecated DPA/FlexIO APIs will be removed.	Users who utilized these APIs must remove them from their code and re-build their applications.
DOCA-Host 3.2.0	OpenSM support is removed from DOCA-Host for Xenserver operating systems.	No impact.
	The HCOLL and MPItests packages have been removed from all DOCA-Host deliverables.	No impact listed.
	The `phy_stats` sysfs counters are deprecated. • Deprecated: `/sys/class/net/<iface>/phy_stats/` • Replacement: Use `ethtool -S <iface>` and filter for `packets_phy` or `bytes_phy`.	No impact listed; recommendation is in the description.
	The has_smi sysfs entry is deprecated. • Deprecated: `/sys/class/infiniband/mlx5_0/ports/1/has_smi` • Replacement: Use the `ib_stat` tool	Users should parse the `ib_stat` output for the "Capability mask" value: • Bit #1 indicates `IsSM` • Bit #10 indicates `IsSMdisabled`

Introduced in Version	Description	Customer Impact and Recommendation
	The hfunc sysfs entry for setting/showing hash functions is deprecated. • Show (Deprecated): `cat /sys/class/net/<iface>/settings/hfunc` • Show (Replacement): `ethtool -x --show-rxfh <devname>` • Set (Deprecated): `echo [hfunc] > /sys/class/net/<iface>/settings/hfunc` • Set (Replacement): `ethtool -X --rxfh <devname> [hfunc FUNC]`	No impact listed; recommendation is in the description.
	The `pfc_stall_prevention` sysfs entry is deprecated. • Show (Deprecated): `cat /sys/class/net/<iface>/settings/pfc_stall_prevention` • Show (Replacement): `ethtool --get-tunable <iface> pfc-prevention-tout` • Set (Deprecated): `echo [default/auto] > /sys/class/net/<iface>/settings/pfc_stall_prevention` ℹ `default` is mapped to 8000 msec per critical prevention. `auto` is mapped to 100 msec per critical prevention. • Set (Replacement): `ethtool -X --set-tunable <devname> [pfc-prevention-tout N]`	No impact listed; recommendation is in the description.
	The sysfs entry for displaying VF statistics is deprecated. • Deprecated: `/sys/class/infiniband/mlx5_2/device/sriov/2/stats`	Users should use the `ip` command instead: `/opt/mellanox/iproute2/sbin/ip -s link show dev <VF-interface-name>`
	VGT+ capability will be unsupported as of October 2025.	No impact listed.

Introduced in Version	Description	Customer Impact and Recommendation
	The "Per Channel Statistics" `ethtool` private-flag will be unsupported as of October 2025.	No impact listed.
	The `commands_cache` sysfs entry is deprecated.	No impact listed.
	Dynamic Kernel Module Support (DKMS) is now integrated, allowing customers to build their own DOCA-Host packages.	As part of the DKMS build process, driver signing is removed. Customers must sign the drivers themselves.
	The `mlxdevm` tool, part of DOCA-Host, is deprecated.	Use the `devlink` tool, which is included in DOCA-Host. `mlxdevm` is redundant as it provides the exact same functionality.
	Backward compatibility with older MFT (NVIDIA Firmware Tools) versions 4.31.0-149 and 4.30.0-139 is no longer supported.	Using these unsupported MFT versions may lead to unexpected behavior or prevent device detection.
DOCA-Host 3.1.0	In the DOCA-OFED profile, the `openvswitch` component was deprecated and replaced with the `doca-openvswitch` component.	The content and features are identical, but customers using the `openvswitch` component must re-build their applications with the new `doca-openvswitch` package.
	The "Shared RQ" feature was replaced by the "memory consumption reduction for representors" feature.	As a result, the `esw_pet_insert` devlink parameter is no longer available, and setting `ALLOW_SHARED_RQ` in `/etc/mellanox/mlnx-bf.conf` has no effect.
DOCA-Host 3.0.0 (Apr 2025)	The `mlxdevm` tool was aligned with `devlink` v6.12. In this version, a zero-rate value indicates "unlimited" and is omitted from the display.	When using `mlxdevm port function rate show`, the `tx_rate` and `tx_max` fields are omitted if their value is zero. The driver also only accepts `tx_rate` and `tx_max` values that are multiples of 1 Mb/s.
	The `mlxdevm` tool was aligned with `devlink` v6.12 and conforms to its `man` page specifications.	To set rate, use: `port function rate set [DEV/PORT_INDEX | DEV/NODE_NAME] tx_max VALUE`
DOCA-Host 2.10.0 (Jan 2025)	When Dynamic Interrupt Moderation (DIM) is enabled, static coalescing parameters cannot be set because the dynamic algorithm will override their values.	To set static coalescing parameters, you must first disable DIM.
	The ConnectX-4 adapter card family is no longer supported.	N/A
	Support was removed for several OS versions (e.g., RHEL 8.0, 8.1, 8.3, 8.5, 8.7, 9.1, 9.3, and OL 7.9).	N/A

Introduced in Version	Description	Customer Impact and Recommendation
DOCA-Flow 2.10.0	Several features were temporarily unsupported in the 2.10 release, including ACL pipe, LPM pipe, CT pipe, `ordered_list` pipe, external send queue (SQ), and pipe resize.	Users needing these features were advised to use DOCA Flow 2.9.1.
	The requirement to use a `dummy_id` for IPsec encryption actions during pipe creation was removed.	When creating a pipe with a crypto action, `UINT32_MAX` now represents a changeable shared object, while a `crypto_id` between 0 and `UINT32_MAX` indicates a constant.
	Memory for "modify field" and "encap" actions must be allocated upfront on a per-port basis.	If these actions are used, the `doca_flow_port_cfg_set_actions_mem_size()` function must be called to configure the memory size.
	"Strict matching" is no longer supported.	The `doca_flow_pipe_cfg_set_enable_strict_matching()` function is deprecated. Users must use "relaxed matching".
	The entry aging mechanism no longer relies on DPDK. The `service_core=` and `svc_cycle_time=` DPDK arguments are no longer supported.	Users must use the new APIs: `doca_flow_port_cfg_set_service_threads_core()` and `doca_flow_port_cfg_set_service_threads_interval()`.
	The `doca_flow_parser_meta.random` field was changed to big-endian.	Users must update code to provide this value in big-endian format. The `DOCA_HTOBE16` macro is recommended.
	The RSS config API was refactored to align with other resource types.	Users must now specify whether the RSS resource is shared or non-shared.
	Values for the `doca_flow_l2_meta` and `doca_flow_meter_color` enums were changed.	Users must recompile their applications.
	The match fields `parser_meta.outer_l4_ok` and `parser_meta.inner_l4_ok` no longer validate the L4 checksum.	To check for a valid checksum, users must use `parser_meta.outer_l4_checksum_ok` and `parser_meta.inner_l4_checksum_ok`.
	Entry "life event" callback was previously invoked only on "entry add" and "entry removal".	The callback is now also invoked upon "port stop" during the removal of pipe entries. Therefore, the user context must still be defined at that stage.
DOCA-ETH 2.10.0	The `eth_rxq_managed_mempool_receive` and `eth_rxq_regular_receive` samples were not supported.	Users needing these samples were advised to use DOCA 2.9.1.

Introduced in Version	Description	Customer Impact and Recommendation
DOCA 2.9.0 (Oct 2024)	DPA Outbox Blocking-Mode: Due to a silicon issue (as of firmware v28.43.2026), the DPA outbox was set to non-blocking mode, causing requests to complete immediately without waiting for the actual operation to finish.	The DPA stack must poll a "busy" bit before starting another operation. Users programming the DPA must update firmware to 28.43.2026+ and DOCA-Host to 2.9.x+.
	DPA Thread Context: As of firmware v28.43.2026, internal DPA stack API changes altered the DPA thread context, impacting the overlying DPA stack.	This requires the same action as the Outbox Blocking-Mode: update firmware to 28.43.2026+ and DOCA-Host to 2.9.x+.

2.2.5.6.4 Discontinued Features

List of features which are supported in previous generations of hardware devices:

- N/A

2.2.5.7 Declared Unsupported Features

This section provides a list of features that are not supported in this release:

- ETS and VF rate limit (Cross E-Switch Scheduling)
- CPPC CPU frequency monitoring

2.2.6 General Support

2.2.6.1 Supported Platforms and NIC Firmware Versions

> ⚠ NVIDIA® BlueField®-2 and NVIDIA® ConnectX®-7 are the last generation of BlueField and ConnectX families that are supported by MLNX_OFED.

Adapter Card	Bundled Firmware Version
BlueField-3	32.50.1002
BlueField-2	24.50.1002
ConnectX-9	82.50.1002
ConnectX-8	40.50.1002
ConnectX-7	28.50.1002
ConnectX-6 Lx	26.50.1002
ConnectX-6 Dx	22.50.1002

Adapter Card	Bundled Firmware Version
ConnectX-6	20.43.8004
ConnectX-5/ConnectX-5 Ex	16.35.8008
ConnectX-4 Lx	14.32.1912

For official firmware versions, refer to Firmware Downloads.

2.2.6.2 Embedded DOCA BlueField System Components

Component	Version	Description
BMC firmware	26.07-7	BlueField BMC firmware
BlueField-3 eROT (Glacier)	00.02.0208.0000	BlueField-3 ERoT firmware
BlueField-2 eROT (CEC)	04.0f	BlueField-2 ERoT firmware
ATF	4.16.0-8-gda930bca9	Bootloader which initializes the Arm cores and manages the secure boot process
UEFI	4.16.0-48-g3e1cb60aa0	Standardized firmware interface for booting the system and initializing hardware before the OS loads
BSP	4.16.0.14075	Drivers and configurations required to initialize and support the BlueField platform's hardware

2.2.6.3 Embedded DOCA Drivers

Component	Version	Description	License
collectx-clxapi	1.26.3-1	CollectX API library which allows any 3rd party to easily use CollectX functionality in their own programs	BSD-3-Clause
dpacc	2.3.0.10-1	High-level compiler for the DPA processor which compiles code targeted for the data-path accelerator (DPA) processor into a device executable and generates a DPA program	Apache-2.0 with LLVM exception and NVIDIA Proprietary License and GCC-exception-3.1
dpcp	1.1.65-1	Provides a unified flexible interface for programming IB devices using DevX	BSD-3-Clause
flexio-sdk	26.07.3297-1	SDK which exposes an API for managing the device and executing native code over the DPA processor	Proprietary
ibarr	2607.0.0-1	ip2gid address resolution and gid2lid path record resolution	(GPL-2.0 WITH Linux-syscall-note) OR BSD-2-Clause

Component	Version	Description	License
ibdump	6.0.0-4.37.0.1.101.g1ed852f	Dump of InfiniBand traffic; diagnostic tool	BSD2+GPL2
ibsim	0.12.1-5	Open-source InfiniBand fabric simulator	GPLv2 or BSD
ibutils2	2.1.1-0.22700.MLNX20260722181 4.gd8ddfe9f2	Utilities for InfiniBand	Mellanox Confidential and Proprietary
iser	2607.0.100-1	Storage related drivers	GPLv2
isert	2607.0.100-1	Storage related drivers	GPLv2
kernel-mft	4.37.0-154	Kernel part of MFT tools (for firmware burning, etc.)	Dual BSD/GPL
libvma	9.8.84-1	Accelerates latency-sensitive and throughput-demanding TCP and UDP socket-based applications by offloading traffic from the user-space directly to the NIC or HCA	GPLv2 or BSD
libxlio	3.71.1-1	Boosts the performance of TCP/IP applications based on NGINX (CDN, DoH, etc.) and storage solutions as part of the SPDK	GPLv2 or BSD
mft	4.37.0-154	Set of firmware management and debug tools for NVIDIA devices	Proprietary
mlnx-ethtool	2607.0.2-1	Ethtool with optional MLNX adaptation	GPL
mlnx-iproute2	2607.0.3-1	IPRoute with optional MLNX adaptation	GPL
mlnx-nfsrdma	2607.0.100-1	Storage related driver for NFS over RDMA	GPLv2
mlnx-nvme	2607.0.100-1	Storage related driver for NVMe	GPLv2
mlnx-ofa_kernel	2607.0.100-1	Kernel drivers for Ethernet InfiniBand together	GPLv2
mlnx-tools	2607.0.6-1	Tools for loading modules, configurations, scripts, etc.	GPLv2 or BSD
mlx-steering-dump	2604.0.2-1	Hardware/software steering dump parsing tools	GPLv2

Component	Version	Description	License
multiperf	3.0-3.0.2607076	Linux tool for perf testing	BSD 3-Clause, GPL v2 or later
ofed-scripts	26.07-OFED.26.07.0.7.6	Scripts used to build OFED	GPL/BSD
openmpi	5.0.10rc2.2608140153-1.3c5e09e178	MPI implementation (for RDMA/RoCE) with some improvements done by the HPC team	BSD
opensm	5.28.1-20260 7221814.3ae6a892	InfiniBand Subnet Manager and Subnet Administrator based on OpenSM	GPLv2 or BSD
doca-openvswitch	3.5.0052-1	OVS (virtual switch), DPDK based	Proprietary
perftest	26.07.7-1	Test suite for performance	BSD 3-Clause, GPL v2 or later
rdma-core	2607.0.8-1	Implementation of the RDMA verbs	GPLv2 or BSD
rshim	2.8.5-1	User-space driver to access the BlueField SoC via the RShim interface, providing ways to push boot stream, debug the target, or login via the virtual console or network interface	GPLv2
sockperf	3.10-1	Network benchmarking utility over socket API UDP/ TCP designed for testing network performance (latency and throughput)	BSD-3-Clause
srp	2607.0.100-1	Storage-related driver for SCSI RDMA Protocol initiator	GPLv2
ucx	1.22.0.2608140153-1.8a6b06fb8	High-level application-oriented API for high-performance communication over RDMA networks	BSD
xpmem	2607.0.7-1	Kernel module to enable inter-process mapping for memory copy in user space	GPLv2 and LGPLv2.1

2.2.6.4 CUDA Compatibility with DOCA

This version of DOCA is compatible with CUDA 13, and built with CUDA 13.2.

2.2.6.5 DOCA Packages

Device	Component	Version	Description
Host	DOCA Devel	3.5.0	Software development kit package and tools for developing host software

Device	Component	Version	Description
	DOCA Runtime	3.5.0	Runtime libraries and tools required to run DOCA-based software applications on host
	DOCA Extra	3.5.0	Contains helper scripts (doca-info, doca-kernel-support)
	DOCA OFED	3.5.0	Software stack which operates across all NVIDIA network adapter solutions
Target BlueField DPU (Arm)	BlueField BSP	4.16.0	BlueField image and firmware
	DOCA SDK	3.5.0	Software development kit packages and tools for developing Arm software
	DOCA Runtime	3.5.0	Runtime libraries and tools required to run DOCA-based software applications on Arm

2.2.6.6 BlueField Operating System

The default operating system provided with the BlueField bundle (for DPU and SuperNIC) is Ubuntu 24.04 64k.

Primary BlueField operating systems are supported and require users to build their own BF-Bundle (BFB) with the binary DOCA package provided by NVIDIA. For instruction on how to build your own BF-Bundle please see bfb-build.

Primary OS supported with this DOCA release:

OS and version	Kernel
Ubuntu 24.04	6.8
Ubuntu 22.04	5.15
Rocky Linux 9.2	5.14
Debian 13	6.12
Debian 12	5.15 / 6.1
OpenEular 22.03 SP3	5.10
Anolis 8.6	5.10
CTYunOS 23.01	5.10
RHEL 9.6	5.14
RHEL 10.2	6.12
OL 8	5.15
OL 9	5.15

2.2.6.7 Supported Host OS per DOCA-Host Installation Profile

As of DOCA 3.3.0, NVIDIA will leverage DKMS to build and install host drivers from source on customer machines. Dynamic Kernel Module Support (DKMS) is a program/framework that enables generating Linux kernel modules whose sources generally reside outside the kernel source tree. This would affect the users as follows:

- NVIDIA no longer ships prebuilt and signed drivers, now only the source code for those drivers is provided and DKMS will build and install those drivers on the customers' machines during DOCA-Host installation.
- Furthermore, this means that NVIDIA no longer provides DOCA-Host for minor OS releases (e.g. RHEL9.X). We only provide major OS releases builds, and DKMS will handle the build and installation of the kernel modules for the different minor releases.
- Since kernel modules are built on client machines from source, NVIDIA can no longer sign the drivers.

DOCA-Host for FIPS enabled operating systems - DOCA-Host packages use SHA-256 for package signing as required by FIPS and operating systems that are FIPS compliant. Installing DOCA-Host will not impact the FIPS certificate of the OS.

The default operating system included with the BlueField bundle (for DPU and SuperNIC) is Ubuntu 24.04.

The supported operating systems on the host machine per DOCA-Host installation profile are the following:

> ⚠ Only the following generic kernel versions are supported for DOCA local repo package for host installation.

> ⚠ Dynamic Kernel Module Support (DKMS) in DOCA-Host includes the minor versions of Major versions with .x in the version. Minor versions that are EOL by the OS distribution are not support by DOCA.

OS	OS Version	Tested Kernel	Arch	doca-ofed/ doca-roce	doca-networking	doca-all
Ubuntu	26.04	kernel 7.0.0-14-generic	aarch64	✓	✓	✓
		kernel 7.0.0-14-generic	x86	✓	✓	✓
		kernel 7.0.0-14-generic	ppc64le	✓	X	X
	24.04.x (x<=4)	6.8.0, 6.14-HWE	aarch64	✓	✓	✓
		6.8.0, 6.14-HWE	x86	✓	✓	✓
		6.8.0	ppc64le	✓	X	X

OS	OS Version	Tested Kernel	Arch	doca-ofed/ doca-roce	doca-networking	doca-all
	22.04.x (x<=5)	5.15.0, 6.8-HWE	aarch64	✓	✓	✓
		5.15.0, 6.8-HWE	x86	✓	✓	✓
		5.15.0	ppc64le	✓	X	X
RHEL/ Rocky	10.x (x<=2)	6.12.0-211.7.1.el10_2	aarch64	✓	✓	✓
		6.12.0-211.7.1.el10_2	x86	✓	✓	✓
		6.12.0-211.7.1.el10_2	ppc64le	✓	X	X
	9.x (x<=8)	5.14.0-687.5.1.el9_8	aarch64	✓	✓	✓
		5.14.0-687.5.1.el9_8	x86	✓	✓	✓
		5.14.0-687.5.1.el9_8	ppc64le	✓	X	X
	8.x (x=10)	4.18.0-553.el8_10.aarch64	aarch64	✓	✓	✓
		4.18.0-553.el8_10.x86_64	x86	✓	✓	✓
		4.18.0-553.el8_10.ppc64le	ppc64le	✓	X	X
Debian	13.x (x<=6)	6.12.101	aarch64	✓	X	✓
		6.12.101	x86	✓	X	✓
	12.x (x<=15)	6.1.0-49	aarch64	✓	X	✓
		6.1.0-49	x86	✓	✓	✓
SLES	15 SP6/ SP7	6.4.0-150700.51-default	aarch64	✓	X	✓
		6.4.0-150700.51-default	x86	✓	X	✓
		6.4.0-150700.51-default	ppc64le	✓	X	X
	16 SP0	6.12.0-160000.5	aarch64	✓	X	✓
		6.12.0-160000.5	x86	✓	X	✓
		6.12.0-160000.5	ppc64le	✓	X	X
Oracle Linux	9	6.12.0-105.51.5.el9uek	aarch64	✓	X	X
		6.12.0-105.51.5.el9uek	x86	✓	X	X
	8	5.15.0-206.153.7.1.el8uek.x86_64	x86	✓	✓	✓

OS	OS Version	Tested Kernel	Arch	doca-ofed/ doca-roce	doca-networking	doca-all
Azure Linux	3.0	6.6.57.1-2.azl3	aarch64	✓	X	X
		6.6.35.1-5.azl3	x86	✓	X	X
Amazon Linux	2023	6.1.147-172.266.aarch64	aarch64	✓	✓	X
		6.1.147-172.266.x86_64	x86	✓	✓	X
Kylin	V11-2503	6.6.0-32.7.v2505.ky11.aarch64	aarch64	✓	X	X
		6.6.0-32.7.v2505.ky11.x86_64	x86	✓	X	X
Alinux	3.12	6.6.63-test.1.al8.aarch64	aarch64	✓	X	✓
	3.2	5.10.134-16.3.al8.aarch64	aarch64	✓	X	X
		5.10.134-13.al8.x86_64	x86	✓	✓	✓
Anolis	8.6	5.10.134+	aarch64	✓	X	X
		5.10.134+	x86	✓	X	X
BCLinux	22.10	5.10.0-153.24.0.100.6.oe2203sp2.bclinux.aarch64	aarch64	✓	X	X
		5.10.0-153.24.0.100.6.oe2203sp2.bclinux.x86_64	x86	✓	X	X
CTYunOS	3.0 (23.01)	5.10.0-136.12.0.86.ctl3.aarch64	aarch64	✓	✓	✓
		5.10.0-136.12.0.86.ctl3.x86_64	x86	✓	✓	✓
	25.07	6.6.0-0001	aarch64	✓	✓	✓
		6.6.0-0001	x86	✓	✓	✓
OpenEuler	24.03-SP3	6.6.0-132.0.0.111	aarch64	✓	X	X
		6.6.0-132.0.0.111	x86	✓	X	X
	22.03-SP1	5.10.0-136.12.0.86.oe2203sp1.x86_64	x86	✓	X	✓
	22.03-SP3	5.10.0-182.0.0.95.oe2203sp3.x86_64	x86	✓	X	✓
	22.03-SP4	5.10.0-216.0.0.115.oe2203sp4.aarch64	aarch64	✓	X	X
	22.03-SP4	5.10.0-216.0.0.115.oe2203sp4.x86_64	x86	✓	X	X
Tencent Linux	4.4	4.4-kernel 6.6.98-40.6.tl4.aarch64	aarch64	✓	X	X
		4.4-kernel 6.6.98-40.6.tl4.x86_64	x86	✓	X	X

OS	OS Version	Tested Kernel	Arch	doca-ofed/ doca-roce	doca-networking	doca-all
UOS	20.1070a	5.10.0-79.uelc20	aarch64	✓	X	X
		5.10.0-79.uelc20	x86	✓	X	X
	20.1070e	5.10.0-79.uelc20	aarch64	✓	X	X
		5.10.0-79.uelc20	x86	✓	X	X
VeLinux	2.1	5.15.152.ve.6u2-arm64	aarch64	✓	X	X
		5.15.152.ve.6u2-amd64	x86	✓	X	✓
kernel.org	7.1	7.1	aarch64	✓	X	X
		7.1	x86	✓	X	X

2.2.6.8 Storage Supported DOCA-Host Operating Systems

Operating System	Architecture	Default Kernel Version (Primary)/ Tested with Kernel Version (Community)	NFS-over-RDMA	NVMe-OF	GPUDirect Storage
Ubuntu 22.04	x86	5.15.0 generic	✓	✓	✓
Ubuntu 24.04	x86	6.8.0 -generic	✓	✓	✓
Ubuntu 25.10 (x86, aarch64)	x86	6.17.0-8	✓	✓	✓
RHEL/Rocky 8.x (x<=10)	x86	4.18.0-553.el8_10.x86_64	✓	✓	✓
RHEL/Rocky 9.x (x<=7)	x86	5.14.0-Kernel-611.5.1.el9_7	✓	✓	✓
RHEL/Rocky10.x (x<=1)	x86	6.12.0-124.8.1	✓	✓	✓
Debian 12.x (x<=13)	x86 / aarch64	6.1.0-42	✓	✓	X
Debian 13.x (x<=3)	x86 / aarch64	6.12.63	✓	✓	X
SLES 15 SP6/SP7	x86 / aarch65	6.4.0-150700.51-default	✓	✓	X
SLES 16 SP0	x86 / aarch66	6.12.0-160000.5	✓	✓	X
BCLinux 22.10	x86	5.10.0-153.24.0.100.6.oe2203sp2.bclinux.x86_64	✓	✓	X
CTYunOS 3.0 (23.01)	x86 / aarch67	5.10.0-136.12.0.86.ctl3.x86_64 / 5.10.0-136.12.0.86.ctl3.aarch64	✓	✓	X

Operating System	Architecture	Default Kernel Version (Primary)/ Tested with Kernel Version (Community)	NFS-over-RDMA	NVMe-OF	GPUDirect Storage
CTYunOS 25.07	x86 / aarch67	6.6.0-0001	✓	✓	X
OpenEuler 24.03-SP0	x86	6.6.0-28.0.0.34.oe2403.x86_64	✓	✓	X
OpenEuler 22.03-SP4	x86	5.10.0-216.0.0.115.oe2203sp4.x86_64	✓	✓	X
veLinux2.1	x86	5.15.152.ve.6u2-amd64	✓	✓	X

2.2.6.9 DOCA-Host Version Interoperability

The following versions were tested and verified in multi-version environments (environments with more than one **doca-ofed** version on host servers).

Current Version	Versions Verified for Interoperability	Release Type	Release Date
3.5.0 July 2026	DOCA-OFED 3.4.0	GA	April 2026
	DOCA-OFED 3.2.3	LTS-Update	July 2026
	DOCA-OFED 2.9.5 / MLNX_OFED 24.10-5	LTS-Update	June 2026
	DOCA-OFED 2.5.6 / MLNX_OFED 23.10-7	LTS-Update	May 2026

2.2.6.10 Supported Devices

For a complete list of supported devices, refer to the hardware user manuals.

2.2.7 Bug Fixes in This Version

2.2.7.1 DOCA Framework Bug Fixes

Ref #	Issue Details
4980835	Description: Attempting to modify the SRv6 Segments Left field using a direct Set action returned an unclear error instead of indicating that the action is unsupported.
	Keyword: SRv6; DOCA Flow; error reporting; unsupported operation
	Detected in version: 3.5.0
5149843 5165212 5165213	Description: OVS-DOCA may fail to attach a virtual-function representor to OVS, disrupting OVN-Kubernetes traffic.
	Keyword: OVS-DOCA; VF representor; traffic disruption
	Detected in version: 3.5.0
5230613	Description: On B300 NetPlugin systems, the bundled target maps may not represent supported multi-rail configurations, preventing a usable deployment plan.
	Keyword: DMS; B300; NetPlugin; target map

Ref #	Issue Details
	Detected in version: 3.5.0
5230252	Description: When no stage is specified for a DMS single-plane plan, the plan may default to configuration and leave the host partially configured without warning.
	Keyword: DMS; single-plane; deployment plan; partial configuration
	Detected in version: 3.5.0
5229962	Description: On GB200 systems, planning a single-plane deployment with the bundled target map may fail.
	Keyword: DMS; GB200; single-plane; target map
	Detected in version: 3.5.0
5227828	Description: On single-plane deployments, the DMS configuration stage may fail while bringing up network interfaces.
	Keyword: DMS; single-plane; configuration; network interfaces
	Detected in version: 3.5.0
5227092	Description: On single-plane deployments, DMS may leave network and RDMA interfaces with their default names instead of the expected rail-based names.
	Keyword: DMS; single-plane; interface naming; RDMA
	Detected in version: 3.5.0
5223152 5228590	Description: In NetPlugin deployments, DMS may apply an incomplete RoCE traffic-class configuration to physical interfaces.
	Keyword: DMS; NetPlugin; RoCE; traffic class
	Detected in version: 3.5.0
5221883	Description: In dual-plane NetPlugin deployments, DMS may apply inconsistent RDMA interface names, leaving some interfaces with their default names.
	Keyword: DMS; NetPlugin; RDMA; interface naming
	Detected in version: 3.5.0
5221880	Description: In dual-plane NetPlugin deployments, DMS may assign an RDMA name to the wrong device, causing the preparation stage to fail.
	Keyword: DMS; NetPlugin; RDMA; preparation
	Detected in version: 3.5.0
5221409	Description: DO-SPCX deployment tasks may fail when the DMS command-line tool location is not set in the environment.
	Keyword: DMS; DO-SPCX; command-line tool; deployment
	Detected in version: 3.5.0
5217068	Description: DMS verification may report MTU, virtual-function, and huge-page mismatches even when the host is configured correctly.
	Keyword: DMS; verification; false mismatch; host configuration
	Detected in version: 3.5.0
5217061	Description: On systems with multiple NUMA nodes, DMS may reduce an existing huge-page allocation or leave fewer huge pages than configured.
	Keyword: DMS; huge pages; NUMA; memory allocation

Ref #	Issue Details
	Detected in version: 3.5.0
5211680	Description: On GB300 dual-plane systems, DMS may apply network and congestion-control settings that do not match the deployment guide.
	Keyword: DMS; GB300; dual-plane; network configuration
	Detected in version: 3.5.0
5203065	Description: After a DMS installation fails, the status API may incorrectly report the plan as completed.
	Keyword: DMS; status API; installation; status reporting
	Detected in version: 3.5.0
5201742 5201640	Description: During DO-SPCX configuration, the DMS configuration stage may remain stuck at 13 of 14 phases.
	Keyword: DMS; DO-SPCX; configuration; stalled deployment
	Detected in version: 3.5.0
5201632	Description: During a DO-SPCX deployment, incorrect interface naming may cause the DMS preparation stage to fail after reboot.
	Keyword: DMS; DO-SPCX; preparation; interface naming
	Detected in version: 3.5.0
5199086	Description: In NetPlugin deployments, DMS configuration may fail while bringing up network interfaces.
	Keyword: DMS; NetPlugin; network interfaces; configuration
	Detected in version: 3.5.0
5197210 / 5174929	Description: The DMS Verify API may return error values instead of configuration results for a DO-SPCX preparation plan.
	Keyword: DMS; Verify API; DO-SPCX; validation
	Detected in version: 3.5.0
5191384 / 5191380	Description: DMS may reject attempts to configure congestion-control counter groups or enable congestion-control tracing through its command-line interface.
	Keyword: DMS; congestion control; counter groups; tracing
	Detected in version: 3.5.0
5185448	Description: DMS status may remain incomplete even after all configuration phases finish successfully.
	Keyword: DMS; status reporting; configuration completion
	Detected in version: 3.5.0
5185431	Description: On multi-plane systems using per-rail bonding, DMS may fail to bring up RDMA links and may assign incorrect RDMA interface names.
	Keyword: DMS; multi-plane; RDMA; link activation
	Detected in version: 3.5.0
5185406	Description: In multi-plane deployments, DMS may select the wrong network interface during link configuration, causing configuration to fail.
	Keyword: DMS; multi-plane; network interfaces; link configuration

Ref #	Issue Details
	Detected in version: 3.5.0
5180951	Description: During DO-SPCX configuration, DMS may report that programmable congestion control is not ready even when it is available.
	Keyword: DMS; DO-SPCX; congestion control; readiness check
	Detected in version: 3.5.0
5180863	Description: The DO-SPCX configuration stage may stall while initializing programmable congestion control.
	Keyword: DMS; DO-SPCX; congestion control; configuration
	Detected in version: 3.5.0
5179634	Description: In NetPlugin deployments, the DOCA SPCX congestion-control service may fail to start for RDMA devices.
	Keyword: DMS; NetPlugin; congestion control; RDMA
	Detected in version: 3.5.0
5179628	Description: In NetPlugin deployments with split ports, the DMS preparation stage may fail even after network settings have been applied.
	Keyword: DMS; NetPlugin; preparation; split ports
	Detected in version: 3.5.0
5179435	Description: In DOCA XPlane, log rotation may fail, allowing XPlane logs to accumulate.
	Keyword: DOCA XPlane; log rotation; log files
	Detected in version: 3.5.0
5179302	Description: When DOCA XPlane uses a nondefault container registry, DMS boot validation may reject the image and block the deployment.
	Keyword: DMS; DOCA XPlane; container registry; boot validation
	Detected in version: 3.5.0
5179253	Description: On four-rail, dual-plane deployments, DMS may create unused virtual functions and representors, leaving extra interfaces active but unconfigured.
	Keyword: DMS; multi-plane; virtual functions; network interfaces
	Detected in version: 3.5.0
5175085	Description: After a successful preparation stage, cleaning up the active DMS plan may prevent the configuration stage from starting.
	Keyword: DMS; preparation; configuration; plan cleanup
	Detected in version: 3.5.0
5175073	Description: During DO-SPCX configuration, DMS may fail to identify virtual-function representors, causing configuration to stop.
	Keyword: DMS; DO-SPCX; VF representors; configuration
	Detected in version: 3.5.0
5166426	Description: When an XPlane client requests initial state after a local link has already failed, the existing link-down event may be omitted.
	Keyword: DOCA XPlane; initial state; link failure; event reporting
	Detected in version: 3.5.0

Ref #	Issue Details
5165499	Description: DMS status may remain in progress indefinitely when a plan service fails.
	Keyword: DMS; status reporting; plan failure
	Detected in version: 3.5.0
5160589	Description: The DMS command-line tool may fail when querying the pending link type.
	Keyword: DMS; command-line tool; link type; query failure
	Detected in version: 3.5.0
5160542	Description: DMS may fail to configure ECN receive parameters on a network device.
	Keyword: DMS; ECN; congestion control; configuration
	Detected in version: 3.5.0
5158946	Description: DMS may return an empty response when querying breakout lane assignments for a module port.
	Keyword: DMS; port breakout; lane query
	Detected in version: 3.5.0
5158192	Description: DMS blueprint verification may report error values for supported settings, causing valid profiles to appear mismatched.
	Keyword: DMS; blueprint verification; configuration validation
	Detected in version: 3.5.0
5156646	Description: Installing the RA2.2 software multi-plane preparation profile on ConnectX-8 may fail during configuration verification.
	Keyword: DMS; DO-SPCX; profile installation; ConnectX-8
	Detected in version: 3.5.0
5154317	Description: DO-SPCX planning may fail because required planner data is missing from the installed package.
	Keyword: DMS; DO-SPCX; planning; packaging
	Detected in version: 3.5.0
5153995	Description: DMS may be unable to locate required DO-SPCX blueprint data, preventing deployment planning.
	Keyword: DMS; DO-SPCX; blueprints; planning
	Detected in version: 3.5.0
5151565	Description: The DOCA XPlane Kubernetes container may report version 0.0.0 in startup logs instead of the installed version.
	Keyword: DOCA XPlane; Kubernetes; version reporting; logs
	Detected in version: 3.4.0
5139760	Description: DMS may fail to apply link-aggregation resource settings during preparation, blocking DO-SPCX bring-up.
	Keyword: DMS; DO-SPCX; LAG; preparation
	Detected in version: 3.5.0
5139743	Description: DMS may generate an invalid port-breakout configuration during preparation, blocking DO-SPCX and XPlane bring-up.

Ref #	Issue Details
	Keyword: DMS; DO-SPCX; port breakout; XPlane
	Detected in version: 3.5.0
5131673	Description: DOCA DMS packages may fail to build for openEuler 22.03 and CTyunOS 23.01, blocking BFB image creation.
	Keyword: DOCA DMS; openEuler; CTyunOS; packaging
	Detected in version: 3.5.0
5128524	Description: On BlueField-2, DMS may fail to change interface MTU.
	Keyword: DMS; BlueField-2; MTU; configuration
	Detected in version: 3.5.0
5128496	Description: The DO-SPCX installer may complete without applying configuration values from the generated deployment plan.
	Keyword: DMS; DO-SPCX; installer; deployment plan
	Detected in version: 3.5.0
5107936	Description: OVS-DOCA offload may stall after an asynchronous flow-insertion failure, which can also prevent Kubernetes pods from being created.
	Keyword: OVS-DOCA; flow offload; deployment stall; Kubernetes
	Detected in version: 3.2.2
5098855	Description: The DOCA Flow Tune package may be unavailable for Anolis 8.6, blocking installation and BFB creation.
	Keyword: DOCA Flow Tune; Anolis 8.6; packaging; BFB
	Detected in version: 3.5.0
5089659	Description: On Debian BFB images, DOCA Bench may report supported device capabilities as unavailable and fail to create benchmark pipeline steps.
	Keyword: DOCA Bench; Debian; device capabilities; packaging
	Detected in version: 3.4.0
5082213	Description: DMS operating-system verification may fail when required network tooling is unavailable.
	Keyword: DMS; OS verification; network tools
	Detected in version: 3.5.0
5042755	Description: DOCA Flow Performance may fail to start in DOCA mode, reporting that no queues are available.
	Keyword: DOCA Flow Performance; queue allocation; application startup
	Detected in version: 3.5.0
5011919	Description: DOCA PSP Gateway may stop forwarding encrypted traffic received on a physical function managed by the embedded switch.
	Keyword: DOCA PSP Gateway; encryption; eSwitch; traffic forwarding
	Detected in version: 3.5.0
4991583	Description: OVS may crash during startup after a DPU reboot or driver restart when DOCA initialization is enabled manually but not in the system configuration.
	Keyword: Open vSwitch; service crash; DOCA initialization; configuration

Ref #	Issue Details
	Detected in version: 3.4.0
4989366	Description: DOCA Flow control-pipe entries may fail to match metadata tags, preventing the expected traffic rules from taking effect.
	Keyword: DOCA Flow; metadata; control pipe; traffic matching
	Detected in version: 3.5.0
4980744	Description: DOCA Argus may omit container-termination events, resulting in incomplete container lifecycle reporting.
	Keyword: DOCA Argus; container events; lifecycle reporting
	Detected in version: 3.4.0
4979016	Description: DOCA Bench throughput may drop significantly when DMA job sizes are not powers of two.
	Keyword: DOCA Bench; DMA; performance; job size
	Detected in version: 3.4.0
4958775	Description: NAT64 communication may fail across mixed-version deployments because its Unix-domain socket interface is not backward compatible.
	Keyword: NAT64; backward compatibility; UDS; mixed versions
	Detected in version: 3.3.0
4882015	Description: The DMS API may report a pending BlueField mode change as active before the required power cycle instead of returning the current mode.
	Keyword: DMS; BlueField mode; state reporting; API
	Detected in version: 3.3.0
4877227	Description: DOCA Flow may fail to create a flooding pipe containing a single entry.
	Keyword: DOCA Flow; flooding pipe; flow entry
	Detected in version: 3.3.0
4868003	Description: OVS-DOCA may lose flow context while processing hardware-offloaded traffic, resulting in packet-processing errors.
	Keyword: OVS-DOCA; flow context; hardware offload; packet processing
	Detected in version: 3.2.1
4784797	Description: Rapidly creating DOCA Flow pipes until resources are exhausted may trigger a device firmware reset instead of returning a resource-exhaustion error.
	Keyword: DOCA Flow; resource exhaustion; firmware reset; pipe allocation
	Detected in version: 3.0.0
4771333	Description: DOCA Bench may fail to run DOCA Comch workloads when the sweep option is used.
	Keyword: DOCA Bench; DOCA Comch; parameter sweep; benchmarking
	Detected in version: 3.2.0
4686141	Description: DOCA DPU-GPU remote offload may report errors while establishing communication between the server and orchestrator.
	Keyword: DPU-GPU remote offload; server; orchestrator; communication
	Detected in version: 3.2.0

2.2.7.2 DOCA-Host and DOCA Drivers Bug Fixes

Ref #	Issue Details
5218407 / 5220065 / NVbug 6583798	Description: Resolved a rare issue where mlx5_alloc_dbrec() in rdma-core could return an invalid non-NULL pointer instead of NULL when doorbell-record allocation failed due to an out-of-memory condition. As a result, the caller could incorrectly treat the allocation failure as a success.
	Keyword: RDMA
	Detected in version: 3.3.0
5106990 / 5126719 / NVbug 6494049	Description: Resolved an issue where an IPoIB interface could hang when a fabric change, multicast membership refresh, and administrative teardown occurred simultaneously. In this scenario, the interface could remain administratively down while still appearing operationally up inside the driver. As a result, the pending “mark link usable” operation did not complete, while the teardown flow waited for it to finish, causing network configuration on the host to freeze.
	Keyword: IPoIB interface
	Detected in version: 3.1.0
4870459 / 4409167 / 4722911 / 4837736 / 4877800 / 4930375 / 49889678 / NVbug 6494839	Description: Resolved an issue where configuring SFs on controller 1 could cause the mlxfwreset flow to time out and fail. The underlying queue pairs did not properly transition to a reset state, creating a command resource leak that blocked the reset flow.
	Keyword: mlxfwreset; SF; QPs
	Detected in version: 3.2.0
5005721 / 5069001 / 5219847 / 5220048 / 5220049 / 5220050 / NVbug 6552636	Description: Resolved a truncation issue for block sizes of 4GB or larger. This issue could occur when a DPDK program used an RDMA memory region above 4GB, where the driver might not map the memory correctly. As a result, the translation table could be filled with repeated addresses instead of advancing addresses.
	Keyword: RDMA memory, DPDK
	Detected in version: 3.4.0
4994944 / 4999889 / 4999890 / 5000477 / NVbug 6494527	Description: Resolved an issue that could occur when ovs-vswitchd queries netdev statistics over Netlink during NIC bring-up, potentially leading to a NULL pointer dereference. **Unable to handle kernel NULL ...**
	Keyword: ovs-vswitchd queries
	Detected in version: 3.4.0
5119365	Description: Resolved a potential kernel panic that could occur when switching eSwitch modes between legacy and switchdev while accessing mlx5 hairpin settings under /sys/class/net/<interface>/.
	Keyword: kernel panic, eSwitch
	Detected in version: 3.4.0
5001191 / 3573149 / 5004447 / NVbug 6494498	Description: Resolved an issue where, on an Arm VM, performing an FLR (PCI) reset while simultaneously sending commands through cmdif could cause the VM to crash.
	Keyword: Arm VM, FLR
	Detected in version: 3.4.0

Ref #	Issue Details
5105915 / 5191705 / NVbug 6548013	Description: Resolved an issue where missing coordination between eSwitch reconfiguration, including LAG creation and representor registration/loading, could lead to incorrect behavior.
	Keyword: eSwitch, LAG
	Detected in version: 3.4.0
5145461 / 5154015 / 5189266 / 5189267 / NVbug 6435606	Description: Resolved a race condition between uplink close and eswitch devcom pair/unpair events during the teardown of representor send-to-vport rules, which could result in unexpected behavior in some cases.
	Keyword: mlx5 driver
	Detected in version: 3.4.0
5106086 / NVbug 6494055	Description: Resolved an issue where a memory leak could occur if device registration with mlxdevm failed.
	Keyword: Memory leak, mlxdevm
	Detected in version: 3.4.0
5057168 / 5057748 / 5066051 / NVBug 6494265	Description: Fixed an issue where rdma_resolve_addr() returned an ENODEV error on an IPoIB interface when the destination was an IPv4 broadcast address.
	Keyword: RDMA
	Detected in version: 3.2.0
5158982	Description: Resolved an issue where DOCA-Host 3.4.0 kernel drivers could fail to build with the following RHEL errata kernels and later: • RHEL 9.8: kernel 5.14.0-687.17.1.el9_8 and later • RHEL 10.2: kernel 6.12.0-211.20.1.el10_2 and later Users running these or newer errata kernels should use the packages for their RHEL version from the DOCA-Host 3.4.1 repository, which adds support for the updated RHEL errata kernels and includes an updated kernel-driver source package that resolves the build failure.
	Keyword: RHEL errata kernel compatibility
	Detected in version: 3.4.0
5036525 / 5042525 / NVbug 6188726	Description: Resolved an issue on RPM-based systems with DKMS version 3.4.0 or later where DOCA-Host DKMS packages could fail to automatically run DKMS commands in post-install scripts because the `dkms` command path changed from `/usr/sbin/dkms` to `/usr/bin/dkms`. As a result, DOCA-Host kernel drivers, as well as `nvidia_peermem` from `nvidia-kernel-open-dkms`, could become unavailable.
	Keyword: DOCA-Host DKMS packages
	Detected in version: 3.4.0

2.2.7.3 BSP Bug Fixes

Ref #	Details
5125190	Description: On BlueField-3 systems running the DOCA 3.4 Ubuntu 24.04 BFB, `apt` update may fail, preventing packages from being updated or installed.

Ref #	Details
	Keywords: APT; Ubuntu; package management
	Detected in version: 4.15.0
5180884	Description: On RHEL 9 and RHEL 10, DOCA Installer may fail to start and report a missing library.
	Keywords: DOCA Installer; RHEL; missing library
	Detected in version: 4.15.0
5172294	Description: On JBOF systems, BFB installation may target an NVMe device other than the intended BlueField NVMe.
	Keywords: BFB installation; NVMe; JBOF
	Detected in version: 4.15.0
5172025	Description: After installing a RHEL 10.2 BFB, the BlueField-3 DPU may stop at the UEFI menu instead of booting.
	Keywords: BFB installation; RHEL; UEFI
	Detected in version: 4.15.0
4989952	Description: On certain BlueField-3 devices upgraded from firmware 4.7.1.4 to 4.13.0, the configured 5200 MT/s DDR speed may not take effect after the first reboot.
	Keywords: DDR; firmware upgrade
	Detected in version: 4.13.0.13932
4986356	Description: On BlueField-2 systems running DOCA 3.3 or 3.4, `dmidecode` may reverse the reported sizes of the L1 data and instruction caches.
	Keywords: dmidecode; cache information; BlueField-2
	Detected in version: DOCA 3.4.0, DOCA 3.3.0
4966842	Description: A latent Day 1 bug in the UEFI RTC driver causes the month to be calculated incorrectly (off by one). This results in unexpected date jumps and inaccurate system time reporting.
	Keywords: real-time clock; system date
	Detected in version: 4.15.0
4867794	Description: On RHCOS 4.20, `/etc/mlnx-release` may show a release name and date that do not match the installed BFB.
	Keywords: RHCOS; release metadata; BFB
	Detected in version: 4.15.0
4742314 4546931 4822210	Description: Running `mlxfwreset` on BlueField-3 with Ubuntu 24.04 may trigger an ext4 journal warning call trace (`ext4_journal_check_start+0xc8/0x120`); no functional impact observed, safe to ignore.
	Keywords: mlxfwreset; ext4; kernel warning
	Detected in version: 4.15.0
4234285	Description: During BlueField boot, the BlueField BIOS/UEFI may fail to acquire a DHCP IP address from the BMC for Redfish communication. This would cause the BIOS to bypass the Redfish step and to ignore any configuration changes in Redfish.
	Keywords: DHCP; UEFI; Redfish

Ref #	Details
	Detected in version: 4.15.0

2.2.7.4 BMC Bug Fixes

Ref #	Issue Details
5149126	Description: During a golden-image BMC firmware update, activation may fail and leave the Redfish task stuck at 0%, preventing subsequent updates.
	Reported in version: 26.04
5107116	Description: In rare cases, the PLDM firmware update process in NIC mode may fail during the activation phase due to a transient communication error on the device.
	Reported in version: 26.04
4966486	Description: On BlueField-3 systems, requests to `/redfish/v1/Managers/Bluefield_BMC` may take several minutes and return an internal error or timeout. Other Redfish resources may remain responsive, and the UEFI System Configuration page may also load slowly.
	Reported in version: 26.04
5159819	Description: After a watchdog-triggered DPU reset, the reset reason may be missing from the BMC system event log.
	Reported in version: 26.04
5071652	Description: In rare cases, the CEC/ERoT firmware version reported through Redfish may appear incomplete or contain blank characters.
	Reported in version: 26.04
5071052	Description: During a DPU OS reboot, the BMC system event log may incorrectly report a P1 link configuration error even when the link is healthy.
	Reported in version: 26.04
5044522	Description: A timing side channel in HMAC-SHA256-128 integrity verification may allow an adjacent-network attacker to infer authentication data.
	Reported in version: 26.04
4833493	Description: When the BMC event log is full, rebooting the BMC may cause existing events to be lost.
	Reported in version: 26.04

2.2.7.5 BlueField-3 Firmware Bug Fixes

Internal Ref.	Issue
5085054 / 5103944	**Description:** Resolved an issue where resetting the ID list register released only the domain counters.
	Keywords: Counters
	Detected in version: 32.49.1014

Internal Ref.	Issue
	Fixed in Release: 32.50.1002
5199016	**Description:** Resolved an issue where running PRDMA traffic could cause a call trace due to an iRISC halt caused by miss-access handling during retransmission parsing.
	Keywords: PRDMA
	Detected in version: 32.49.1014
	Fixed in Release: 32.50.1002
5083504 / NVbug 6078783	**Description:** Resolved an issue where firmware internal errors could occur due to FW_INTERNAL_USAGE partition data corruption. The firmware now recovers by erasing the corrupted partition.
	Keywords: Firmware internal errors
	Detected in version: 32.49.1014
	Fixed in Release: 32.50.1002
5174504 / 5140787	**Description:** Resolved an issue where intensive parallel QP INIT2RTR / DESTROY operations could incur high operation latency and reduce connection establishment rates when PCC was enabled.
	Keywords: PCC, INIT2RTR, DOCA PCC
	Detected in version: 32.49.1014
	Fixed in Release: 32.50.1002
4947006 / 4980956 / 4982511	**Description:** Resolved two rare race conditions in the PCC steering extension when the feature is enabled through path migration, empty RTT, or a custom header. Under rare timing conditions during QP creation or destruction, these issues could cause traffic disruption or resource leaks.
	Keywords: PCC steering extension
	Detected in version: 32.49.1014
	Fixed in Release: 32.50.1002
5166648	**Description:** Resolved an issue involving an uninitialized stack read and an inverted EtherType check in the RX parser.
	Keywords: RX parser
	Detected in version: 32.47.1026
	Fixed in Release: 32.50.1002
4937640	**Description:** Resolved an issue where a PPCC mlxreg read operation could succeed when executed on a PF that is not the port owner, which could have security implications.
	Keywords: PCC, PPCC

Internal Ref.	Issue
	Discovered in Version: 32.49.1014
	Fixed in Release: 32.50.1002
4923806 / 4867111 / 4927875	**Description:** Resolved an issue that could cause RoCE communication failures between specific NIC pairs after repeated queue pair (QP) create/destroy cycles. In the affected cases, traffic could enter a bad state and fail with transport retry counter exceeded errors. This fix prevents the NIC from entering that condition.
	Keywords: RDMA; connection failure
	Discovered in Version: 32.46.1006
	Fixed in Release: 32.50.1002
5093695	**Description:** Resolved an issue where using mlxconfig to configure all modules with the 0xff module split value could reset the modules to the default configuration.
	Keywords: mlxconfig
	Discovered in Version: 32.48.1000
	Fixed in Release: 32.50.1002
5090639	**Description:** Resolved an issue in single-domain mode where an FLR on a GVMI did not clear the per-domain usage bits in `SP_core_diag_multi_domain->pcie_link_latency_max_read` and `SP_core_diag_multi_domain->pcie_link_latency_min_read`. As a result, when the next GVMI took ownership and attempted to configure these counters, it could receive `DIAG_DATA_LIST_WRITE_STATUS_NO_RESOURCES`.
	Keywords: PCIe Link Latency Arch counters
	Discovered in Version: 32.49.1014
	Fixed in Release: 32.50.1002
5077326 / 4902171	**Description:** Resulved an issue where a slow module response to the NCSI GetModuleSerialData command could incorrectly trigger a dead IRISC watchdog assert. The firmware now handles this waiting state correctly, so slow module responses no longer cause a false watchdog hang indication.
	Keywords: NCSI
	Detected in version: 32.43.1014
	Fixed in Release: 32.50.1002
4987961	**Description:** Resolved an issue where older versions failed to configure an odd `num_msix` value for a VF. After upgrading to the new version, odd `num_msix` values can now be configured for virtio VFs.
	Keywords: `num_msix`, virtio VF

Internal Ref.	Issue
	Discovered in Version: 32.48.1000
	Fixed in Release: 32.50.1002
4925548 / 4901065	**Description:** Resolved an issue where HIDE_PORT2_PF could be configured in non-DPU mode. Since HIDE_PORT2_PF is only relevant in DPU mode, firmware now reverts it to the valid/default configuration when set in non-DPU mode.
	Keywords: DPU NIC mode
	Discovered in Version: 32.49.1014
	Fixed in Release: 32.50.1002
4918436	**Description:** Added VSEC FLR support for ECPF and other previously skipped functions.
	Keywords: VSEC FLR support for ECPF
	Discovered in Version: 32.48.1000
	Fixed in Release: 32.50.1002
4976266	**Description:** Resolved an issue with legacy queue affinity where traffic only used the bandwidth of one physical port at all times.
	Keywords: Queue affinity
	Discovered in Version: 32.48.1000
	Fixed in Release: 32.50.1002
4763579 / 4877802	**Description:** Resolved an issue on BlueField-3 Socket Direct configurations where an mlxfwreset PCI reset could time out during MGIR polling because the SD links reset outside the firmware sync window. The firmware PCI link-down/sync window was enlarged to prevent the timeout.
	Keywords: Socket Direct, mlxfwreset PCI reset
	Discovered in Version: 32.47.1026
	Fixed in Release: 32.50.1002

2.2.7.6 BlueField-2 Firmware Bug Fixes

N/A

2.2.8 Known Issues

2.2.8.1 DOCA Framework Known Issues

Reference	Issue Details
5146023	Description: When OVS hardware offload and sFlow are enabled together, sampled TCP traffic may be briefly forwarded out of order.
	Workaround: Disable hardware offload or sFlow.
	Keyword: OVS-DOCA; sFlow; TCP packet ordering
	Reported in version: 3.5.0
4915314	Description: On Grace systems with ConnectX-7 or ConnectX-8, OVS-DOCA maximum packet rate and bandwidth may be up to approximately 20% lower than on Intel systems under high-cache-miss workloads.
	Workaround: N/A
	Keyword: OVS-DOCA; performance; Grace; ConnectX-7/8
	Reported in version: 3.5.0
5000400	Description: Creating and destroying FlexIO SDK resources from different system threads might result in an error.
	Workaround: Implement thread safety in application level.
	Keyword: FlexIO; DPA
	Reported in version: 3.4.0
4889386	Description: **PRDMA_MP** application does not start in switch/FDB mode if the **plan-iface-map** argument is not provided, even when running with a single plane configuration.
	Workaround: N/A
	Keyword: Non-multiplane; FDB initialization
	Reported in version: 3.4.0
4874324	Description: When running the default DOCA Device Management Service (dmsd) via systemctl on openEuler DPUs, client get and set commands fail with a PAM authentication permission denied error. This prevents authorized users, including root, from querying or configuring device parameters through the standard background service.
	Workaround: Stop the default systemd service (**systemctl stop dmsd**) and manually launch the **dmsd_binary** with the explicitly defined authentication and user flags (e.g., **-auth pam -tls_enabled=false -allowed_users root**). Once the manual daemon is running, the **dmsc** client commands will successfully authenticate and execute.
	Keyword: DOCA DMS; openEuler; PAM
	Reported in version: 3.4.0
4871396 4849953	Description: The **dpa-ps** and **dpa-statistics** diagnostic tools are missing from the BlueField-3 BFB image for Oracle Linux 9 and Oracle Linux 8.
	Workaround: N/A
	Keyword: DPA; missing package

Reference	Issue Details
	Reported in version: 3.3.0
4833932 4833931	Description: The DMS container for x86 hosts is missing the **awk** utility dependency.
	Workaround: N/A
	Keyword: Container dependency; OS verify failure
	Reported in version: 3.3.0
4659016	Description: The **doca_spcx_cc** application does not support running multiple concurrent instances on the same network device.
	Workaround: N/A
	Keyword: DOCA PCC; FlexIO; QP Transition (RTR)
	Reported in version: 3.3.0
4761527	Description: The DOCA Flow ACL pipe (PIPE_ACL) does not currently support IPv6 matching.
	Workaround: N/A
	Keyword: IPv6 support; hardware steering (HWS); matcher failure
	Reported in version: 3.3.0
4085538	Description: The **doca-bench** tool fails to initialize GPU acceleration, returning a **DOCA_ERROR_NOT_SUPPORTED** error.
	Workaround: N/A
	Keyword: doca-bench; GPUNetIO; **libdoca_bench_cuda_impl**
	Reported in version: 3.3.0
4774876	Description: The **dpa-resource-mgmt** tool fails to run on systems defaulting to Python 3.12, displaying the error **ModuleNotFoundError: No module named 'pkg_resources'**.
	Workaround: Execute the tool using Python 3.10 or older using one of the following methods: • Run explicitly with python3: **python3 /usr/bin/dpa-resource-mgmt <arguments>** • Modify the script header to point to the generic python3 executable: `sudo sed -i '1s|#!/usr/bin/python3.12|#!/usr/bin/python3|' /usr/bin/dpa-resource-mgmt`
	Keyword: FlexIO; Single Point of Resource Distribution; missing module
	Reported in version: 3.3.0
4401408	Description: When monitoring multiple PTP interfaces, the **ptp_ports** field in PTP Monitor logs incorrectly displays the same clock id for all interfaces. This clock ID corresponds to the first interface due to a bug in the ptp4l module. The port number, however, is correct.
	Workaround: N/A
	Keyword: DOCA Firefly; Logging; PTP
	Reported in version: 3.1.0
4549785	Description: DPA samples are not supported when running in switchdev mode from the host.
	Workaround: N/A

Reference	Issue Details
	Keyword: DOCA Core; switchdev
	Reported in version: 3.1.0
4540734	Description: OVS-DOCA live update is not supported over AML2.
	Workaround: N/A
	Keyword: OVS
	Reported in version: 3.1.0
4273101	Description: High PPS traffic on PF ports (>15Mpps) causes CPS degradation in OVS.
	Workaround: Apply a software meter to limit PPS on the PF port using `ovs-vsctl set interface <pf> options:sw-meter=pps:200000:32` then restart `openvswitch` for the setting to take effect.
	Keyword: Open vSwitch
	Reported in version: 3.1.0
4487794	Description: Upgrading to DOCA FW-Bundle 3.0.0 or later on a system with DOCA BF-Bundle (BFB) 2.10.0 or earlier is not supported. Full BF-Bundle (BFB) upgrade is required.
	Workaround: N/A
	Keyword: Software update
	Reported in version: 3.1.0
4415267	Description: When DPA code running by Flex IO RPC or CMD-Q runs into a recoverable error, code execution does not resume from calling context, which results in a timeout.
	Workaround: N/A
	Keyword: DPA; Timeout; Flex IO SDK; RPC
	Reported in version: 3.0.0
4297489	Description: Due to incompatibility between DPA and host libraries, a DPA device application must be recompiled after updating DOCA to a newer version.
	Workaround: N/A
	Keyword: DPA; host library; update
	Reported in version: 2.10.0
4270602	Description: UEFI/ATF firmware does not upgrade as part of the Linux Standard Tool process when Secure Boot is disabled.
	Workaround: Remove PK key and initiate UEFI/ATF firmware upgrade again. To remove the PK key, use the UEFI menu to navigate to Device Manager → Secure Boot Configuration → Custom Secure Boot Options → PK Options → Delete Signature.
	Keyword: UEFI/ATF; PK; Secure Boot; EFI Capsule Authentication
	Reported in version: 2.10.0
4200690	Description: The fTPM trusted application is signed for testing proposes only (i.e., not securely) with a development key.
	Workaround: N/A
	Keyword: fTPM over OP-TEE
	Reported in version: 2.10.0

Reference	Issue Details
N/A	Description: Applications using DPA might not work with older firmware versions.
	Workaround: Full upgrade of all DOCA 2.9.0 components including the firmware (i.e., doca-host and BF-Bundle).
	Keyword: DPA; backward compatibility
	Reported in version: 2.9.0
N/A	Description: Applications using FlexIO SDK API may have missing symbols during runtime.
	Workaround: Re-compile FlexIO-based applications with the DOCA 2.9.0 release.
	Keyword: FlexIO; backward compatibility
	Reported in version: 2.9.0
4095728	Description: Corrupt create repo causes doca-kernel repo to not contain the repo data.
	Workaround: If repo data is missing after installing the doca-kernel repo, run **createrepo --help**. If no output is generated, then the createrepo is corrupted and must be removed and reinstalled.
	Keyword: Kernel; repo
	Reported in version: 2.9.0
4049034	Description: On openEuler 22.03 SP3 and openEuler 20.03 SP1, it is not possible to do yum update after BFB installation.

	Workaround: To perform yum update with either openEuler 22.03 SP3 and openEuler 20.03 SP1, follow these procedures depending on the use case:
	1. To update DOCA only, disable ALL repos:
	`dnf --disablerepo='*' --enablerepo='kubernetes,doca' -y update`
	2. To update OS components, exclude **rdma-core** using one of the following options:
	• Option 1 – disable **rdma-core** from **dnf.conf** using **excludepkgs=rdma-core*oe2203sp3***
	• Option 2 – disable **rdma-core** from **openEuler.repo** file under OS and everything using **exlude=rdma-core***
	• Option 3 – run
	`yum update -x rdma-core`

	Keyword: openEuler
	Reported in version: 2.9.0
4046180	Description: PCIe data IDs that require **Node**, **PCIe index**, and **Depth** parameters in **doca_telemetry_diag**, the only valid values are 0, 0, 0.
	Workaround: N/A
	Keyword: DOCA Telemetry
	Reported in version: 2.9.0
4129715	Description: Compiling Rocky 9.2 may fail when using GCC with the "native" arch flag.
	Keyword: Upgrade to toolset 13 (gcc 13).
	Keyword: Linux; GCC

Reference	Issue Details
	Reported in version: 2.9.0
4035553	Description: **oper_sample_period** does not always reflect the correct sample period. In some cases, it will reflect the **admin_sample_period** instead.
	Workaround: N/A
	Keyword: Core
	Reported in version: 2.8.0
4022563	Description: OVS-DOCA connection tracking with E2E enabled is not supported.
	Workaround: N/A
	Keyword: OVS-DPDK; connection tracking; E2E
	Reported in version: 2.8.0
3844705	Description: In OpenEuler 20.03, the Linux Kernel version 4.19.90 is affected by an issue that impacts the discard/trim functionality for the BlueField eMMC device which may cause degraded performance of the BlueField eMMC over time.
	Workaround: Upgrade to Linux Kernel version 5.10 or later.
	Keyword: eMMC discard; trim functionality
	Reported in version: 2.7.0

3877725	Description: During BFB installation in NIC mode on BlueField-3, too much information is added into RShim log which fills it, causing the Linux installation progress log to not appear in the RShim log.
	`echo "DISPLAY_LEVEL 2" > /dev/rshim0/misc`
	`cat /dev/rshim0/misc`

	Workaround: Monitor the BlueField-3 Arm's UART console to check whether BFB installation has completed or not for NIC mode.
	`[13:58:39] INFO: Installation finished`
	`...`
	`[14:01:53] INFO: Rebooting...`

	Keyword: NIC mode; BFB install
	Reported in version: 2.7.0
3855702	Description: Trying to jump from a steering level in the hardware to a lower level using software steering is not supported on **rdma-core** lower than 48.x.
	Workaround: N/A
	Keyword: RDMA; SWS
	Reported in version: 2.7.0

Reference	Issue Details

3855485	Description: When enabling the **PCI_SWITCH_EMULATION_ENABLE** NVconfig, the mlx devices, and potentially the RShim devices disappear. Also, looking at the kernel logs using **dmesg** shows the following messages:
	`pci 0000:29:00.0: BAR 0: no space for [mem size 0x0200 0000 64bit pref]`
	`pci 0000:29:00.0: BAR 2: no space for [mem size 0x0080 0000 64bit pref]`
	`...`

	Workaround: N/A
	Keyword: NVconfig; RShim; dmsg
	Reported in version: 2.7.0
3831230	Description: In OpenEuler 20.03, the Linux Kernel version 4.19.90 is affected by an issue that impacts the discard/trim functionality for BlueField eMMC device which may cause degraded performance of BlueField eMMC over time.
	Workaround: Upgrade to Linux Kernel version 5.10 or later.
	Keyword: eMMC discard; trim functionality
	Reported in version: 2.7.0
3743879	Description: **mlxfwreset** could timeout on servers where the RShim driver is running and INTx is not supported. The following error message is printed: **BF reset flow encountered a failure due to a reset state error of negotiation timeout** .

	Workaround: Set **PCIE_HAS_VFIO=0** and **PCIE_HAS_UIO=0** in **/etc/rshim.conf** and restart the RShim driver. Then re-run the **mlxfwreset** command.
	If host Linux kernel lockdown is enabled, then manually unbind the RShim driver before **mlxfwreset** and bind it back after **mlxfwreset** :
	`echo "DROP_MODE 1" > /dev/rshim0/misc`
	`mlxfwreset <arguments>`
	`echo "DROP_MODE 0" > /dev/rshim0/misc`

	Keyword: Timeout; mlxfwreset; INTx
	Reported in version: 2.7.0
3678069	Description: If using BlueField with NVMe and mmcbld and configured to boot from mmcblk, users must create **bf.cfg** file with **device=/dev/mmcblk0** , then install the ***.bfb** as normal.
	Workaround: N/A
	Keyword: NVMe
	Reported in version: 2.5.0
N/A	Description: Execution unit partitions are still not implemented and would be added in a future release.
	Workaround: N/A
	Keyword: EU tool
	Reported in version: 2.5.0

Reference	Issue Details
3666160	Description: Installing BFB using **bfb-install** when **mlxconfig** **PF_TOTAL_SF** >1700, triggers server reboot immediately.
	Workaround: Change **PF_TOTAL_SF** to 0, perform a graceful shutdown, power cycle, then installing BFB.
	Keyword: SF; **PF_TOTAL_SF** ; BFB installation
	Reported in version: 2.2.1
3594836	Description: When enabling Flex IO SDK tracer at high rates, a slow-down in processing may occur and/or some traces may be lost.
	Workaround: Keep tracing limited to ~1M traces per second to avoid a significant processing slow-down. Use tracer for debug purposes and consider disabling it by default.
	Keyword: Tracer FlexIO
	Reported in version: 2.2.1
3592080	Description: When using UEK8 on the host in DPU mode, creating a VF on the host consumes about 100MB memory on BlueField
	Workaround: N/A
	Keyword: UEK; VF
	Reported in version: 2.2.1
3546202	Description: After rebooting a BlueField-3 DPU running Rocky Linux 8.6 BFB, the kernel log shows the following error: `[ 3.787135] mlxbf_gige MLNXBF17:00: Error getting PHY irq. Use polling instead` This message indicates that the Ethernet driver will function normally in all aspects, except that PHY polling is enabled.
	Workaround: N/A
	Keyword: Linux; PHY; kernel
	Reported in version: 2.2.0
3566042	Description: Virtio hotplug is not supported in GPU-HOST mode on the NVIDIA Converged Accelerator.
	Workaround: N/A
	Keyword: Virtio; Converged Accelerator
	Reported in version: 2.2.0
3546474	Description: PXE boot over ConnectX interface might not work due to an invalid MAC address in the UEFI boot entry.
	Workaround: On BlueField, create **/etc/bf.cfg** file with the relevant PXE boot entries, then run the command **bfcfg** .
	Keyword: PXE; boot; MAC
	Reported in version: 2.2.0
3561723	Description: Running **mlxfwreset sync 1** on NVIDIA Converged Accelerators may be reported as supported although it is not. Executing the reset will fail.

Reference	Issue Details
	Workaround: N/A
	Keywords: mlxfwreset
	Reported in version: 2.2.0
3306489	Description: When performing longevity tests (e.g., mlxfwreset, DPU reboot, burning of new BFBs), a host running an Intel CPU may observer errors related to "CPU 0: Machine Check Exception".
	Workaround: Add **intel_idle.max_cstate=1** entry to the kernel command line.
	Keywords: Longevity; mlxfwreset; DPU reboot
	Reported in version: 2.2.0
3534219	Description: On BlueField-3 devices, from DOCA 2.2.0 to 32.37.1306 (or lower), the host crashes when executing partial Arm reset (e.g., Arm reboot; BFB push; mlxfwreset).
	Workaround: Before downgrading the firmware: 1. Run: `echo 0 > /sys/bus/platform/drivers/mlxbf-bootctl/large_icm` 2. Reboot Arm.
	Keyword: BlueField-3; downgrade
	Reported in version: 2.2.0

3462630	When trying to perform a PXE installation when UEFI Secure Boot is enabled, the following error messages may be observed:
	`error: shim_lock protocol not found.`
	`error: you need to load the kernel first.`

	Workaround: Download a Grub EFI binary from the Ubuntu website. For further information on Ubuntu UEFI Secure Boot PXE Boot, please visit Ubuntu's official website.
	Keyword: PXE; UEFI Secure Boot
	Reported in version: 2.0.2
3448841	Description: While running CentOS 8.2, switchdev Ethernet BlueField runs in "shared" RDMA net namespace mode instead of "exclusive".
	Workaround: Use **ib_core** module parameter **netns_mode=0**. For example: `echo "options ib_core netns_mode=0" >> /etc/modprobe.d/mlnx-bf.conf`
	Keyword: RDMA; isolation; Net NS
	Reported in version: 2.0.2
2706803	Description: When an NVMe controller, SoC management controller, and DMA controller are configured, the maximum number of VFs is limited to 124.
	Workaround: N/A
	Keyword: VF; limitation
	Reported in version: 2.0.2

Reference	Issue Details
3273435	Description: Changing the mode of operation between NIC and DPU modes results in different capabilities for the host driver which might cause unexpected behavior.
	Workaround: Reload the host driver or reboot the host.
	Keyword: Modes of operation; driver
	Reported in version: 2.0.2

3264749	Description: In Rocky and CentOS 8.2 inbox-kernel BFBs, RegEx requires the following extra huge page configuration for it to function properly:
	`sudo hugeadm --pool-pages-min DEFAULT:2048M`
	`sudo systemctl start mlx-regex.service`
	`systemctl status mlx-regex.service`
	If these commands have executed successfully you should see **active (running)** in the last line of the output.

	Workaround: N/A
	Keyword: RegEx; hugepages
	Reported in version: 1.5.1
3240153	Description: DOCA kernel support only works on a non-default kernel.
	Workaround: N/A
	Keyword: Kernel
	Reported in version: 1.5.0
3217627	Description: The **doca_devinfo_rep_list_create** API returns success on the host instead of **Operation not supported**.
	Workaround: N/A
	Keyword: DOCA core; InfiniBand
	Reported in version: 1.5.0

2.2.8.2 DOCA-Host and DOCA Drivers Known Issues

Reference	Issue Details
5225803 / NVbug 6627053	Description: On systems with a high CPU core count and limited RAM, for example, 32 GB RAM and 256 or more cores, installing mlnx-ofed-kernel-dkms or mlnx-ofa_kernel-dkms may fail with an out-of-memory (OOM) error or cause severe system thrashing.

	Workaround:
	1. Limit the number of parallel DKMS build jobs before installing DOCA-Host.
	`mkdir -p /etc/dkms/framework.conf.d`
	`echo 'parallel_jobs=8' > /etc/dkms/framework.conf.d/00-limit-jobs.conf`
	2. Adjust the value according to the system's available memory and CPU resources.

Reference	Issue Details
	Keyword: CPU, RAM, mlnx-ofa_kernel-dkms, out-of-memory (OOM)
	Reported in version: 3.5.0
5211043	Description: DOCA installation may fail due to MFT repository priority. On Ubuntu/Debian systems where both CUDA and DOCA repositories are enabled, APT may select MFT from the higher-priority CUDA repository instead of the DOCA repository. If DOCA requires a different MFT version, the installation may fail due to an unmet dependency.
	Workaround: Configure APT pinning to prefer the DOCA repository for MFT, or explicitly install the required MFT version from the DOCA repository.
	Keyword: DOCA installation
	Reported in version: 3.5.0
4994138 / 4957392	Description: When running `dkms status` after an upgrade, the error “Missing the module source directory or the symbolic link pointing to it. Manual intervention is required” is benign. It does not affect functionality and can be safely ignored.
	Workaround: N/A
	Keyword: Upgrade
	Reported in version: 3.4.0
4943253	Description: A hw csum failure error may appear in dmesg when a packet is encapsulated and decapsulated multiple times in an OVS-DOCA configuration with connection tracking. This has no functional impact.
	Workaround: N/A
	Keyword: hw csum failure error
	Reported in version: 3.4.0
5001191 / 3573149 / 5004447	Description: On an Arm VM, performing an FLR (PCI) reset while simultaneously sending commands through cmdif could cause the VM to crash.
	Workaround: N/A
	Keyword: Arm VM, FLR
	Reported in version: 3.4.0

Reference	Issue Details

5012067 / 5019988 / 55019991 / 5019994 / 5020421 / 5020422 / 5020423	Description: When configuring TC CT rules for a set of VFs assigned to VM ports, a firmware timeout could cause the driver to crash while deleting flow rules.
	Example call trace:
	`? page_fault+0x1e/0x30`
	`? mlx5_del_flow_rules+0x12/0x140 [mlx5_core]`
	`mlx5e_tc_action_miss_mapping_put+0x49/0x50 [mlx5_core]`
	`mlx5_tc_ct_delete_flow+0x4d/0x70 [mlx5_core]`
	`mlx5_free_flow_attr_actions+0xd2/0x160 [mlx5_core]`
	`mlx5e_tc_del_fdb_flow+0x15d/0x210 [mlx5_core]`
	`mlx5e_flow_put+0x23/0x40 [mlx5_core]`
	`__mlx5e_add_fdb_flow+0xf3/0x430 [mlx5_core]`
	`mlx5e_tc_add_flow+0x2ab/0x9c0 [mlx5_core]`
	`mlx5e_configure_flower+0x2f4/0x620 [mlx5_core]`
	`tc_setup_cb_add+0xca/0x1e0`
	`fl_hw_replace_filter+0x143/0x1e0 [cls_flower]`
	`[...]`

	Workaround: N/A
	Keyword: TC CT rules, VF, VM
	Reported in version: 3.4.0
5031064	Description: When upgrading from version 3.1.x or earlier on UOS1060, VElinux (aarch64), or BClinux22, the DOCA-OFED upgrade may fail because of an OpenMPI dependency conflict. The following is an example error that may appear in this scenario: Problem: the installed openmpi requires 'libhcoll.so.1()(64bit)'
	Workaround: Before starting the upgrade, remove the existing OpenMPI package using **`yum remove openmpi`** or **`apt remove openmpi`**.
	Keyword: DOCA-OFED, OpenMPI, upgrade, UOS1060, VElinux (aarch64), BClinux22
	Reported in version: 3.4.0
4880346	Description: On RHEL10 systems, after installing DOCA-Host, the **`mlx5_core`** and **`mlx5_ib`** drivers typically load correctly (networking and InfiniBand), but **`mlx5_fwctl`** may fail to load, resulting in missing fwctl functionality (e.g., firmware loading via the newer fwctl interface). This occurs because the fwctl kernel module is not installed when the running kernel's kernel-modules package is missing. The kernel-modules package must be installed before installing **`mlnx-ofa_kernel-dkms`**. If it is installed afterward, refer to the workaround instructions.

	Workaround:
	Install the package that provides the **`fwctl`** kernel module for the running kernel before installing **`mlnx-ofa_kernel-dkms`**.
	If you already installed **`kernel-modules`** afterward, DKMS may have left the kernel module state inconsistent. To correct this (where **`$KERNEL_VERSION`** is the target kernel version—needed especially if multiple kernels are installed), run:
	`dkms uninstall mlnx-ofa_kernel/26.01 $KERNEL_VERSION`
	`dnf install kernel-modules-$KERNEL_VERSION`
	`dkms install mlnx-ofa_kernel/26.01 $KERNEL_VERSION`

Reference	Issue Details
	Keyword: mlx5_fwctl, RHEL10, DOCA-Host installation
	Reported in version: 3.4.0
4639103	Description: OVS userspace modes DPDK and DOCA are not supported on RHEL 10.0, only kernel mode is supported.
	Workaround: N/A
	Keyword: OVS userspace modes, RHEL 10.0
	Reported in version: 3.2.0
4656429	Description: DOCA-Host installation caused CIFS to break in RHEL 8 and above. A dummy module was added so that CIFS will be disabled after DOCA-Host installation in RHEL 8 and above.
	Workaround: N/A
	Keyword: Installation, CIFS, RHEL 8 and above
	Reported in version: 3.0.0
4662007	Description: The same GCC version used to build the kernel must be used during module compilation to ensure compatibility.
	Workaround: N/A
	Keyword: GCC version
	Reported in version: 3.2.0
4694905	Description: The system requires DKMS version 3.2 or higher to be installed as a package. Please make sure that DKMS is properly installed and available on the system.
	Workaround: N/A
	Keyword: DKMS
	Reported in version: 3.2.0
4662554	Description: DOCA-OFED compilation fails when CONFIG_XFRM_OFFLOAD is disabled in the kernel.
	Workaround: N/A
	Keyword: CONFIG_XFRM_OFFLOAD
	Reported in version: 3.1.0
4574425	Description: After shutting down a Xen VM with a PCI device attached, rebinding the device to mlx5_core may fail with the following error and can result in a leak of a command resourc.: **mlx5_core 0000:05:00.2: wait_func:1205:(pid 11034): ENABLE_HCA(0x104) timeout.**
	Workaround: Re-enable SR-IOV after the Xen VM shutdown issue, run the following commands: **# echo 0000:05:00.2 > /sys/bus/pci/drivers/pciback/unbind** **# echo 0 > /sys/class/infiniband/mlx5_0/device/sriov_numvfs** **# echo 2 > /sys/class/infiniband/mlx5_0/device/sriov_numvfs** **# echo 0000:05:00.2 > /sys/bus/pci/drivers/pciback/unbind** **# echo 0000:05:00.2 > /sys/bus/pci/drivers/mlx5_core/bind**
	Keyword: Xen VM, PCI device
	Reported in version: 3.2.0

Reference	Issue Details
4641515	Description: Starting with the October 2025 firmware release (32.47.1xxx), backward compatibility with older MFT versions (4.31.0-149 and 4.30.0-139) is no longer supported. Using these older versions may cause unexpected behavior or prevent device detection.
	Workaround: N/A
	Keyword: Firmware; MFT; compatibility
	Reported in version: 3.2.0
4542517	Description: **systemd-networkd** continuously monitors all network interfaces, including unmanaged ones. On systems with thousands of virtual interfaces (e.g., over 2,000), this behavior causes high CPU usage, slow network command responses, and timeouts. Since there is no option to disable statistics collection for unmanaged interfaces, the issue cannot be mitigated without risking loss of connectivity.
	Workaround: N/A
	Keyword: systemd-networkd
	Reported in version: 3.2.0
4545488	Description: When upgrading from DOCA with RShim version 2.1.13 and below, a missing support table prevents firmware version extraction from the image bundle, resulting in warning messages. Installation proceeds, but firmware versions are not verified post-installation.
	Workaround: N/A
	Keyword: Doca-installer; DOCA-Host; firmware extraction
	Reported in version: 3.1.0
4405369	Description: Hash action is not supported concurrently with mirroring.
	Workaround: N/A
	Keyword: OVS; ECMP; tunnel mirroring
	Reported in version: 3.1.0
4522588	Description: Installing any profile on Amazon Linux fails due to two issues: incorrect priorities assigned to inbox repositories and an unsupported signing method.

	Workaround: Change the repository priority and disable GPG checks by updating **/etc/yum.repos.d/doca.repo** to the following:
	`[doca]`
	`name=DOCA 3.1.0-056000`
	`baseurl=file:///usr/share/doca-host-3.1.0/repo`
	`enabled=1`
	`gpgkey=file:///usr/share/doca-host-3.1.0/repo/RPM-GPG-KEY-doca`
	`priority=10`
	`gpgcheck=0`

	Keyword: Installation
	Reported in version: 3.1.0
4352000	Description: Modified ptp4l to limit LISTENING port prints to only occur on Best Master Clock (BMC) changes.
	Workaround: N/A

Reference	Issue Details
	Keyword: PTP sync
	Reported in version: 3.1.0
4446815	Description: Since **mlxdevm** does not yet support lightweight SFs, attempting to activate an SF after setting **max_io_eqs** to zero will fail with the following error: "alloc_netdev: Unable to allocate device with zero RX queues."
	Workaround: N/A
	Keyword: Lightweight SF, mlxdevm
	Reported in version: 3.1.0
4449280	Description: When one port on the switch is shut down, the driver does not update the LAG map.
	Workaround: To ensure proper ECMP behavior, set **ignore_routes_with_linkdown** to 1.
	Keyword: ECMP
	Reported in version: 3.1.0
4404719	Description: See DOCA-HOST 3.1.0 (July 2025) changes in section Customer Affecting Changes.
	Workaround: N/A
	Keyword: PCI information
	Reported in version: 3.0.0
4273881	Description: PCI information is missing on RedHat p host.
	Workaround: Matching an interface name with its PCI address requires running: **ll /sys/class/net/ | grep <interface name>**
	Keyword: PCI information
	Reported in version: 3.0.0
4155701	Description: When offloading xfrm states to hardware, the offloading device is linked to the skb's secpath. If an skb is freed or deferred, an unregister netdevice operation may hang because the netdevice is still being reference-counted.
	Workaround: Remove the netdevice from the xfrm states when the netdevice is unregistered.
	Keyword: IPSec Crypto Offload
	Reported in version: 2.10.0

2.2.8.3 BlueField-3 Firmware Known Issues

Internal Ref.	Issue
4970263 / 4989004	**Description:** A high-temperature assert may be reported only the first time the chip reaches the thermal threshold.
	Workaround: N/A
	Keywords: Temperature

Internal Ref.	Issue
	Detected in version: 32.49.1014
4931335	**Description:** RoCE LAG is not supported when BlueField-3 operates in Socket Direct mode with NIC mode enabled. Although a Linux bonding interface can be created, the RoCE LAG device will not appear, and individual port devices will be shown instead.
	Workaround: N/A
	Keywords: ROCE, LAG, NIC mode, BlueField-3
	Detected in version: 32.50.1002
5079086 / 5096997	**Description:** Resolved an issue where plug and unplug operations failed to complete when the host was powered off.
	Workaround: N/A
	Keywords: Plug and unplug operations
	Detected in version: 32.50.1002
5173135 / 5074561	**Description:** The execution time of the flexio_al_outbox_set_gvmi function may increase by approximately 20%.
	Workaround: N/A
	Keywords: DPA_OUTBOX, credit leakage, flexio_al_outbox_set_gvmi
	Detected in version: 32.50.1002
4901395 / 4919262	**Description:** When PXE is enabled for both Legacy (FlexBoot) and UEFI drivers, the UEFI driver supports only a single TFTP session after the DHCP handshake.
	Workaround: For scenarios that require multiple concurrent TFTP sessions, disable PXE filters via the HCA HII menu.
	Keywords: PXE, Filter enabled, UEFI
	Detected in version: 32.48.1000
4999151	**Description:** When erasing a large number of VFs (64+) and immediately performing an ARM reset, the ARM OS may not be fully up yet.
	Workaround: To resolve the issue, add a delay between the two actions (VF erase and ARM reset).
	Keywords: VFs, ARM reset
	Detected in version: 32.49.1014
4776492	**Description:** Occasionally, upgrading PLDM BFB from DOCA v3.2.0 to v3.2.1 may lead to an assert 0x7 in dmesg.
	Workaround: Install DOCA v3.2.1 bfb file using "bfb-install" command, and power cycle the server.
	Keywords: PLDM
	Detected in version: 32.47.1088

Internal Ref.	Issue
4604969	**Description:** Probe packets might be dropped at the transmission stage when multiple congestion control flows are active.
	Workaround: N/A
	Keywords: PCC, RTT, probe
	Detected in version: 32.47.1026
4683823	**Description:** Some diagnostic data counters share hardware resources and cannot be configured simultaneously since 64-bit counter formats (e.g., DIAG_DATA_PARAMS_CONTEXT.output_format set to FORMAT_0 or FORMAT_1) consume more hardware resources per counter.
	Workaround: If a NO_RESOURCES error occurs, use output_format FORMAT_2 to reduce resource usage.
	Keywords: DOCA Telemetry Diagnostics
	Detected in version: 32.47.1026
4685736	**Description:** Creating a DPA process that allocates a 128 MB data segment and loads a dynamic library may fail with syndrome 0xdc30ac.
	Workaround: Limit the DPA application's data segment size to 64 MB.
	Keywords: DPA
	Detected in version: 32.47.1026
4535791	**Description:** When running sync2 on an EP configuration, the following error may appear during the MLXFWReset sync2 operation: **ERROR: System Off: operation not handled**. Although sync2 is intended for use in a switch topology, it can technically run on an EP configuration. However, this is not the default mode, nor is it a typical or recommended use case.
	Workaround: N/A
	Keywords: sync2, EP configuration
	Detected in version: 32.46.1006
4394475	**Description:** The existing congestion control configuration applies globally, rather than on a per-priority basis.
	Workaround: Ensure that the configuration values for all priorities are aligned in either `mlxconfig ROCE_CC_PRIO_MASK_P$port` or `sysfs ecn/roce_rp/enable/$port`.
	Keywords: Congestion control, ROCE_CC_PRIO
	Detected in version: 32.45.1020
4422120	**Description:** Any BFB upgrade from the October GA (2.9.2) to the new BFB will trigger a 0x00b4 assert. Nonetheless, the update will complete successfully, and the customer can safely ignore the assert.
	Workaround: N/A
	Keywords: BFB upgrade

Internal Ref.	Issue
	Detected in version: 32.45.1020
4216761	**Description:** For all host-related counters, the buffers used by the Arm are the same as those used by the host. Buffer usage is tracked collectively, combining both ARM and host consumption.
	Workaround: N/A
	Keywords: Counters
	Detected in version: 32.45.1020
4125431	**Description:** The MKEY created by software (VirtIo.Net DPA App is created with a length of 1 byte and used to access L2 memory. Since the minimum translation size is 64 bytes, using a 1-byte MKEY results in a translation error and triggers an exception.
	Workaround: N/A
	Keywords: MKEY
	Detected in version: 32.45.1020
4303583	**Description:** The query_header_modify_pattern command may produce inaccurate results when specific fields are used.
	Workaround: N/A
	Keywords: query_header_modify_pattern command
	Detected in version: 32.45.1020
4296168	**Description:** Running mlxfwreset fails when the DPU is configured as the root complex for NVMe drives. This issue impacts the configuration use case where the DPU acts as the root complex for NVMe drives, rather than the BF-3 in the host functioning as a PCIe Switch for the NVMe.
	Workaround: To ensure the firmware reset works correctly, explicitly run the fwreset command from the host using the "--method 1" flag (hot reset).
	Keywords: mlxfwreset
	Detected in version: 32.45.1020
4193036	**Description:** The initial allocation of DPA_THREAD on group affinity allocates memory for all EUs, including stack, core dump, and other resources.
	Workaround: N/A
	Keywords: DPA
	Detected in version: 32.44.1036
4007228	**Description:** NC-SI pass-through requires the user to allocate a MAC address to the platform BMC.
	Workaround: N/A
	Keywords: NC-SI pass-through
	Discovered in Version: 32.41.1000

Internal Ref.	Issue
3787618	**Description:** NVIA register is not allowed for external host if any field of EXTERNAL_HOST_PRIV or EXTERNAL_HOST_PRIV_FAST TLVs is not set as the default.
	Workaround: N/A
	Keywords: Host privilege
	Discovered in Version: 32.41.1000
3636631	**Description:** When configuring BlueField-3 Arm cores as PCIe root-complex, all non-mlx5 devices must always set the BlueField-3's IOMMU to disabled or passthrough mode. Turning IOMMU "ON" requires special handling of interrupts in the driver or the use of polling. For further assistance, contact NVIDIA support.
	Workaround: N/A
	Keywords: IOMMU
	Discovered in Version: 32.39.2048
3614529	**Description:** The supported DDR5 link speed in SKU B3220, is 5200 MT/s.
	Workaround: N/A
	Keywords: DDR5 link speed
	Discovered in Version: 32.39.2048
3728450	**Description:** SW_RESET with a pending image is currently not supported.
	Workaround: N/A
	Keywords: SW_RESET
	Discovered in Version: 32.39.2048
3614288	**Description:** Occasionally, the device may hang when there a hot plug is performed from a unknown direction.
	Workaround: N/A
	Keywords: Hot-plug operation
	Discovered in Version: 32.39.2048
-	**Description:** The I^2C clock fall time is lower than the 12ns minimum defined in the I2C-bus specification. For further information, refer to the I^2C-bus Specification, Version 7.0, October 2021, https://www.i2c-bus.org/.
	Workaround: N/A
	Keywords: I^2C clock
	Discovered in Version: 32.39.2048
3439438	**Description:** When connecting to a High Speed Traffic Generator in 400G speed, the linkup time may takes up to 3 minutes.
	Workaround: N/A

Internal Ref.	Issue
	Keywords: 400G linkup time
	Discovered in Version: 32.38.1002
3534128	**Description:** External flash access such as flash read using the MFT tools will fail if there is a pending image on the flash.
	Workaround: N/A
	Keywords: Flash access
	Discovered in Version: 32.38.1002
3534219	**Description:** On BlueField-3 devices, from DOCA 2.2.0 to 32.37.1306 (or lower), the host crashes when executing partial Arm reset (e.g., Arm reboot; BFB push; mlxfwreset).
	Workaround: Before downgrading the firmware, perform: • echo 0 > /sys/bus/platform/drivers/mlxbf-bootctl/large_icm • Arm reboot
	Keywords: BlueField-3; downgrade
	Discovered in Version: 32.38.1002
3547022	**Description:** When unloading the network drivers on an external host, sync1 reset may be still reported as 'supported' although it is not. Thus, initiating the reset flow may result in reset failure after a few minutes.
	Workaround: N/A
	Keywords: Sync1 reset
	Discovered in Version: 32.38.1002
3439438	**Description:** When connecting to a Spirent switch in 400G speed, the linkup time may takes up to 3 minutes.
	Workaround: N/A
	Keywords: Spirent, 400G, linkup time
	Discovered in Version: 32.38.1002
3178339	**Description:** PCIe PML1 is disabled.
	Workaround: N/A
	Keywords: PCIe PML1
	Discovered in Version: 32.38.1002
3525865	**Description:** Unexpected system behavior might be observed if the driver is loaded while reset is in progress.
	Workaround: N/A
	Keywords: Sync 1 reset, firmware reset
	Discovered in Version: 32.38.1002

Internal Ref.	Issue
3275394	**Description:** When performing PCIe link secondary-bus-reset, disable/enable or mlxfwreset on AMD based Genoa systems, the device takes longer then expected to link up, due to a PCIe receiver termination misconfiguration.
	Workaround: N/A
	Keywords: PCIe
	Discovered in Version: 32.37.1306
2878841	**Description:** The firmware rollback fails for the signature retransmit flow if the QPN field is configured in the mkey (as it only allows the given QP to use this Mkey) as the firmware rollback flow relies on an internal QP that uses the mkey.
	Workaround: N/A
	Keywords: Signature retransmit flow
	Discovered in Version: 32.37.1306
3412847	**Description:** Socket-Direct is currently not supported.
	Workaround: N/A
	Keywords: Socket-Direct
	Discovered in Version: 32.37.1306

2.2.8.4 BlueField-2 Firmware Known Issues

Internal Ref.	Issue
4970263 / 4989004	**Description:** A high-temperature assert may be reported only the first time the chip reaches the thermal threshold.
	Workaround: N/A
	Keywords: Temperature
	Detected in version: 24.49.1014
4901395 / 4919262	**Description:** When PXE is enabled for both Legacy (FlexBoot) and UEFI drivers, the UEFI driver supports only a single TFTP session after the DHCP handshake.
	Workaround: For scenarios that require multiple concurrent TFTP sessions, disable PXE filters via the HCA HII menu.
	Keywords: PXE, Filter enabled, UEFI
	Detected in version: 24.48.1000
4394475	**Description:** The existing congestion control configuration applies globally, rather than on a per-priority basis.
	Workaround: Ensure that the configuration values for all priorities are aligned in either `mlxconfig ROCE_CC_PRIO_MASK_P$port` or `sysfs ecn/roce_rp/enable/$port`.

Internal Ref.	Issue
	Keywords: Congestion control, ROCE_CC_PRIO
	Detected in version: 24.45.1020
3754913	**Description:** PHYless Reset is currently not supported.
	Workaround: N/A
	Keywords: PHYless Reset
	Discovered in Version: 24.40.1000
3525865	**Description:** Unexpected system behavior might be observed if the driver is loaded while reset is in progress.
	Workaround: N/A
	Keywords: Sync 1 reset, firmware reset
	Discovered in Version: 24.39.2048
-	**Description:** When `tx_port_ts` is set to `"true"`, due to a compensation mechanism in the Tx timestamp available in some hardware Rx timestamp errors, a symmetrical error and no clock offset occur when using the timestamps to synchronize the device clock. This might also cause an error while using timestamps for delay measurements (e,g., delay measurements reported by a PTP daemon) and even negative delay measurements in some cases.
	Workaround: N/A
	Keywords: PTP path delay
	Discovered in Version: 24.38.1002
2878841	**Description:** The firmware rollback fails for the signature retransmit flow if the QPN field is configured in the mkey (as it only allows the given QP to use this Mkey) as the firmware rollback flow relies on an internal QP that uses the mkey.
	Workaround: N/A
	Keywords: Signature retransmit flow
	Discovered in Version: 24.37.1300
3329109	**Description:** MFS1S50-H003E cable supports only HDR rate when used as a split cable.
	Workaround: N/A
	Keywords: HDR, split cable, MFS1S50-H003E
	Discovered in Version: 24.37.1300
3267506	**Description:** CRC is included in the traffic byte counters as a port byte counter.
	Workaround: N/A
	Keywords: Counters, CRC
	Discovered in Version: 24.35.2000

Internal Ref.	Issue
3141072	**Description:** The "max_shaper_rate" configuration query via QEEC mlxreg returns a value translated to hardware granularity.
	Workaround: N/A
	Keywords: RX Rate-Limiter, Multi-host
	Discovered in Version: 24.34.1002
2870970	**Description:** GTP encapsulation (flex parser profile 3) is limited to the NIC domain. Encapsulating in the FDB domain will render a 0-size length in GTP header.
	Workaround: N/A
	Keywords: GTP encapsulation
	Discovered in Version: 24.34.1002
2870213	**Description:** Servers do not recover after configuring `PCI_SWITCH_EMULATION_NUM_PORT` to 32 followed by power cycle.
	Workaround: N/A
	Keywords: VirtIO-net; power cycle
	Discovered in Version: 24.33.1048
2855592	**Description:** When working with 3rd party device (e.g., Paragon) in 25GbE speed, the 25GbE speed must be configured in force mode.
	Workaround: N/A
	Keywords: Force mode, 3rd party devices, 25GbE
	Discovered in Version: 24.33.1048
2850003	**Description:** Occasionally, when rising a logical link, the link recovery counter is increase by 1.
	Workaround: N/A
	Keywords: Link recovery counter
	Discovered in Version: 24.33.1048
2616755	**Description:** Forward action for IPoIB is not supported on RX RDMA Flow Table.
	Workaround: N/A
	Keywords: Steering, IPoIB
	Discovered in Version: 24.33.1048

Internal Ref.	Issue
4970263 / 4989004	**Description:** A high-temperature assert may be reported only the first time the chip reaches the thermal threshold.
	Workaround: N/A
	Keywords: Temperature
	Detected in version: 24.49.1014
4901395 / 4919262	**Description:** When PXE is enabled for both Legacy (FlexBoot) and UEFI drivers, the UEFI driver supports only a single TFTP session after the DHCP handshake.
	Workaround: For scenarios that require multiple concurrent TFTP sessions, disable PXE filters via the HCA HII menu.
	Keywords: PXE, Filter enabled, UEFI
	Detected in version: 24.48.1000
4394475	**Description:** The existing congestion control configuration applies globally, rather than on a per-priority basis.
	Workaround: Ensure that the configuration values for all priorities are aligned in either `mlxconfig ROCE_CC_PRIO_MASK_P$port` or `sysfs ecn/roce_rp/enable/$port`.
	Keywords: Congestion control, ROCE_CC_PRIO
	Detected in version: 24.45.1020
3754913	**Description:** PHYless Reset is currently not supported.
	Workaround: N/A
	Keywords: PHYless Reset
	Discovered in Version: 24.40.1000
3525865	**Description:** Unexpected system behavior might be observed if the driver is loaded while reset is in progress.
	Workaround: N/A
	Keywords: Sync 1 reset, firmware reset
	Discovered in Version: 24.39.2048
-	**Description:** When `tx_port_ts` is set to `"true"`, due to a compensation mechanism in the Tx timestamp available in some hardware Rx timestamp errors, a symmetrical error and no clock offset occur when using the timestamps to synchronize the device clock. This might also cause an error while using timestamps for delay measurements (e,g., delay measurements reported by a PTP daemon) and even negative delay measurements in some cases.
	Workaround: N/A
	Keywords: PTP path delay
	Discovered in Version: 24.38.1002

Internal Ref.	Issue
2878841	**Description:** The firmware rollback fails for the signature retransmit flow if the QPN field is configured in the mkey (as it only allows the given QP to use this Mkey) as the firmware rollback flow relies on an internal QP that uses the mkey.
	Workaround: N/A
	Keywords: Signature retransmit flow
	Discovered in Version: 24.37.1300
3329109	**Description:** MFS1S50-H003E cable supports only HDR rate when used as a split cable.
	Workaround: N/A
	Keywords: HDR, split cable, MFS1S50-H003E
	Discovered in Version: 24.37.1300
3267506	**Description:** CRC is included in the traffic byte counters as a port byte counter.
	Workaround: N/A
	Keywords: Counters, CRC
	Discovered in Version: 24.35.2000
3141072	**Description:** The "max_shaper_rate" configuration query via QEEC mlxreg returns a value translated to hardware granularity.
	Workaround: N/A
	Keywords: RX Rate-Limiter, Multi-host
	Discovered in Version: 24.34.1002
2870970	**Description:** GTP encapsulation (flex parser profile 3) is limited to the NIC domain. Encapsulating in the FDB domain will render a 0-size length in GTP header.
	Workaround: N/A
	Keywords: GTP encapsulation
	Discovered in Version: 24.34.1002
2870213	**Description:** Servers do not recover after configuring `PCI_SWITCH_EMULATION_NUM_PORT` to 32 followed by power cycle.
	Workaround: N/A
	Keywords: VirtIO-net; power cycle
	Discovered in Version: 24.33.1048
2855592	**Description:** When working with 3rd party device (e.g., Paragon) in 25GbE speed, the 25GbE speed must be configured in force mode.
	Workaround: N/A
	Keywords: Force mode, 3rd party devices, 25GbE

Internal Ref.	Issue
	Discovered in Version: 24.33.1048
2850003	**Description:** Occasionally, when rising a logical link, the link recovery counter is increase by 1.
	Workaround: N/A
	Keywords: Link recovery counter
	Discovered in Version: 24.33.1048
2616755	**Description:** Forward action for IPoIB is not supported on RX RDMA Flow Table.
	Workaround: N/A
	Keywords: Steering, IPoIB
	Discovered in Version: 24.33.1048

2.2.9 BlueField Firmware Cables

The firmware version included in this DOCA release supports the following cables.

- BlueField-3 Validated and Supported Cables and Modules
- BlueField-2 Validated and Supported Cables and Modules

2.2.9.1 BlueField-3 Validated and Supported Cables and Modules

The BlueField-3 firmware version included in this DOCA release supports the following cables.

2.2.9.1.1 Cables Lifecycle Legend

Lifecycle Phase	Definition
EOL	End of Life
LTB	Last Time Buy
HVM	GA level
MP	GA level
P-Rel	GA level
Preliminary	Engineering Sample
Prototype	Engineering Sample

2.2.9.1.2 NDR / 400GbE / 800GbE Cables

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
OSFP to 2xQSFP112	NDR	800GE	980-9I81B-00N004	MCA7J65-N004	NVIDIA Active copper splitter cable, 800(2x400)Gbps to 2x400Gbps, OSFP to 2xQSFP112, 4m, fin to flat	EOL [P-Rel]
OSFP to 2xQSFP112	NDR	800GE	980-9I81C-00N005	MCA7J65-N005	NVIDIA Active copper splitter cable, 800(2x400)Gbps to 2x400Gbps, OSFP to 2xQSFP112, 5m, fin to flat	EOL [P-Rel]
OSFP to 4xQSFP112	NDR	800GE	980-9I76G-00N004	MCA7J75-N004	NVIDIA Active copper splitter cable, 800(2x400)Gbps to 4x200Gbps, OSFP to 4xQSFP112, 4m, fin to flat	EOL [P-Rel]
OSFP to 4xQSFP112	NDR	800GE	980-9I76H-00N005	MCA7J75-N005	NVIDIA Active copper splitter cable, 800(2x400)Gbps to 4x200Gbps, OSFP to 4xQSFP112, 5m, fin to flat	EOL [P-Rel]
OSFP to 2xQSFP112	NDR	800GE	980-9I928-00N001	MCP7Y10-N001	NVIDIA passive copper splitter cable, 800(2x400)Gbps to 2x400Gbps, OSFP to 2xQSFP112,1m, fin to flat	LTB [P-Rel]
OSFP to 2xQSFP112	NDR	800GE	980-9I929-00N002	MCP7Y10-N002	NVIDIA passive copper splitter cable, 800(2x400)Gbps to 2x400Gbps, OSFP to 2xQSFP112,2m, fin to flat	LTB [P-Rel]
OSFP to 2xQSFP112	NDR	800GE	980-9I80P-00N003	MCP7Y10-N003	NVIDIA passive copper splitter cable, 800(2x400)Gbps to 2x400Gbps, OSFP to 2xQSFP112,3m, fin to flat	LTB [P-Rel]
OSFP to 2xQSFP112	NDR	800GE	980-9I80A-00N01A	MCP7Y10-N01A	NVIDIA passive copper splitter cable, 800(2x400)Gbps to 2x400Gbps, OSFP to 2xQSFP112,1.5m, fin to flat	EOL [P-Rel]
OSFP to 2xQSFP112	NDR	800GE	980-9I80Q-00N02A	MCP7Y10-N02A	NVIDIA passive copper splitter cable, 800(2x400)Gbps to 2x400Gbps, OSFP to 2xQSFP112,2.5m, fin to flat	EOL [P-Rel]
OSFP to 4xQSFP112	NDR	800GE	980-9I80B-00N001	MCP7Y40-N001	NVIDIA passive copper splitter cable, 800(2x400)Gbps to 4x200Gbps, OSFP to 4xQSFP112, 1m, fin to flat	LTB [P-Rel]
OSFP to 4xQSFP112	NDR	800GE	980-9I80C-00N002	MCP7Y40-N002	NVIDIA passive copper splitter cable, 800(2x400)Gbps to 4x200Gbps, OSFP to 4xQSFP112, 2m, fin to flat	LTB [P-Rel]
OSFP to 4xQSFP112	NDR	800GE	980-9I75R-00N003	MCP7Y40-N003	NVIDIA passive copper splitter cable, 800(2x400)Gbps to 4x200Gbps, OSFP to 4xQSFP112, 3m, fin to flat	LTB [P-Rel]
OSFP to 4xQSFP112	NDR	800GE	980-9I75D-00N01A	MCP7Y40-N01A	NVIDIA passive copper splitter cable, 800(2x400)Gbps to 4x200Gbps, OSFP to 4xQSFP112, 1.5m, fin to flat	EOL [P-Rel]
OSFP to 4xQSFP112	NDR	800GE	980-9I75S-00N02A	MCP7Y40-N02A	NVIDIA passive copper splitter cable, 800(2x400)Gbps to 4x200Gbps, OSFP to 4xQSFP112, 2.5m, fin to flat	EOL [P-Rel]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP112	NDR	400GE	980-9I693-00NS00	MMA1Z00-NS400	NVIDIA single port transceiver, 400Gbps, QSFP112, MPO12 APC, 850nm MMF, up to 50m, flat top	P-Rel
QSFP112	NDR	400GE	980-9I068-00NM00	MMS1X00-NS400	NVIDIA single port transceiver, 400Gbps, QSFP112, MPO, 1310nm SMF, up to 500m, flat top	P-Rel

2.2.9.1.3 HDR / 200GbE Cables

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP56	HDR	200GE	980-9I548-00H001	MCP1650-H001E30	Nvidia Passive Copper cable, up to 200Gbps, QSFP56 to QSFP56, 1m	HVM
QSFP56	HDR	200GE	980-9I549-00H002	MCP1650-H002E26	Nvidia Passive Copper cable,up to 200Gbps, QSFP56 to QSFP56, 2m	HVM
QSFP56	HDR	200GE	980-9I54A-00H00A	MCP1650-H00AE30	Nvidia Passive Copper cable, up to 200Gbps, QSFP56 to QSFP56, 0.5m	EOL [HVM]
QSFP56	HDR	200GE	980-9I54B-00H01A	MCP1650-H01AE30	Nvidia Passive Copper cable, up to 200Gbps, QSFP56 to QSFP56, 1.5 m	HVM
QSFP56	NA	200GE	980-9I54C-00V001	MCP1650-V001E30	Mellanox Passive Copper cable, 200GbE, 200Gb/s, QSFP56, LSZH, 1m, black pulltab, 30AWG	EOL [HVM]
QSFP56	NA	200GE	980-9I54D-00V002	MCP1650-V002E26	Mellanox Passive Copper cable, 200GbE, 200Gb/s, QSFP56, LSZH, 2m, black pulltab, 26AWG	EOL [HVM]
QSFP56	NA	200GE	980-9I54G-00V003	MCP1650-V003E26	Mellanox Passive Copper cable, 200GbE, 200Gb/s, QSFP56, LSZH, 3m, black pulltab, 26AWG	EOL [HVM]
QSFP56	NA	200GE	980-9I54H-00V00A	MCP1650-V00AE30	Mellanox Passive Copper cable, 200GbE, 200Gb/s, QSFP56, LSZH, 0.5m, black pulltab, 30AWG	EOL [HVM]
QSFP56	NA	200GE	980-9I54I-00V01A	MCP1650-V01AE30	Mellanox Passive Copper cable, 200GbE, 200Gb/s, QSFP56, LSZH, 1.5m, black pulltab, 30AWG	EOL [HVM]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP56	NA	200GE	980-9I54L-00V02A	MCP1650-V02AE26	Mellanox Passive Copper cable, 200GbE, 200Gb/s, QSFP56, LSZH, 2.5m, black pulltab, 26AWG	EOL [HVM]
QSFP56 to 2xQSFP56	HDR	200GE	980-9I39E-00H001	MCP7H50-H001R30	Nvidia Passive copper splitter cable, 200Gbps to 2x100Gbps, QSFP56 to 2xQSFP56, 1m	EOL [HVM]
QSFP56 to 2xQSFP56	HDR	200GE	980-9I99F-00H002	MCP7H50-H002R26	Nvidia Passive copper splitter cable, 200Gbps to 2x100Gbps, QSFP56 to 2xQSFP56, 2m	EOL [HVM]
QSFP56 to 2xQSFP56	HDR	200GE	980-9I98G-00H01A	MCP7H50-H01AR30	Nvidia Passive copper splitter cable, 200Gbps to 2x100Gbps, QSFP56 to 2xQSFP56, 1.5m	EOL [HVM]
QSFP56 to 2xQSFP56	NA	200GE	980-9I98H-00V001	MCP7H50-V001R30	Mellanox passive copper hybrid cable, 200GbE 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, colored, 1m, 30AWG	EOL [HVM]
QSFP56 to 2xQSFP56	NA	200GE	980-9I98I-00V002	MCP7H50-V002R26	Mellanox passive copper hybrid cable, 200GbE 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, colored, 2m, 26AWG	EOL [HVM]
QSFP56 to 2xQSFP56	NA	200GE	980-9I98J-00V003	MCP7H50-V003R26	Mellanox passive copper hybrid cable, 200GbE 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, colored, 3m, 26AWG	EOL [HVM]
QSFP56 to 2xQSFP56	NA	200GE	980-9I98K-00V01A	MCP7H50-V01AR30	Mellanox passive copper hybrid cable, 200GbE 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, colored, 1.5m, 30AWG	EOL [HVM]
QSFP56 to 2xQSFP56	NA	200GE	980-9I98M-00V02A	MCP7H50-V02AR26	Mellanox passive copper hybrid cable, 200GbE 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, colored, 2.5m, 26AWG	EOL [HVM]
QSFP56 to 4xSFP56	NA	200GE	980-9IA3X-00V001	MCP7H70-V001R30	Mellanox passive copper hybrid cable, 200GbE 200Gb/s to 4x50Gb/s, QSFP56 to 4xSFP56, colored, 1m, 30AWG	EOL [P-Rel]
QSFP56 to 4xSFP56	NA	200GE	980-9IA3Y-00V002	MCP7H70-V002R26	Mellanox passive copper hybrid cable, 200GbE 200Gb/s to 4x50Gb/s, QSFP56 to 4xSFP56, colored, 2m, 26AWG	EOL [P-Rel]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP56 to 4xSFP56	NA	200GE	980-9I43Z-00V003	MCP7H70-V003R26	Mellanox passive copper hybrid cable, 200GbE 200Gb/s to 4x50Gb/s, QSFP56 to 4x4SFP56, colored, 3m, 26AWG	EOL [P-Rel]
QSFP56 to 4xSFP56	NA	200GE	980-9I430-00V01A	MCP7H70-V01AR30	Mellanox passive copper hybrid cable, 200GbE 200Gb/s to 4x50Gb/s, QSFP56 to 4xSFP56, colored, 1.5m, 30AWG	EOL [P-Rel]
QSFP56 to 4xSFP56	NA	200GE	980-9I431-00V02A	MCP7H70-V02AR26	Mellanox passive copper hybrid cable, 200GbE 200Gb/s to4x50Gb/s, QSFP56 to 4xSFP56, colored, 2.5m, 26AWG	EOL [P-Rel]
QSFP56	HDR	200GE	980-9I457-00H003	MFS1S00-H003V	Nvidia active optical cable, up to 200Gbps , QSFP56 to QSFP56, 3m	EOL [MP]
QSFP56	HDR	200GE	980-9I45D-00H005	MFS1S00-H005V	Nvidia active optical cable, up to 200Gbps , QSFP56 to QSFP56, 5m	MP
QSFP56	HDR	200GE	980-9I45J-00H010	MFS1S00-H010V	Nvidia active optical cable, up to 200Gbps , QSFP56 to QSFP56, 10m	MP
QSFP56	HDR	200GE	980-9I45O-00H015	MFS1S00-H015V	Nvidia active optical cable, up to 200Gbps , QSFP56 to QSFP56, 15m	EOL [MP]
QSFP56	HDR	200GE	980-9I45T-00H020	MFS1S00-H020V	Nvidia active optical cable, up to 200Gbps , QSFP56 to QSFP56, 20m	MP
QSFP56	HDR	200GE	980-9I440-00H030	MFS1S00-H030V	Nvidia active optical cable, up to 200Gbps , QSFP56 to QSFP56, 30m	MP
QSFP56	HDR	200GE	980-9I447-00H050	MFS1S00-H050V	Nvidia active optical cable, up to 200Gbps , QSFP56 to QSFP56, 50m	MP
QSFP56	HDR	200GE	980-9I44H-00H100	MFS1S00-H100V	Nvidia active optical cable, up to 200Gbps , QSFP56 to QSFP56, 100m	MP
QSFP56	HDR	200GE	980-9I44K-00H130	MFS1S00-H130V	Nvidia active optical cable, up to 200Gbps , QSFP56 to QSFP56, 130m	EOL [MP]
QSFP56	NA	200GE	980-9I44P-00V003	MFS1S00-V003E	Mellanox active fiber cable, 200GbE, 200Gb/s, QSFP56, LSZH, black pulltab, 3m	EOL [HVM]
QSFP56	NA	200GE	980-9I45Q-00V005	MFS1S00-V005E	Mellanox active fiber cable, 200GbE, 200Gb/s, QSFP56, LSZH, black pulltab, 5m	EOL [HVM]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP56	NA	200GE	980-9I45R-00V010	MFS1S00-V010E	Mellanox active fiber cable, 200GbE, 200Gb/s, QSFP56, LSZH, black pulltab, 10m	EOL [HVM]
QSFP56	NA	200GE	980-9I44S-00V015	MFS1S00-V015E	Mellanox active fiber cable, 200GbE, 200Gb/s, QSFP56, LSZH, black pulltab, 15m	EOL [HVM]
QSFP56	NA	200GE	980-9I44T-00V020	MFS1S00-V020E	Mellanox active fiber cable, 200GbE, 200Gb/s, QSFP56, LSZH, black pulltab, 20m	EOL [HVM]
QSFP56	NA	200GE	980-9I44U-00V030	MFS1S00-V030E	Mellanox active fiber cable, 200GbE, 200Gb/s, QSFP56, LSZH, black pulltab, 30m	EOL [HVM]
QSFP56	NA	200GE	980-9I44V-00V050	MFS1S00-V050E	Mellanox active fiber cable, 200GbE, 200Gb/s, QSFP56, LSZH, black pulltab, 50m	EOL [HVM]
QSFP56	NA	200GE	980-9I44W-00V100	MFS1S00-V100E	Mellanox active fiber cable, 200GbE, 200Gb/s, QSFP56, LSZH, black pulltab, 100m	EOL [HVM] [HIBERNATE]
QSFP56 to 2xQSFP56	HDR	200GE	980-9I445-00H003	MFS1S50-H003V	Nvidia active optical splitter cable, 200Gbps to 2x100Gbps , QSFP56 to 2x QSFP56, 3m	EOL [HVM]
QSFP56 to 2xQSFP56	HDR	200GE	980-9I969-00H005	MFS1S50-H005V	Nvidia active optical splitter cable, 200Gbps to 2x100Gbps , QSFP56 to 2x QSFP56, 5m	EOL [HVM]
QSFP56 to 2xQSFP56	HDR	200GE	980-9I96D-00H010	MFS1S50-H010V	Nvidia active optical splitter cable, 200Gbps to 2x100Gbps , QSFP56 to 2x QSFP56, 10m	EOL [HVM]
QSFP56 to 2xQSFP56	HDR	200GE	980-9I96H-00H015	MFS1S50-H015V	Nvidia active optical splitter cable, 200Gbps to 2x100Gbps , QSFP56 to 2x QSFP56, 15m	EOL [HVM]
QSFP56 to 2xQSFP56	HDR	200GE	980-9I96L-00H020	MFS1S50-H020V	Nvidia active optical splitter cable, 200Gbps to 2x100Gbps , QSFP56 to 2x QSFP56, 20m	EOL [HVM]
QSFP56 to 2xQSFP56	HDR	200GE	980-9I96P-00H030	MFS1S50-H030V	Nvidia active optical splitter cable, 200Gbps to 2x100Gbps , QSFP56 to 2x QSFP56, 30m	EOL [HVM]
QSFP56 to 2xQSFP56	NA	200GE	980-9I95Q-00V003	MFS1S50-V003E	Mellanox active fiber splitter cable, 200GbE, 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, LSZH, black pulltab, 3m	EOL [HVM]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP56 to 2xQSFP56	NA	200GE	980-9I96R-00V005	MFS1S50-V005E	Mellanox active fiber splitter cable, 200GbE, 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, LSZH, black pulltab, 5m	EOL [HVM]
QSFP56 to 2xQSFP56	NA	200GE	980-9I96S-00V010	MFS1S50-V010E	Mellanox active fiber splitter cable, 200GbE, 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, LSZH, black pulltab, 10m	EOL [HVM]
QSFP56 to 2xQSFP56	NA	200GE	980-9I96T-00V015	MFS1S50-V015E	Mellanox active fiber splitter cable, 200GbE, 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, LSZH, black pulltab, 15m	EOL [HVM]
QSFP56 to 2xQSFP56	NA	200GE	980-9I95U-00V020	MFS1S50-V020E	Mellanox active fiber splitter cable, 200GbE, 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, LSZH, black pulltab, 20m	EOL [HVM]
QSFP56 to 2xQSFP56	NA	200GE	980-9I95V-00V030	MFS1S50-V030E	Mellanox active fiber splitter cable, 200GbE, 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, LSZH, black pulltab, 30m	EOL [HVM]
QSFP56	NA	200GE	980-9I20T-00V000	MMA1T00-VS	Mellanox transceiver, 200GbE, up to 200Gb/s, QSFP56, MPO, 850nm, SR4, up to 100m	EOL [HVM]

2.2.9.1.4 EDR / 100GbE Cables

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP28	NA	100GE	980-9I620-00C001	MCP1600-C001E30N	Mellanox Passive Copper cable, ETH 100GbE, 100Gb/s, QSFP28, 1m, Black, 30AWG, CA-N	EOL [HVM]
QSFP28	NA	100GE	980-9I62V-00C002	MCP1600-C002E30N	Mellanox Passive Copper cable, ETH 100GbE, 100Gb/s, QSFP28, 2m, Black, 30AWG, CA-N	EOL [HVM]
QSFP28	NA	100GE	980-9I62Z-00C003	MCP1600-C003E26N	Mellanox Passive Copper cable, ETH 100GbE, 100Gb/s, QSFP28, 3m, Black, 26AWG, CA-N	EOL [HVM]
QSFP28	NA	100GE	980-9I620-00C003	MCP1600-C003E30L	Mellanox Passive Copper cable, ETH 100GbE, 100Gb/s, QSFP28, 3m, Black, 30AWG, CA-L	EOL [HVM]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP	NA	100GE	980-9I625-00C005	MCP1600-C005E26L	Mellanox Passive Copper cable, ETH 100GbE, 100Gb/s, QSFP28, 5m, Black, 26AWG, CA-L	EOL [HVM]
QSFP28	NA	100GE	980-9I627-00C00A	MCP1600-C00AE30N	Mellanox Passive Copper cable, ETH 100GbE, 100Gb/s, QSFP28, 0.5m, Black, 30AWG, CA-N	EOL [HVM]
QSFP28	NA	100GE	980-9I629-00C00B	MCP1600-C00BE30N	Mellanox Passive Copper cable, ETH 100GbE, 100Gb/s, QSFP28, 0.75m, Black, 30AWG, CA-N	EOL [HVM]
QSFP28	NA	100GE	980-9I62C-00C01A	MCP1600-C01AE30N	Mellanox Passive Copper cable, ETH 100GbE, 100Gb/s, QSFP28, 1.5m, Black, 30AWG, CA-N	EOL [HVM]
QSFP28	NA	100GE	980-9I62H-00C02A	MCP1600-C02AE26N	Mellanox Passive Copper cable, ETH 100GbE, 100Gb/s, QSFP28, 2.5m, Black, 26AWG, CA-N	EOL [HVM]
QSFP28	NA	100GE	980-9I62I-00C02A	MCP1600-C02AE30L	Mellanox Passive Copper cable, ETH 100GbE, 100Gb/s, QSFP28,2.5m, Black, 30AWG, CA-L	EOL [HVM]
QSFP	EDR	100GE	980-9I62V-00C003	MCP1600-E003	Mellanox Passive Copper cable, IB EDR, up to 100Gb/s, QSFP, LSZH, 3m 26AWG	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I486-00C001	MCP7F00-A001R30N	Mellanox passive copper hybrid cable, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 1m, Colored, 30AWG, CA-N	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I48B-00C002	MCP7F00-A002R30N	Mellanox passive copper hybrid cable, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 2m, Colored, 30AWG, CA-N	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I48G-00C003	MCP7F00-A003R26N	Mellanox passive copper hybrid cable, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 3m, Colored, 26AWG, CA-N	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I48H-00C003	MCP7F00-A003R30L	Mellanox passive copper hybrid cable, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 3m, Colored, 30AWG, CA-L	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I48J-00C005	MCP7F00-A005R26L	Mellanox passive copper hybrid cable, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 5m, Colored, 26AWG, CA-L	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I48N-00C01A	MCP7F00-A01AR30N	Mellanox passive copper hybrid cable, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 1.5m, Colored, 30AWG, CA-N	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I48S-00C02A	MCP7F00-A02AR26N	Mellanox passive copper hybrid cable, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 2.5m, Colored, 26AWG, CA-N	EOL [HVM]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP28 to 4xSFP28	NA	100GE	980-9I48T-00C02A	MCP7F00-A02AR30L	Mellanox passive copper hybrid cable, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 2.5m, Colored, 30AWG, CA-L	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I48X-00C03A	MCP7F00-A03AR26L	Mellanox passive copper hybrid cable, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 3.5m, Colored, 26AWG, CA-L	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I99G-00C001	MCP7H00-G001R30N	Mellanox passive copper hybrid cable, ETH 100Gb/s to 2x50Gb/s, QSFP28 to 2xQSFP28, 1m, Colored, 30AWG, CA-N	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I99L-00C002	MCP7H00-G002R30N	Mellanox passive copper hybrid cable, ETH 100Gb/s to 2x50Gb/s, QSFP28 to 2xQSFP28, 2m, Colored, 30AWG, CA-N	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I99Q-00C003	MCP7H00-G003R26N	Mellanox passive copper hybrid cable, ETH 100Gb/s to 2x50Gb/s, QSFP28 to 2xQSFP28, 3m, Colored, 26AWG, CA-N	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I39R-00C003	MCP7H00-G003R30L	Mellanox passive copper hybrid cable, ETH 100Gb/s to 2x50Gb/s, QSFP28 to 2xQSFP28, 3m, Colored, 30AWG, CA-L	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I99S-00C004	MCP7H00-G004R26L	Mellanox passive copper hybrid cable, ETH 100Gb/s to 2x50Gb/s, QSFP28 to 2xQSFP28, 4m, Colored, 26AWG, CA-L	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I99X-00C01A	MCP7H00-G01AR30N	Mellanox passive copper hybrid cable, ETH 100Gb/s to 2x50Gb/s, QSFP28 to 2xQSFP28, 1.5m, Colored, 30AWG, CA-N	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I395-00C02A	MCP7H00-G02AR30L	Mellanox passive copper hybrid cable, ETH 100Gb/s to 2x50Gb/s, QSFP28 to 2xQSFP28, 2.5m, Colored, 30AWG, CA-L	EOL [HVM]
QSFP28	NA	100GE	980-9I13S-00C003	MFA1A00-C003	Mellanox active fiber cable, ETH 100GbE, 100Gb/s, QSFP, LSZH, 3m	EOL [HVM]
QSFP28	NA	100GE	980-9I13X-00C005	MFA1A00-C005	Mellanox active fiber cable, ETH 100GbE, 100Gb/s, QSFP, LSZH, 5m	EOL [HVM]
QSFP28	NA	100GE	980-9I134-00C010	MFA1A00-C010	Mellanox active fiber cable, ETH 100GbE, 100Gb/s, QSFP, LSZH, 10m	EOL [HVM]
QSFP28	NA	100GE	980-9I13A-00C015	MFA1A00-C015	Mellanox active fiber cable, ETH 100GbE, 100Gb/s, QSFP, LSZH, 15m	EOL [HVM]
QSFP28	NA	100GE	980-9I13F-00C020	MFA1A00-C020	Mellanox active fiber cable, ETH 100GbE, 100Gb/s, QSFP, LSZH, 20m	EOL [HVM]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP28	NA	100GE	980-9I13N-00C030	MFA1A00-C030	Mellanox active fiber cable, ETH 100GbE, 100Gb/s, QSFP, LSZH, 30m	EOL [HVM]
QSFP28	NA	100GE	980-9I130-00C050	MFA1A00-C050	Mellanox active fiber cable, ETH 100GbE, 100Gb/s, QSFP, LSZH, 50m	EOL [HVM]
QSFP28	NA	100GE	980-9I13B-00C100	MFA1A00-C100	Mellanox active fiber cable, ETH 100GbE, 100Gb/s, QSFP, LSZH, 100m	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I37H-00C003	MFA7A20-C003	Mellanox active fiber hybrid solution, ETH 100GbE to 2x50GbE, QSFP28 to 2xQSFP28, 3m	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I37I-00C005	MFA7A20-C005	Mellanox active fiber hybrid solution, ETH 100GbE to 2x50GbE, QSFP28 to 2xQSFP28, 5m	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I40J-00C010	MFA7A20-C010	Mellanox active fiber hybrid solution, ETH 100GbE to 2x50GbE, QSFP28 to 2xQSFP28, 10m	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I40K-00C020	MFA7A20-C020	Mellanox active fiber hybrid solution, ETH 100GbE to 2x50GbE, QSFP28 to 2xQSFP28, 20m	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I40N-00C003	MFA7A50-C003	Mellanox active fiber hybrid solution, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 3m	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I40O-00C005	MFA7A50-C005	Mellanox active fiber hybrid solution, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 5m	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I49P-00C010	MFA7A50-C010	Mellanox active fiber hybrid solution, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 10m	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I49Q-00C015	MFA7A50-C015	Mellanox active fiber hybrid solution, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 15m	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I49R-00C020	MFA7A50-C020	Mellanox active fiber hybrid solution, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 20m	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I49S-00C030	MFA7A50-C030	Mellanox active fiber hybrid solution, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 30m	EOL [HVM]
QSFP28	NA	100GE	980-9I149-00CS00	MMA1B00-C100D	Mellanox transceiver, 100GbE, QSFP28, MPO, 850nm, SR4, up to 100m, DDMI	HVM
QSFP	NA	100GE	980-9I17P-00CR00	MMA1L10-CR	Mellanox optical transceiver, 100GbE, 100Gb/s, QSFP28, LC-LC, 1310nm, LR4 up to 10km	LTB [HVM]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP	NA	100GE	980-9I17Q-00CM00	MMA1L30-CM	Mellanox optical module, 100GbE, 100Gb/s, QSFP28, LC-LC, 1310nm, CWDM4, up to 2km	LTB [MP]
QSFP	NA	100GE	980-9I16X-00C000	MMS1C10-CM	Mellanox active optical module, 100Gb/s, QSFP, MPO, 1310nm, PSM4, up to 500m	EOL [MP]
QSFP28	NA	100GE	980-9I042-00C000	MMS1V70-CM	Mellanox transceiver, 100GbE, QSFP28, LC-LC, 1310nm, DR1	P-Rel
	NA	100GE	980-9I53X-00C000	SPQ-CE-ER-CDFL-M	40km 100G QSFP28 ER Optical Transceiver	EOL [P-Rel]

2.2.9.1.5 FDR / 56GbE Cables

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP	FDR	56GE	980-9I679-00L004	MC2207126-004	Mellanox passive copper cable, VPI, up to 56Gb/s, QSFP, 4m	EOL [HVM]
QSFP	FDR	56GE	980-9I67A-00L003	MC2207128-003	Mellanox passive copper cable, VPI, up to 56Gb/s, QSFP, 3m	EOL [HVM]
QSFP	FDR	56GE	980-9I67C-00L02A	MC2207128-0A2	Mellanox passive copper cable, VPI, up to 56Gb/s, QSFP, 2.5m	EOL [MP]
QSFP	FDR	56GE	980-9I67D-00L001	MC2207130-001	Mellanox passive copper cable, VPI, up to 56Gb/s, QSFP, 1m	EOL [HVM]
QSFP	FDR	56GE	980-9I67E-00L002	MC2207130-002	Mellanox passive copper cable, VPI, up to 56Gb/s, QSFP, 2m	EOL [HVM]
QSFP	FDR	56GE	980-9I67F-00L00A	MC2207130-00A	Mellanox passive copper cable, VPI, up to 56Gb/s, QSFP, 0.5m	EOL [HVM]
QSFP	FDR	56GE	980-9I67G-00L01A	MC2207130-0A1	Mellanox passive copper cable, VPI, up to 56Gb/s, QSFP, 1.5m	EOL [HVM]
QSFP	FDR	56GE	980-9I15U-00L003	MC220731V-003	Mellanox active fiber cable, VPI, up to 56Gb/s, QSFP, 3m	EOL [HVM]
QSFP	FDR	56GE	980-9I15V-00L005	MC220731V-005	Mellanox active fiber cable, VPI, up to 56Gb/s, QSFP, 5m	EOL [HVM]
QSFP	FDR	56GE	980-9I15W-00L010	MC220731V-010	Mellanox active fiber cable, VPI, up to 56Gb/s, QSFP, 10m	EOL [HVM]
QSFP	FDR	56GE	980-9I15X-00L015	MC220731V-015	Mellanox active fiber cable, VPI, up to 56Gb/s, QSFP, 15m	EOL [HVM]
QSFP	FDR	56GE	980-9I15Y-00L020	MC220731V-020	Mellanox active fiber cable, VPI, up to 56Gb/s, QSFP, 20m	EOL [HVM]
QSFP	FDR	56GE	980-9I15Z-00L025	MC220731V-025	Mellanox active fiber cable, VPI, up to 56Gb/s, QSFP, 25m	EOL [HVM]
QSFP	FDR	56GE	980-9I150-00L030	MC220731V-030	Mellanox active fiber cable, VPI, up to 56Gb/s, QSFP, 30m	EOL [HVM]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP	FDR	56GE	980-9I152-00L050	MC220731V-050	Mellanox active fiber cable, VPI, up to 56Gb/s, QSFP, 50m	EOL [HVM]
QSFP	FDR	56GE	980-9I154-00L100	MC220731V-100	Mellanox active fiber cable, VPI, up to 56Gb/s, QSFP, 100m	EOL [HVM]

2.2.9.1.6 40GbE Cables

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP	NA	40GE	980-9I666-00B004	MC2210126-004	Mellanox passive copper cable, ETH 40GbE, 40Gb/s, QSFP, 4m	EOL [HVM]
QSFP	NA	40GE	980-9I667-00B005	MC2210126-005	Mellanox passive copper cable, ETH 40GbE, 40Gb/s, QSFP, 5m	EOL [HVM]
QSFP	NA	40GE	980-9I668-00B003	MC2210128-003	Mellanox passive copper cable, ETH 40GbE, 40Gb/s, QSFP, 3m	EOL [HVM]
QSFP	NA	40GE	980-9I66A-00B001	MC2210130-001	Mellanox passive copper cable, ETH 40GbE, 40Gb/s, QSFP, 1m	EOL [HVM]
QSFP	NA	40GE	980-9I66C-00B002	MC2210130-002	Mellanox passive copper cable, ETH 40GbE, 40Gb/s, QSFP, 2m	EOL [HVM]
QSFP	NA	40GE	980-9I14D-00B003	MC2210310-003	Mellanox active fiber cable, ETH 40GbE, 40Gb/s, QSFP, 3m	EOL [MP]
QSFP	NA	40GE	980-9I14E-00B005	MC2210310-005	Mellanox active fiber cable, ETH 40GbE, 40Gb/s, QSFP, 5m	EOL [MP]
QSFP	NA	40GE	980-9I14F-00B010	MC2210310-010	Mellanox active fiber cable, ETH 40GbE, 40Gb/s, QSFP, 10m	EOL [MP]
QSFP	NA	40GE	980-9I14G-00B015	MC2210310-015	Mellanox active fiber cable, ETH 40GbE, 40Gb/s, QSFP, 15m	EOL [MP]
QSFP	NA	40GE	980-9I14H-00B020	MC2210310-020	Mellanox active fiber cable, ETH 40GbE, 40Gb/s, QSFP, 20m	EOL [MP]
QSFP	NA	40GE	980-9I14I-00B030	MC2210310-030	Mellanox active fiber cable, ETH 40GbE, 40Gb/s, QSFP, 30m	EOL [MP]
QSFP	NA	40GE	980-9I14J-00B050	MC2210310-050	Mellanox active fiber cable, ETH 40GbE, 40Gb/s, QSFP, 50m	EOL [MP]
QSFP	NA	40GE	980-9I14K-00B100	MC2210310-100	Mellanox active fiber cable, ETH 40GbE, 40Gb/s, QSFP, 100m	EOL [MP]
Q24S	NA	40GE	980-9I64V-00B005	MC2609125-005	Mellanox passive copper hybrid cable, ETH 40GbE to 4x10GbE, QSFP to 4xSFP+, 5m	EOL [P-Rel]
Q24S	NA	40GE	980-9I64W-00B001	MC2609130-001	Mellanox passive copper hybrid cable, ETH 40GbE to 4x10GbE, QSFP to 4xSFP+, 1m	EOL [HVM]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
Q24S	NA	40GE	980-9I64Y-00B003	MC2609130-003	Mellanox passive copper hybrid cable, ETH 40GbE to 4x10GbE, QSFP to 4xSFP+, 3m	EOL [HVM]
QSFP	NA	40GE	980-9I66U-00B001	MCP1700-B001E	Mellanox passive copper cable, ETH 40GbE, 40Gb/s, QSFP, 1m, Black Pulltab	EOL [HVM]
QSFP	NA	40GE	980-9I66V-00B002	MCP1700-B002E	Mellanox passive copper cable, ETH 40GbE, 40Gb/s, QSFP, 2m, Black Pulltab	EOL [HVM]
QSFP	NA	40GE	980-9I66W-00B003	MCP1700-B003E	Mellanox passive copper cable, ETH 40GbE, 40Gb/s, QSFP, 3m, Black Pulltab	EOL [HVM]
QSFP	NA	40GE	980-9I66X-00B01A	MCP1700-B01AE	Mellanox passive copper cable, ETH 40GbE, 40Gb/s, QSFP, 1.5m, Black Pulltab	EOL [MP]
QSFP	NA	40GE	980-9I66Y-00B02A	MCP1700-B02AE	Mellanox passive copper cable, ETH 40GbE, 40Gb/s, QSFP, 2.5m, Black Pulltab	EOL [MP]
Q24S	NA	40GE	980-9I640-00B002	MCP7904-X002A	Mellanox passive copper hybrid cable, ETH 40GbE to 4x10GbE, QSFP to 4xSFP+, 2m, Black Pulltab, customized label	EOL [HVM]
QSFP	NA	40GE	980-9I426-00BM00	MMA1B00-B150D	Mellanox transceiver, 40GbE, QSFP+, MPO, 850nm, SR4, up to 150m, DDMI	EOL [HVM]

2.2.9.1.7 25GbE Cables

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
Q2S	NA	25GE	980-9I78I-00A000	MAM1Q00A-QSA28	Mellanox cable module, ETH 25GbE, 100Gb/s to 25Gb/s, QSFP28 to SFP28	LTB [HVM]
SFP28	NA	25GE	980-9I63L-00A001	MCP2M00-A001E30N	Mellanox Passive Copper cable, ETH, up to 25Gb/s, SFP28, 1m, Black, 30AWG, CA-N	EOL [HVM]
SFP28	NA	25GE	980-9I63O-00A002	MCP2M00-A002E30N	Mellanox Passive Copper cable, ETH, up to 25Gb/s, SFP28, 2m, Black, 30AWG, CA-N	EOL [HVM]
SFP28	NA	25GE	980-9I63R-00A003	MCP2M00-A003E26N	Mellanox Passive Copper cable, ETH, up to 25Gb/s, SFP28, 3m, Black, 26AWG, CA-N	EOL [HVM]
SFP28	NA	25GE	980-9I63S-00A003	MCP2M00-A003E30L	Mellanox Passive Copper cable, ETH, up to 25Gb/s, SFP28, 3m, Black, 30AWG, CA-L	EOL [HVM]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
SFP+	NA	25GE	980-9I63T-00A004	MCP2M00-A004E26L	Mellanox Passive Copper cable, ETH, up to 25Gb/s, SFP28, 4m, Black, 26AWG, CA-L	EOL [HVM]
SFP+	NA	25GE	980-9I63V-00A005	MCP2M00-A005E26L	Mellanox Passive Copper cable, ETH, up to 25Gb/s, SFP28, 5m, Black, 26AWG, CA-L	EOL [HVM]
SFP28	NA	25GE	980-9I63X-00A00A	MCP2M00-A00AE30N	Mellanox Passive Copper cable, ETH, up to 25Gb/s, SFP28, 0.5m, Black, 30AWG, CA-N	EOL [HVM]
SFP28	NA	25GE	980-9I63Z-00A01A	MCP2M00-A01AE30N	Mellanox Passive Copper cable, ETH, up to 25Gb/s, SFP28, 1.5m, Black, 30AWG, CA-N	EOL [HVM]
SFP28	NA	25GE	980-9I631-00A02A	MCP2M00-A02AE26N	Mellanox Passive Copper cable, ETH, up to 25Gb/s, SFP28, 2.5m, Black, 26AWG, CA-N	EOL [HVM]
SFP28	NA	25GE	980-9I632-00A02A	MCP2M00-A02AE30L	Mellanox Passive Copper cable, ETH, up to 25Gb/s, SFP28, 2.5m, Black, 30AWG, CA-L	EOL [HVM]
SFP28	NA	25GE	980-9IA1T-00A003	MFA2P10-A003	Mellanox active optical cable 25GbE, SFP28, 3m	EOL [HVM]
SFP28	NA	25GE	980-9I53W-00A005	MFA2P10-A005	Mellanox active optical cable 25GbE, SFP28, 5m	EOL [HVM]
SFP28	NA	25GE	980-9I53Z-00A007	MFA2P10-A007	Mellanox active optical cable 25GbE, SFP28, 7m	EOL [HVM]
SFP28	NA	25GE	980-9I532-00A010	MFA2P10-A010	Mellanox active optical cable 25GbE, SFP28, 10m	EOL [HVM]
SFP28	NA	25GE	980-9I535-00A015	MFA2P10-A015	Mellanox active optical cable 25GbE, SFP28, 15m	EOL [HVM]
SFP28	NA	25GE	980-9I536-00A020	MFA2P10-A020	Mellanox active optical cable 25GbE, SFP28, 20m	EOL [HVM]
SFP28	NA	25GE	980-9I539-00A030	MFA2P10-A030	Mellanox active optical cable 25GbE, SFP28, 30m	EOL [HVM]
SFP+	NA	25GE	980-9I094-00AR00	MMA2L20-AR	Mellanox optical transceiver, 25GbE, 25Gb/s, SFP28, LC-LC, 1310nm, LR up to 10km	EOL [MP]
SFP28	NA	25GE	980-9I595-00AM00	MMA2P00-AS	Mellanox transceiver, 25GbE, SFP28, LC-LC, 850nm, SR	HVM

2.2.9.1.8 10GbE Cables

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP to SFP+	NA	10GE	980-9I71G-00J000	MAM1Q00A-QSA	Mellanox cable module, ETH 10GbE, 40Gb/s to 10Gb/s, QSFP to SFP+	LTB [HVM]
QSFP to SFP+	NA	10GE	980-9I65P-00J005	MC2309124-005	Mellanox passive copper hybrid cable, ETH 10GbE, 10Gb/s, QSFP to SFP+, 5m	EOL [P-Rel]
QSFP to SFP+	NA	10GE	980-9I65Q-00J007	MC2309124-007	Mellanox passive copper hybrid cable, ETH 10GbE, 10Gb/s, QSFP to SFP+, 7m	EOL [P-Rel]
QSFP to SFP+	NA	10GE	980-9I65R-00J001	MC2309130-001	Mellanox passive copper hybrid cable, ETH 10GbE, 10Gb/s, QSFP to SFP+, 1m	EOL [HVM]
QSFP to SFP+	NA	10GE	980-9I65S-00J002	MC2309130-002	Mellanox passive copper hybrid cable, ETH 10GbE, 10Gb/s, QSFP to SFP+, 2m	EOL [HVM]
QSFP to SFP+	NA	10GE	980-9I65T-00J003	MC2309130-003	Mellanox passive copper hybrid cable, ETH 10GbE, 10Gb/s, QSFP to SFP+, 3m	EOL [HVM]
SFP+	NA	10GE	980-9I682-00J004	MC3309124-004	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 4m	EOL [HVM]
SFP+	NA	10GE	980-9I683-00J005	MC3309124-005	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 5m	EOL [HVM]
SFP+	NA	10GE	980-9I684-00J006	MC3309124-006	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 6m	EOL [HVM]
SFP+	NA	10GE	980-9I685-00J007	MC3309124-007	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 7m	EOL [HVM]
SFP+	NA	10GE	980-9I686-00J001	MC3309130-001	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 1m	EOL [HVM]
SFP+	NA	10GE	980-9I688-00J002	MC3309130-002	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 2m	EOL [HVM]
SFP+	NA	10GE	980-9I68B-00J003	MC3309130-003	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 3m	EOL [HVM]
SFP+	NA	10GE	980-9I68F-00J00A	MC3309130-00A	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 0.5m	EOL [HVM]
SFP+	NA	10GE	980-9I68G-00J01A	MC3309130-0A1	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 1.5m	EOL [HVM]
SFP+	NA	10GE	980-9I68H-00J02A	MC3309130-0A2	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 2.5m	EOL [HVM]
SFP+	NA	10GE	980-9I68C-00J003	MCP2100-X003B	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 3m, Blue Pulltab, Connector Label	EOL [HVM]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
SFP+	NA	10GE	980-9I68F-00J002	MCP2104-X002B	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 2m, Black Pulltab, Connector Label	EOL [HVM]
SFP+	NA	10GE	980-9I68G-00J003	MCP2104-X003B	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 3m, Black Pulltab, Connector Label	EOL [HVM]
SFP+	NA	10GE	980-9I68I-00J02A	MCP2104-X02AB	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 2.5m, Black Pulltab, Connector Label	EOL [HVM]

2.2.9.1.9 1GbE Cables

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA P/N	Legacy P/N	Description	LifeCycle Phase
SFP	N/A	1GE	980-9I270-00IM00	MC3208011-SX	NVIDIA Optical module, ETH 1GbE, 1Gb/s, SFP, LC-LC, SX 850nm, up to 500m	EOL [P-Rel]
SFP	N/A	1GE	980-9I251-00IS00	MC3208411-T	NVIDIA module, ETH 1GbE, 1Gb/s, SFP, Base-T, up to 100m	HVM

2.2.9.1.10 Supported 3rd Party Cables and Modules

> ⚠ Third-party devices that have not been qualified by NVIDIA may be used; however, please be aware that no performance guarantees are provided.
> Any issues that arise will require initiating a new feature request process for third-party support.

Form Factor	Eth Data Rate	Cable P/N	Description
QSFP+	40GE	606770005	INTEL QSFP 40GBASE CR4
Q22Q	100GE	1002971151	ZQSFP+-TO-ZQSFP+ CABLE ASSEMBLY, 30 AWG, 1.50M LENGTH
Q22Q	100GE	1003461071	ZQSFP+TO 2ZQSFP+50G CBL ASSY 0.7
Q22Q	100GE	1003461076	ZQSFP+-TO-2 ZQSFP+ PASSIVE CABLE ASSEMBLY, 100 GBPS-TO-(2) 50 GBPS
Q22Q	100GE	1003461101	ZQSFP+ TO 2ZQSFP+ 50G CBL
Q22Q	100GE	1003461106	ZQSFP+-TO-2 ZQSFP+ PASSIVE CABLE ASSEMBLY, 100 GBPS-TO-(2) 50 GBPS, LOW-SMOKE ZERO-HALOGEN CABLE, 30 AWG, 1.0M LENGTH

Form Factor	Eth Data Rate	Cable P/N	Description
Q22Q	100GE	10034631 56	ZQSFP+-TO-2 ZQSFP+ PASSIVE CABLE ASSEMBLY, 100 GBPS-TO-(2) 50 GBPS, LOW-SMOKE ZERO-HALOGEN CABLE, 26 AWG, 1.50M LENGTH
Q22Q	100GE	10034633 01	ZQSFP+-TO-2 ZQSFP+ PASSIVE CABLE ASSEMBLY, 100 GBPS-TO-(2) 50 GBPS, 26 AWG, 3.0M LENGTH
QSFP	100GE	10137498-2005LF	HPE 100G 2M COPPER CABLE
QSFP	100GE	10137498-2010LF	PASSIVE COPPER CABLE ETH 100GBE QSFP 1M
QSFP	100GE	10137499-4050LF	PASSIVE COPPER CABLE ETH 100GBE QSFP 5M
QSFP28	100GE	10137628-4050LF	HPE 100GBE QSFP28 DAC 5M CABLE - 100GB/S DIRECT ATTACH COPPER QUAD
QSFP28	100GE	1AT-3Q3Q 9211-01A	100G QSFP28 CWDM4 PN: 1AT-3Q3Q9211-01A TRANSCEIVER MODULE
QSFP28	100GE	1AT-3Q4M 01XX-12A	QSFP28 100G ACTIVE CABLE/MODULE
QSFP112	400GE	400G-DR4	CISCO 400G DR4 QSFP112
QSFP-DD to 2xQSFP56	400GE	AAQD2QP 2400C003	AOI 400G BREAKOUT TO 2X200G BREAKOUT AOC
QSFP+	40GE	AFBR-79E BPZ-HP8	40G BIDIRECTIONAL MMF QSFP+ TRANSCEIVER MODULE
QSFP	40GE	AFBR-7QE R15Z-CS1	CISCO 40GE 15M AOC
QSFP	100GE	AFBR-89C DDZ	TRANSCIEVER 100GBE QSFP SR4
QSFP28	100GE	AFBR-89C DDZ-CS1	AVAGO AFBR-89CDDZ COMPATIBLE 100GBASE-SR4 QSFP28 850NM
QSFP28	100GE	AFBR-89C DDZ-JU1	100G QUADWIRE QSFP28 ACTIVE OPTICAL CABLE
QSFP28	100GE	AFBR-89C EDZ	100GBE QSFP28 PLUGGABLE, PARALLEL FIBER-OPTICS TRANSCEIVER MODULE, EXTENDED REACH 300M
QSFP28	100GE	AQPA9N09 ADLN0817	ACTIVE OPTICAL CABLE 100G QSFP28
QSFP28	100GE	AQPA9N12 ADLN0778	QSFP28 AOC 100G MMF 850NM TRANSCEIVER
QSFP28	100GE	AQPA9N35 ADLN0817	ACTIVE OPTICAL CABLE 100G QSFP28
QSFP28	100GE	AQPMANQ 4EDMA07 84	QSFP28 100G SMF 500M TRANSCEIVER
QSFP28	100GE	AQPMANQ 4EDMA08 71	QSFP28 100G SMF 500M TRANSCEIVER

Form Factor	Eth Data Rate	Cable P/N	Description
QSFP112	400GE	AQQLBCQ4EDLA1729	AOI 400G FR4 QSFP112 MODULE
QSFP56	400GE	ATRF-C020	HGTECH 200G QSFP56 AOC 20M
QSFP28	100GE	ATRQ-A007	QSFP28 AOCS 7M
QSFP28	100GE	ATRQ-A010	HGTECH 100G QSFP28 AOC 10M
QSFP	100GE	CAB-Q-Q-100G-3M	PASSIVE COPPER CABLE ETH 100GBE QSFP 3M
OSFP IHS to RHS	800GE	CAC82X321A1B-C1-MT	CREDO 800G AEC, 2.2M, OSFP IHS TO RHS
QSFP-DD	400GE	C-DQ8FNM005-N00	MELLANOX SELECT 400GBE QSFP-DD AOC 5M
QSFP-DD	400GE	C-DQ8FNM050-N00	MELLANOX SELECT 400GBE QSFP-DD AOC 5M
QSFP-DD to 2xQSFP56	400GE	C-DQF8FNM005-N00	INNOLIGHT 400G QSFP-DD TO 2X200G AOC
QSFP-DD to 2xQSFP56	400GE	C-DQF8FNMxxx-N00	INNOLIGHT 400G QSFP-DD TO 2X200G QSFP56 BREAKOUT AOC
QSFP112 to QSFP-DD	400GE	C-GD4CNS010-N00	INNOLIGHT 400G QSFP112 TO 400G QSFP-DD AOC
OSFP to 2xQSFP112	800GE	C-OSG8CNSxxx-N00	INNOLIGHT 800G DR8 OSFP TO 2X400G QSFP112 DR4 BREAKOUT AOC
QSFP112	400GE	CTF4XFR4CS1-01	INNOLIGHT 400G-FR4 MODULE
QSFP-DD	200GE	DEF8504-2C06-MB3	QSFP-DD ACTIVE OPTICAL CABLE (AOC) TO 2XQSFP-28 ACTIVE OPTICAL CABLE BREAK-OUT
QSFP56	100GE	DHZZjj-KCCC-030	200G QSFP56 TO 2X100G QSFP56 DIRECT ATTACH CABLE
QSFP-DD to 2xQSFP56	200GE	DME8811-EC07	400G-2X200G SPLIT 7M AOC CABLES (400G QSFP-DD BREAKING OUT TO 2X 200G QSFP56
OSFP to 2xQSFP112	800GE	DME8821X-ECxx (rev: 05)	HISENSE 800G OSFP IHS TO 2X400G QSFP112 AOC
QSFP-DD	400GE	DMQ8811A-EC05	QSFP-DD AOC 100M

Form Factor	Eth Data Rate	Cable P/N	Description
QSFP+	40GE	DQF8501-4C01	QSFP+ SR4 4X10.3125GB/S QSFP+ SR4 ACTIVE OPTICAL CABLE
QSFP28	100GE	DQF8503-4C01	4X25.78GB/S QSFP28 ACTIVE OPTICAL CABLE
QSFP28	100GE	DQF8503-4C03	QSFP28 100G ACTIVE OPTICAL CABLE (AOC),3M, MM
QSFP28	100GE	DQF8503-4C05	4X25.78GB/S QSFP28 ACTIVE OPTICAL CABLE
QSFP28	100GE	DQF8503-4C07	7M 100GB/S QSFP28 ACTIVE OPTICAL CABLES AOC DQF8503-4C10 100GBE
QSFP28	100GE	DQF8503-4C10	10M 100GB/S QSFP28 ACTIVE OPTICAL CABLES AOC DQF8503-4C10 100GBE
QSFP28	100GE	DQF8503-4C23	QSFP28 AOCS 7M
QSFP28	100GE	EOLQ-131 HG-O-026	100G FR1
QSFP112	200GE	EOLQ-132 HG-5H-M3	EOPTOLINK 200G QSFP112 DR2 MODULE
QSFP112	400GE	EOLQ-134 HG-5H-M1	EOPTOLINK 400G DR4 QSFP112
QSFP112	400GE	EOLQ-134 HG-5H-M12	EOPTOLINK 400G DR4 QSFP112
QSFP28	100GE	ET7402-SR4	100G QSFP28 OPTICAL TRANSCEIVER
QSFP28	100GE	FCBN425QE1C01	100G QUADWIRE QSFP28 ACTIVE OPTICAL CABLE
QSFP	100GE	FCBN425QE1C10-C1	AOC 100GBE QSFP 1M
QSFP28	10GE---100GE	FCBN425QE1C30-C1	CISCO FINISAR CABLE ASSY QSFP28 M-M 30M
QSFP28	100GE	FCBN425QE2C02	HPE (FINISAR PN FCBN425QE2C02) 100G QUADWIRE QSFP28 ACTIVE OPTICAL CABLE, 2M
QSFP28	100GE	FCBN425QE2C05	4X25G, FULL-DUPLEX, ETHERNET, QSFP CABLE ENDS
QSFP28	100GE	FCBN425QE2C07	100GBASE-AOC QSFP28 TO QSFP28 DIRECT ATTACH CABLE, 7M
QSFP28	100GE	FCBN425QE2C10	100GBASE-AOC QSFP28 TO QSFP28 DIRECT ATTACH CABLE, 10M
QSFP28	100GE	FCBN425QE2C30	HPE (FINISAR PN FCBN425QE2C30) 100G QUADWIRE QSFP28 ACTIVE OPTICAL CABLE, 30M
QSFP28	100GE	FCBN425QF1C01	QSFP28 100G ACTIVE OPTICAL CABLE (AOC), 1M, MM

Form Factor	Eth Data Rate	Cable P/N	Description
QSFP-DD to 2xQSFP56	400GE	FCBN950QE1C05	FINISAR 400G QSFP-DD TO 2X200G QSFP56 BREAKOUT AOC 5M
QSFP-DD to 2xQSFP56	400GE	FCBN950QE1C20	FINISAR 400G QSFP-DD TO 2X200G QSFP56 BREAKOUT AOC 20M
QSFP28	100GE	FCBR425QE1C10-HP	FIBRE OPTIC CABLE ASSEMBLIES 4X25G, FULL-DUPLEX, ETHERNET, QSFP CABLE ENDS
QSFP	100GE	FCBR425QF1C01	CBL ASSY 4X25G ETH QSFP 1M
QSFP	100GE	FCBR425QF1C03	4X25G, FULL-DUPLEX, ETHERNET, QSFP CABLE ENDS
QSFP28	100GE	FOQQD33P00001	QSFP28 CWDM4, SINGLE RATE PULL TAB 100GBE 2KM OPTICAL TRANSCEIVER
	100GE	FOQQD33P00009	CABLE ASSEMBLY, QSFP TO QSFP, OM 3 PLENUM, ACTIVE OPTICAL, 100GBPS, 2M
	100GE	FOQQD33P00010	CABLE ASSEMBLY, QSFP TO QSFP, OM 3 PLENUM, ACTIVE OPTICAL, 100GBPS, 3M
QSFP	40GE	FTL410QD2C-HZ	40BASE-SR4/10GBASE-SR 300M QSFP+ GEN2 OPTICAL TRANSCEIVER MODULE
QSFP		FTL414QB2N-E5	TRANSCIEVER FDR QSFP SR4
QSFP+	40GE	FTL4C1QE1C	40G LR4 QSFP+, UP TO 10KM SINGLE MODE TRANSCEIVER
QSFP28	100GE	FTLC9152RGPL	TRANSCEIVER 100GBE QSFP SWDM4
QSFP28	100GE	FTLC9551REPM	100M PARALLEL MMF 100G QSFP28 OPTICAL TRANSCEIVER
QSFP28	100GE	FTLC9551REPM-H1	QSFP28 ACTIVE OPTICAL CABLE HIGH-SPEED INPUT-OUTPUT CONNECTORS 100G ETHERNET OFNP 1 METER
QSFP28	100GE	FTLC9555REPM3-E6	FIBER OPTIC TRANSMITTERS, RECEIVERS, TRANSCEIVERS XCVR,QSFP28,100M,100GBASE-SR4
QSFP28	100GE	FTLC9555REPM3-HD	FINISAR 100G SR4 MODULE
QSFP	10GE	FTLX8570D3BCL-C2	TRANSCIEVER 10GBE SFP SR
QSFP28	100GE	JNP-QSFP-100G-LR	100GBASE LR4 QSFP28 TRANSCEIVER, LC, 10KM OVER SMF, JNP-QSFP-100G-LR4-LU
QSFP+	40GE	JNP-QSFP-40G-LR4	40GBASE-LR4 QSFP+ 1310NM 10KM LC OVER SMF, JNP-QSFP-40G-LR4-LU
QSFP	40GE	L45593-D118-B50	PASSIVE COPPER CABLE ETH 40GBE QSFP 3M
QSFP28	40GE	L45593-D118-D30	PASSIVE COPPER CABLE ETH 40GBE QSFP 3M

Form Factor	Eth Data Rate	Cable P/N	Description
QSFP28	200GE	L6WQF102-SD-R	3M DAC, 200G TO 2X100G
QSFP	100GE	LQ210CR-CPA2	QSFP28 CWDM4, SINGLE RATE PULL TAB 100GBE 2KM OPTICAL TRANSCEIVER
QSFP56	200GE	MFS1S00-H003V	200G AOC
QSFP56	200GE	MFS1S00-H030V	200G AOC
SFP to BaseT	10GE	MTKP1045M1CNN	TRANSCIEVER SFP TO BASET 10G
QSFP28	100GE	NDAAFF-0003	AMPHENOL 100G, QSFP28 CABLE ASSEMBLY, PASSIVE, 30AWG, 3M, 28G / LANE, JACKET
QSFP28	100GE	NDAAFF-C403	CABLE ASSEMBLY UL 20276 3M 30AWG QSFP+ TO QSFP+ 38 TO 38 POS M-M BAG
QSFP28	100GE	NDAAFJ-C102	CISCO AMPHENOL SF-NDAAFJ100G-005M
QSFP28	100GE	NDAAFJ-M203	QSFP28GB 26AWG, 3METER PASSIVE
QSFP56	200GE	NDAAXG-0002	AMPHENOL 200G QSFP CABLE ASSEMBLY, PASSIVE, 28AWG, 2M, 56G / LANE, JACKET
QSFP28	100GE	NDARHF-M206	QSFP28 TO 2X QSFP28 COPPER SPLITTER CABLE ASSEMBLIES 100G/200G, HIGH SPEED INPUT OUTPUT CONNECTORS, QSFP28GB 30AWG, 2.5METER PASSIVE.
QSFP56	100GE	NDARXJ-B303	3M DAC, 200G TO 2X100G
QSFP+	40GE	NDCCGJ-C402	15M (49FT) AVAGO AFBR-7QER15Z COMPATIBLE 40G QSFP+ ACTIVE OPTICAL CABLE
QSFP-DD to 2xQSFP56	200GE	NDYRFH-0003	AMPHENOL QSFP DD - 2X QSFP CABLE ASSEMBLY, PASSIVE, 27AWG, 3M, 28G / LANE, JACKET
QSFP-DD to 2xQSFP56	200GE	NDYRYF-0001	400G TO 2X200G, QSFP DD - 2X QSFP56 CABLE ASSEMBLY, PASSIVE, 30AWG, 1M
QSFP-DD to 2xQSFP56	200GE	NDYRYF-0006	AMPHENOL 400G TO 2X200G QSFP DD - 2X QSFP CABLE ASSEMBLY, PASSIVE, 30AWG, 0.5M, 56G / LANE, JACKET
QSFP-DD to 2xQSFP56	200GE	NDYRYH-0002	400G TO 2X200G, QSFP DD - 2X QSFP56 CABLE ASSEMBLY, PASSIVE, 27AWG, 2M
QSFP-DD to 2xQSFP56	200GE	NDYRYH-0003	400GGTO 2X200FG QSFP DD - 2X QSFP CABLE ASSEMBLY, PASSIVE, 27AWG, 3M, 56G / LANE, JACKET
QSFP-DD to 4xQSFP56	100GE	NDYSV2-0003	400G TO 4X 100G QSFP DD - 4X QSFP56 CABLE ASSEMBLY, LINEAR ACTIVE, 30AWG, 3M, 56G / LANE, JACKET
QSFP-DD to 4xQSFP56	100GE	NDYSV2-0008	400G TO 4X 100G QSFP DD - 4X QSFP CABLE ASSEMBLY, LINEAR ACTIVE, 30AWG, 2.5M, 56G / LANE, JACKET
QSFP-DD to 4xQSFP56	100GE	NDYSYF-0001	400G TO 4X100G QSFP DD - 4X QSFP56 CABLE ASSEMBLY, PASSIVE, 30AWG, 1M

Form Factor	Eth Data Rate	Cable P/N	Description
QSFP-DD to 4xQSFP28	100GE	NDYSYH-0003	AMPHENOL 400G TO 4X100G, QSFP DD - 4X QSFP CABLE ASSEMBLY, PASSIVE, 27AWG, 3M, 56G / LANE, JACKET
QSFP	40GE	QAOC-40G4F1A25-C	CISCO-DELTA 25M 40GBE AOC
QSFP-DD to 2xQSFP56	400GE	QDD-2Q200-CU3M	CISCO 400G QSFP-DD TO 2X200G BREAKOUT DAC 3M
QSFP-DD	400GE	QDD-400G-SR8	400GBASE-SR8 QSFP-DD PAM4 850NM 100M DOM MTP/MPO-16 MMF OPTICAL TRANSCEIVER MODULE
QSFP-DD to 4xQSFP56	400GE	QDD-4ZQ100-CU1M	CISCO 400G QSFP-DD TO 4X100G BREAKOUT DAC 1M
QSFP28	100GE	QSFP-100G-AOC30M	CISCO 100G QSFP28 AOC 30M
QSFP28	100GE	QSFP-100G-CWDM4	100GBASE CWDM4 QSFP TRANSCEIVER, LC, 2KM OVER SMF, JNP-QSFP-100G-CWDM4-LU
QSFP28	100GE	QSFP-100G-DR-LU	100GBASE DR QSFP TRANSCEIVER, LC, 500M OVER SMF, QSFP-100G-DR-S-LU
QSFP28	100GE	QSFP-100G-SR4-I	100GBASE-SR4 QSFP, MPO, 100M OVER OM4 MMF INDUSTRIAL TEMPERATURE RANGE, QSFP-100G-SR4-I-LU
QSFP28	100GE	QSFP-100G-SR4-LU	100GBASE-SR4 QSFP, QSFP-100G-SR4-LU
QSFP56	200GE	QSFP-200-CU3M	200G QSFP56 TO QSFP56 PASSIVE COPPER CABLE, 3M
QSFP28	100GE	QSFP28-FR-C	QSFP28, FR, 1310NM, 100G, 2KM, SMF, LC, DDM
QSFP28	100GE	QSFP28-LR4-AJ	CISCO 100G LR4 QSFP28 MODULE
QSFP28	100GE	QSFP28-SR4-AJ	QSFP28, SR4, 850NM, 100G, 100M, MMF, MPO12, C-TEMP
QSFP28	100GE	QSFP-40/100-SRBD	TRANSCIEVER 100GBE QSFP BI DIRECTIONAL (BIDIR)
QSFP	40GE	QSFP-40G-SRBD	ARISTA NETWORKS QSFP-40G-SRBD COMPATIBLE 40GBASE-SR BI-DIRECTIONAL QSFP+ OPTICAL TRANSCEIVER MODULE FOR DUPLEX MMF
QSFP	40GE	QSFP-40G-SR-BD	TRANSCIEVER 40GBE QSFP BI DIRECTIONAL (BIDIR)
QSFP	10GE	QSFP-4SFP10G-CU5M	QSFP-4SFP10G-CU5M
QSFP	40GE	QSFP-H40G-CU1M	PASSIVE COPPER CABLE ETH 40GBE QSFP 1M
QSFP28	100GE	QSFP-SR4-AJ	
QSFP56	200GE	R5Z83A	200GB QSFP56 MPO SR4 100M

Form Factor	Eth Data Rate	Cable P/N	Description
QSFP56	200GE	R5Z84A	200GB QSFP56 LC CWDM4 FR4 XCVR
QSFP-DD to 2xQSFP56	200GE	R8M49A	HPE 400GBE TO 2X200G QSFP-DD TO 2XQSFP56 5M ACTIVE OPTICAL CABLE
QSFP-DD to 2xQSFP56	200GE	R8M50A	HPE 400GBE TO 2X200G, QSFP-DD TO 2XQSFP56 15M ACTIVE OPTICAL CABLE
QSFP28	100GE	RTXM420-005	QSFP28 100G
QSFP28	100GE	RTXM420-007	ACCELINK 100G QSFP28 AOCS 7M
QSFP28	100GE	RTXM420-010	QSFP28 AOCS 10M
QSFP28	100GE	RTXM420-550	MPO TYPE 210 M OM3 300 M OM4
QSFP28	100GE	RTXM420-551	100M PARALLEL MMF 100G QSFP28 OPTICAL TRANSCEIVER
QSFP56	200GE	RTXM500-301-F1	ACCELINK 200G QSFP56 SR4
QSFP-DD to 2xQSFP56	200GE	RTXM500-905	400G-2X200G SPLIT 5M AOC CABLES (400G QSFP-DD BREAKING OUT TO 2X 200G QSFP56)
QSFP-DD to 2xQSFP56	400GE	RTXM500-910	ACCELINK 400G QSFP-DD TO 2X200G QSFP56 BREAKOUT AOC 10M
QSFP56 to DSFP	100GE	RTXM520-2xx rev:0B	ACCELINK 100G AOC QSFP56 TO DSFP
QSFP112	400GE	RTXM600-308	ACCELINK 400G QSFP112 SR4
QSFP112	200GE	RTXM600-338-R0	ACCELINK 200G QSFP112 VR2 MODULE
QSFP112	400GE	RTXM600-610	ACCELINK 400G QSFP-DD TO QSFP112 AOC
OSFP to 2xQSFP112	800GE	RTXM600-710	ACCELINK 800G OSFP TO 2X400G QSFP112 BREAKOUT AOC
OSFP to 2xQSFP112	800GE	RTXM600-7xx (rev: R1)	WTD 800G OSFP IHS TO 2X400G QSFP112 AOC
QSFP28	100GE	S2T38A	HPE (FINISAR PN FCBN425QN2C05) 100G QSFP28 5M E-TEMP AOC
QSFP28	100GE	SFBR-89BDDZ-CS2	100G AOM BIDI
QSFP28	100GE	SFBR-89BDDZ-CS4	100G AOM BIDI
QSFP	25GE	SFP-H25G-CU2M	15M (49FT) AVAGO AFBR-7QER15Z COMPATIBLE 40G QSFP+ ACTIVE OPTICAL CABLE

Form Factor	Eth Data Rate	Cable P/N	Description
QSFP	100GE	SO-QSFP28-LR4	TRANSCIEVER 100GBE QSFP LR4
QSFP28	100GE	SPQCEERCDFLM	100G ER QSFP28, UP TO 40KM SINGLE MODE TRANSCEIVER
QSFP28	100GE	SPQCELRCDFB	100G LR4 QSFP28, UP TO 10KM SINGLE MODE TRANSCEIVER
QSFP28	100GE	SPTSBP4LLCDF	QSFP28 100G LR4
QSFP-DD	400GE	SPTSHP2PMCBE	400GBASE-DR4 , 500M
QSFP28	100GE	SPTSLP3SLCDF	100G DR1
QSFP28	100GE	SQF1002L4LNC101P	CISCO-SUMITOMO 100GBE AOM
QSFP28	100GE	TF-FC010-N00	100G OPTICAL TRANSCEIVER QSFP28 PARALLEL ACTIVE OPTICAL CABLE
QSFP56	200GE	T-FX4FNS-N00	INNOLIGHT 200G QSFP56 SR4 MODULE
QSFP112	200GE	T-GP2CNH-NR0	INNOLIGHT 200G QSFP112 DR2 MODULE, MPO-12
QSFP112	400GE	T-GQ4CNT-N00	INNOLIGHT 400G QSFP112 FR4 MODULE, LC
QSFP28	100GE	TR-FC13L-N00	100G QSFP28 OPTICAL TRANSCEIVERS QSFP28 LR4
QSFP112	200GE	TR-HM4M085V-CF21	CREALIGHT 200G QSFP112 VR2 MODULE
QSFP28	100GE	TR-VC13T-N00	INNOLIGHT 100G OPTICAL TRANSCEIVER QSFP28 PSM4 TR-VC13T-N00,UP TO 2KM TRANSMISSI
QSFP28	100GE	TR-ZC13H-NML	100G QSFP28 DR1 TRANSCEIVER
QSFP28	100GE	TR-ZC13T-N00	QSFP28 FR1 (PAM4)

2.2.9.2 BlueField-2 Validated and Supported Cables and Modules

The BlueField-2 firmware version included in this DOCA release supports the following cables.

2.2.9.2.1 Cables Lifecycle Legend

Lifecycle Phase	Definition
EOL	End of Life

Lifecycle Phase	Definition
LTB	Last Time Buy
HVM	GA level
MP	GA level
P-Rel	GA level
Preliminary	Engineering Sample
Prototype	Engineering Sample

2.2.9.2.2 NDR Cables

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
MPO-MPO	NA	NA	980-9I73U-000003	MFP7E10-N003	NVIDIA passive fiber cable, MMF , MPO12 APC to MPO12 APC, 3m	MP
MPO-MPO	NA	NA	980-9I73V-000005	MFP7E10-N005	NVIDIA passive fiber cable, MMF, MPO12 APC to MPO12 APC, 5m	MP
MPO-MPO	NA	NA	980-9I57W-000007	MFP7E10-N007	NVIDIA passive fiber cable, MMF, MPO12 APC to MPO12 APC, 7m	LTB [MP]
MPO-MPO	NA	NA	980-9I57X-00N010	MFP7E10-N010	NVIDIA passive fiber cable, MMF, MPO12 APC to MPO12 APC, 10m	MP
MPO-MPO	NA	NA	980-9I57Y-000015	MFP7E10-N015	NVIDIA passive fiber cable, MMF , MPO12 APC to MPO12 APC, 15m	MP
MPO-MPO	NA	NA	980-9I57Z-000020	MFP7E10-N020	NVIDIA passive fiber cable, MMF, MPO12 APC to MPO12 APC, 20m	MP
MPO-MPO	NDR	NA	980-9I573-00N025	MFP7E10-N025	NVIDIA passive fiber cable, MMF, MPO12 APC to MPO12 APC, 25m	EOL [MP]
MPO-MPO	NA	NA	980-9I570-00N030	MFP7E10-N030	NVIDIA passive fiber cable, MMF, MPO12 APC to MPO12 APC, 30m	MP
MPO-MPO	NA	NA	980-9I570-00N035	MFP7E10-N035	NVIDIA passive fiber cable, MMF, MPO12 APC to MPO12 APC, 35m	LTB [MP]
MPO-MPO	NA	NA	980-9I570-00N040	MFP7E10-N040	NVIDIA passive fiber cable, MMF, MPO12 APC to MPO12 APC, 40m	LTB [MP]
MPO-MPO	NA	NA	980-9I57Y-00N050	MFP7E10-N050	NVIDIA passive fiber cable, MMF, MPO12 APC to MPO12 APC, 50m	MP
MPO-2-2 MPO	NA	NA	980-9I571-00N003	MFP7E20-N003	NVIDIA passive fiber cable, MMF, MPO12 APC to 2xMPO12 APC, 3m	LTB [MP]
MPO-2-2 MPO	NA	NA	980-9I572-00N005	MFP7E20-N005	NVIDIA passive fiber cable, MMF, MPO12 APC to 2xMPO12 APC, 5m	MP
MPO-2-2 MPO	NA	NA	980-9I573-00N007	MFP7E20-N007	NVIDIA passive fiber cable, MMF, MPO12 APC to 2xMPO12 APC, 7m	LTB [MP]
MPO-2-2 MPO	NA	NA	980-9I554-00N010	MFP7E20-N010	NVIDIA passive fiber cable, MMF, MPO12 APC to 2xMPO12 APC, 10m	MP

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
MPO-2-2 MPO	NA	NA	980-9I555-00N015	MFP7E20-N015	NVIDIA passive fiber cable, MMF, MPO12 APC to 2xMPO12 APC, 15m	MP
MPO-2-2 MPO	NA	NA	980-9I556-00N020	MFP7E20-N020	NVIDIA passive fiber cable, MMF, MPO12 APC to 2xMPO12 APC, 20m	LTB [MP]
MPO-2-2 MPO	NA	NA	980-9I557-00N030	MFP7E20-N030	NVIDIA passive fiber cable, MMF, MPO12 APC to 2xMPO12 APC, 30m	MP
MPO-2-2 MPO	NA	NA	980-9I55Z-00N050	MFP7E20-N050	NVIDIA passive fiber cable, MMF, MPO12 APC to 2xMPO12 APC, 50m	MP
MPO-MPO	NDR	NA	980-9I559-00N002	MFP7E30-N002	NVIDIA passive fiber cable, SMF, MPO12 APC to MPO12 APC, 2m	EOL [MP]
MPO-MPO	NA	NA	980-9I55A-00N003	MFP7E30-N003	NVIDIA passive fiber cable, SMF, MPO12 APC to MPO12 APC, 3m	MP
MPO-MPO	NA	NA	980-9I55B-00N005	MFP7E30-N005	NVIDIA passive fiber cable, SMF, MPO12 APC to MPO12 APC, 5m	MP
MPO-MPO	NA	NA	980-9I58C-00N007	MFP7E30-N007	NVIDIA passive fiber cable, SMF, MPO12 APC to MPO12 APC, 7m	LTB [MP]
MPO-MPO	NA	NA	980-9I58D-00N010	MFP7E30-N010	NVIDIA passive fiber cable, SMF, MPO12 APC to MPO12 APC, 10m	MP
MPO-MPO	NA	NA	980-9I58E-00N015	MFP7E30-N015	NVIDIA passive fiber cable, SMF, MPO12 APC to MPO12 APC, 15m	MP
MPO-MPO	NA	NA	980-9I58F-00N020	MFP7E30-N020	NVIDIA passive fiber cable, SMF, MPO12 APC to MPO12 APC, 20m	MP
MPO-MPO	NA	NA	980-9I58G-00N030	MFP7E30-N030	NVIDIA passive fiber cable, SMF, MPO12 APC to MPO12 APC, 30m	MP
MPO-MPO	NA	NA	980-9I580-00N030	MFP7E30-N040	NVIDIA passive fiber cable, SMF, MPO12 APC to MPO12 APC, 40m	LTB [MP]
MPO-MPO	NA	NA	980-9I58H-00N050	MFP7E30-N050	NVIDIA passive fiber cable, SMF, MPO12 APC to MPO12 APC, 50m	MP
MPO-MPO	NA	NA	980-9I581-00N050	MFP7E30-N060	NVIDIA passive fiber cable, SMF, MPO12 APC to MPO12 APC, 60m	LTB [MP]
MPO-MPO	NA	NA	980-9I582-00N050	MFP7E30-N070	NVIDIA passive fiber cable, SMF, MPO12 APC to MPO12 APC, 70m	LTB [MP]
MPO-MPO	NA	NA	980-9I58I-00N100	MFP7E30-N100	NVIDIA passive fiber cable, SMF, MPO12 APC to MPO12 APC, 100m	MP
MPO-MPO	NA	NA	980-9I58J-00N150	MFP7E30-N150	NVIDIA passive fiber cable, SMF, MPO12 APC to MPO12 APC, 150m	MP
MPO-2-2 MPO	NA	NA	980-9I58K-00N003	MFP7E40-N003	NVIDIA passive fiber cable, SMF, MPO12 APC to 2xMPO12 APC, 3m	LTB [MP]
MPO-2-2 MPO	NA	NA	980-9I58L-00N005	MFP7E40-N005	NVIDIA passive fiber cable, SMF, MPO12 APC to 2xMPO12 APC, 5m	MP

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
MPO-2-2 MPO	NA	NA	980-9I58M-00N007	MFP7E40-N007	NVIDIA passive fiber cable, SMF, MPO12 APC to 2xMPO12 APC, 7m	LTB [MP]
MPO-2-2 MPO	NA	NA	980-9I58N-00N010	MFP7E40-N010	NVIDIA passive fiber cable, SMF, MPO12 APC to 2xMPO12 APC, 10m	MP
MPO-2-2 MPO	NA	NA	980-9I56O-00N015	MFP7E40-N015	NVIDIA passive fiber cable, SMF, MPO12 APC to 2xMPO12 APC, 15m	LTB [MP]
MPO-2-2 MPO	NA	NA	980-9I56P-00N020	MFP7E40-N020	NVIDIA passive fiber cable, SMF, MPO12 APC to 2xMPO12 APC, 20m	MP
MPO-2-2 MPO	NA	NA	980-9I56Q-00N030	MFP7E40-N030	NVIDIA passive fiber cable, SMF, MPO12 APC to 2xMPO12 APC, 30m	MP
MPO-2-2 MPO	NA	NA	980-9I56R-000050	MFP7E40-N050	NVIDIA passive fiber cable, SMF, MPO12 APC to 2xMPO12 APC, 50m	MP

2.2.9.2.3 HDR / 200GbE / 400GbE Cables

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP56	HDR	200GE	980-9I548-00H001	MCP1650-H001E30	Nvidia Passive Copper cable, up to 200Gbps, QSFP56 to QSFP56, 1m	HVM
QSFP56	HDR	200GE	980-9I549-00H002	MCP1650-H002E26	Nvidia Passive Copper cable,up to 200Gbps, QSFP56 to QSFP56, 2m	HVM
QSFP56	HDR	200GE	980-9I54A-00H00A	MCP1650-H00AE30	Nvidia Passive Copper cable, up to 200Gbps, QSFP56 to QSFP56, 0.5m	EOL [HVM]
QSFP56	HDR	200GE	980-9I54B-00H01A	MCP1650-H01AE30	Nvidia Passive Copper cable, up to 200Gbps, QSFP56 to QSFP56, 1.5 m	HVM
QSFP56	NA	200GE	980-9I54C-00V001	MCP1650-V001E30	Mellanox Passive Copper cable, 200GbE, 200Gb/s, QSFP56, LSZH, 1m, black pulltab, 30AWG	EOL [HVM]
QSFP56	NA	200GE	980-9I54D-00V002	MCP1650-V002E26	Mellanox Passive Copper cable, 200GbE, 200Gb/s, QSFP56, LSZH, 2m, black pulltab, 26AWG	EOL [HVM]
QSFP56	NA	200GE	980-9I54G-00V003	MCP1650-V003E26	Mellanox Passive Copper cable, 200GbE, 200Gb/s, QSFP56, LSZH, 3m, black pulltab, 26AWG	EOL [HVM]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP56	NA	200GE	980-9I54H-00V00A	MCP1650-V00AE30	Mellanox Passive Copper cable, 200GbE, 200Gb/s, QSFP56, LSZH, 0.5m, black pulltab, 30AWG	EOL [HVM]
QSFP56	NA	200GE	980-9I54I-00V01A	MCP1650-V01AE30	Mellanox Passive Copper cable, 200GbE, 200Gb/s, QSFP56, LSZH, 1.5m, black pulltab, 30AWG	EOL [HVM]
QSFP56	NA	200GE	980-9I54L-00V02A	MCP1650-V02AE26	Mellanox Passive Copper cable, 200GbE, 200Gb/s, QSFP56, LSZH, 2.5m, black pulltab, 26AWG	EOL [HVM]
QSFP56 to 2xQSFP56	HDR	200GE	980-9I39E-00H001	MCP7H50-H001R30	Nvidia Passive copper splitter cable, 200Gbps to 2x100Gbps, QSFP56 to 2xQSFP56, 1m	EOL [HVM]
QSFP56 to 2xQSFP56	HDR	200GE	980-9I99F-00H002	MCP7H50-H002R26	Nvidia Passive copper splitter cable, 200Gbps to 2x100Gbps, QSFP56 to 2xQSFP56, 2m	EOL [HVM]
QSFP56 to 2xQSFP56	HDR	200GE	980-9I98G-00H01A	MCP7H50-H01AR30	Nvidia Passive copper splitter cable, 200Gbps to 2x100Gbps, QSFP56 to 2xQSFP56, 1.5m	EOL [HVM]
QSFP56 to 2xQSFP56	NA	200GE	980-9I98H-00V001	MCP7H50-V001R30	Mellanox passive copper hybrid cable, 200GbE 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, colored, 1m, 30AWG	EOL [HVM]
QSFP56 to 2xQSFP56	NA	200GE	980-9I98I-00V002	MCP7H50-V002R26	Mellanox passive copper hybrid cable, 200GbE 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, colored, 2m, 26AWG	EOL [HVM]
QSFP56 to 2xQSFP56	NA	200GE	980-9I98J-00V003	MCP7H50-V003R26	Mellanox passive copper hybrid cable, 200GbE 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, colored, 3m, 26AWG	EOL [HVM]
QSFP56 to 2xQSFP56	NA	200GE	980-9I98K-00V01A	MCP7H50-V01AR30	Mellanox passive copper hybrid cable, 200GbE 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, colored, 1.5m, 30AWG	EOL [HVM]
QSFP56 to 2xQSFP56	NA	200GE	980-9I98M-00V02A	MCP7H50-V02AR26	Mellanox passive copper hybrid cable, 200GbE 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, colored, 2.5m, 26AWG	EOL [HVM]
QSFP56 to 4xSFP56	NA	200GE	980-9IA3X-00V001	MCP7H70-V001R30	Mellanox passive copper hybrid cable, 200GbE 200Gb/s to 4x50Gb/s, QSFP56 to 4xSFP56, colored, 1m, 30AWG	EOL [P-Rel]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP56 to 4xSFP56	NA	200GE	980-9IA3Y-00V002	MCP7H70-V002R26	Mellanox passive copper hybrid cable, 200GbE 200Gb/s to 4x50Gb/s, QSFP56 to 4xSFP56, colored, 2m, 26AWG	EOL [P-Rel]
QSFP56 to 4xSFP56	NA	200GE	980-9I43Z-00V003	MCP7H70-V003R26	Mellanox passive copper hybrid cable, 200GbE 200Gb/s to 4x50Gb/s, QSFP56 to 4x4SFP56, colored, 3m, 26AWG	EOL [P-Rel]
QSFP56 to 4xSFP56	NA	200GE	980-9I430-00V01A	MCP7H70-V01AR30	Mellanox passive copper hybrid cable, 200GbE 200Gb/s to 4x50Gb/s, QSFP56 to 4xSFP56, colored, 1.5m, 30AWG	EOL [P-Rel]
QSFP56 to 4xSFP56	NA	200GE	980-9I431-00V02A	MCP7H70-V02AR26	Mellanox passive copper hybrid cable, 200GbE 200Gb/s to4x50Gb/s, QSFP56 to 4xSFP56, colored, 2.5m, 26AWG	EOL [P-Rel]
QSFP56	HDR	200GE	980-9I457-00H003	MFS1S00-H003V	Nvidia active optical cable, up to 200Gbps , QSFP56 to QSFP56, 3m	EOL [MP]
QSFP56	HDR	200GE	980-9I45D-00H005	MFS1S00-H005V	Nvidia active optical cable, up to 200Gbps , QSFP56 to QSFP56, 5m	MP
QSFP56	HDR	200GE	980-9I45J-00H010	MFS1S00-H010V	Nvidia active optical cable, up to 200Gbps , QSFP56 to QSFP56, 10m	MP
QSFP56	HDR	200GE	980-9I45O-00H015	MFS1S00-H015V	Nvidia active optical cable, up to 200Gbps , QSFP56 to QSFP56, 15m	EOL [MP]
QSFP56	HDR	200GE	980-9I45T-00H020	MFS1S00-H020V	Nvidia active optical cable, up to 200Gbps , QSFP56 to QSFP56, 20m	MP
QSFP56	HDR	200GE	980-9I440-00H030	MFS1S00-H030V	Nvidia active optical cable, up to 200Gbps , QSFP56 to QSFP56, 30m	MP
QSFP56	HDR	200GE	980-9I447-00H050	MFS1S00-H050V	Nvidia active optical cable, up to 200Gbps , QSFP56 to QSFP56, 50m	MP
QSFP56	HDR	200GE	980-9I44H-00H100	MFS1S00-H100V	Nvidia active optical cable, up to 200Gbps , QSFP56 to QSFP56, 100m	MP
QSFP56	HDR	200GE	980-9I44K-00H130	MFS1S00-H130V	Nvidia active optical cable, up to 200Gbps , QSFP56 to QSFP56, 130m	EOL [MP]
QSFP56	NA	200GE	980-9I44P-00V003	MFS1S00-V003E	Mellanox active fiber cable, 200GbE, 200Gb/s, QSFP56, LSZH, black pulltab, 3m	EOL [HVM]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP56	NA	200GE	980-9I45Q-00V005	MFS1S00-V005E	Mellanox active fiber cable, 200GbE, 200Gb/s, QSFP56, LSZH, black pulltab, 5m	EOL [HVM]
QSFP56	NA	200GE	980-9I45R-00V010	MFS1S00-V010E	Mellanox active fiber cable, 200GbE, 200Gb/s, QSFP56, LSZH, black pulltab, 10m	EOL [HVM]
QSFP56	NA	200GE	980-9I44S-00V015	MFS1S00-V015E	Mellanox active fiber cable, 200GbE, 200Gb/s, QSFP56, LSZH, black pulltab, 15m	EOL [HVM]
QSFP56	NA	200GE	980-9I44T-00V020	MFS1S00-V020E	Mellanox active fiber cable, 200GbE, 200Gb/s, QSFP56, LSZH, black pulltab, 20m	EOL [HVM]
QSFP56	NA	200GE	980-9I44U-00V030	MFS1S00-V030E	Mellanox active fiber cable, 200GbE, 200Gb/s, QSFP56, LSZH, black pulltab, 30m	EOL [HVM]
QSFP56	NA	200GE	980-9I44V-00V050	MFS1S00-V050E	Mellanox active fiber cable, 200GbE, 200Gb/s, QSFP56, LSZH, black pulltab, 50m	EOL [HVM]
QSFP56	NA	200GE	980-9I44W-00V100	MFS1S00-V100E	Mellanox active fiber cable, 200GbE, 200Gb/s, QSFP56, LSZH, black pulltab, 100m	EOL [HVM] [HIBERNATE]
QSFP56 to 2xQSFP56	HDR	200GE	980-9I445-00H003	MFS1S50-H003V	Nvidia active optical splitter cable, 200Gbps to 2x100Gbps , QSFP56 to 2x QSFP56, 3m	EOL [HVM]
QSFP56 to 2xQSFP56	HDR	200GE	980-9I969-00H005	MFS1S50-H005V	Nvidia active optical splitter cable, 200Gbps to 2x100Gbps , QSFP56 to 2x QSFP56, 5m	EOL [HVM]
QSFP56 to 2xQSFP56	HDR	200GE	980-9I96D-00H010	MFS1S50-H010V	Nvidia active optical splitter cable, 200Gbps to 2x100Gbps , QSFP56 to 2x QSFP56, 10m	EOL [HVM]
QSFP56 to 2xQSFP56	HDR	200GE	980-9I96H-00H015	MFS1S50-H015V	Nvidia active optical splitter cable, 200Gbps to 2x100Gbps , QSFP56 to 2x QSFP56, 15m	EOL [HVM]
QSFP56 to 2xQSFP56	HDR	200GE	980-9I96L-00H020	MFS1S50-H020V	Nvidia active optical splitter cable, 200Gbps to 2x100Gbps , QSFP56 to 2x QSFP56, 20m	EOL [HVM]
QSFP56 to 2xQSFP56	HDR	200GE	980-9I96P-00H030	MFS1S50-H030V	Nvidia active optical splitter cable, 200Gbps to 2x100Gbps , QSFP56 to 2x QSFP56, 30m	EOL [HVM]
QSFP56 to 2xQSFP56	NA	200GE	980-9I95Q-00V003	MFS1S50-V003E	Mellanox active fiber splitter cable, 200GbE, 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, LSZH, black pulltab, 3m	EOL [HVM]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP56 to 2xQSFP56	NA	200GE	980-9I96R-00V005	MFS1S50-V005E	Mellanox active fiber splitter cable, 200GbE, 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, LSZH, black pulltab, 5m	EOL [HVM]
QSFP56 to 2xQSFP56	NA	200GE	980-9I96S-00V010	MFS1S50-V010E	Mellanox active fiber splitter cable, 200GbE, 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, LSZH, black pulltab, 10m	EOL [HVM]
QSFP56 to 2xQSFP56	NA	200GE	980-9I96T-00V015	MFS1S50-V015E	Mellanox active fiber splitter cable, 200GbE, 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, LSZH, black pulltab, 15m	EOL [HVM]
QSFP56 to 2xQSFP56	NA	200GE	980-9I95U-00V020	MFS1S50-V020E	Mellanox active fiber splitter cable, 200GbE, 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, LSZH, black pulltab, 20m	EOL [HVM]
QSFP56 to 2xQSFP56	NA	200GE	980-9I95V-00V030	MFS1S50-V030E	Mellanox active fiber splitter cable, 200GbE, 200Gb/s to 2x100Gb/s, QSFP56 to 2xQSFP56, LSZH, black pulltab, 30m	EOL [HVM]
QSFP56	NA	200GE	980-9I20T-00V000	MMA1T00-VS	Mellanox transceiver, 200GbE, up to 200Gb/s, QSFP56, MPO, 850nm, SR4, up to 100m	EOL [HVM]
OSFP to 2xQSFP56	HDR	400GE	980-9I46K-00H001	MCP7Y60-H001	NVIDIA passive copper splitter cable, 400(2x200)Gbps to 2x200Gbps, OSFP to 2xQSFP56, 1m, fin to flat	EOL [MP]
OSFP to 2xQSFP56	HDR	400GE	980-9I46L-00H002	MCP7Y60-H002	NVIDIA passive copper splitter cable, 400(2x200)Gbps to 2x200Gbps, OSFP to 2xQSFP56, 2m, fin to flat	EOL [MP]
OSFP to 2xQSFP56	HDR	400GE	980-9I93M-00H01A	MCP7Y60-H01A	NVIDIA passive copper splitter cable, 400(2x200)Gbps to 2x200Gbps, OSFP to 2xQSFP56, 1.5m, fin to flat	EOL [MP]
OSFP to 4xQSFP56	HDR	400GE	980-9I93N-00H001	MCP7Y70-H001	NVIDIA passive copper splitter cable, 400(2x200)Gbps to 4x100Gbps, OSFP to 4xQSFP56, 1m, fin to flat	EOL [MP]
OSFP to 4xQSFP56	HDR	400GE	980-9I93O-00H002	MCP7Y70-H002	NVIDIA passive copper splitter cable, 400(2x200)Gbps to 4x100Gbps, OSFP to 4xQSFP56, 2m, fin to flat	EOL [MP]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
OSFP to 4xQSFP56	HDR	400GE	980-9I47P-00H01A	MCP7Y70-H01A	NVIDIA passive copper splitter cable, 400(2x200)Gbps to 4x100Gbps, OSFP to 4xQSFP56, 1.5m, fin to flat	EOL [MP]

2.2.9.2.4 EDR / 100GbE Cables

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP28	NA	100GE	980-9I620-00C001	MCP1600-C001E30N	Mellanox Passive Copper cable, ETH 100GbE, 100Gb/s, QSFP28, 1m, Black, 30AWG, CA-N	EOL [HVM]
QSFP28	NA	100GE	980-9I62V-00C002	MCP1600-C002E30N	Mellanox Passive Copper cable, ETH 100GbE, 100Gb/s, QSFP28, 2m, Black, 30AWG, CA-N	EOL [HVM]
QSFP28	NA	100GE	980-9I62Z-00C003	MCP1600-C003E26N	Mellanox Passive Copper cable, ETH 100GbE, 100Gb/s, QSFP28, 3m, Black, 26AWG, CA-N	EOL [HVM]
QSFP28	NA	100GE	980-9I620-00C003	MCP1600-C003E30L	Mellanox Passive Copper cable, ETH 100GbE, 100Gb/s, QSFP28, 3m, Black, 30AWG, CA-L	EOL [HVM]
QSFP	NA	100GE	980-9I625-00C005	MCP1600-C005E26L	Mellanox Passive Copper cable, ETH 100GbE, 100Gb/s, QSFP28, 5m, Black, 26AWG, CA-L	EOL [HVM]
QSFP28	NA	100GE	980-9I627-00C00A	MCP1600-C00AE30N	Mellanox Passive Copper cable, ETH 100GbE, 100Gb/s, QSFP28, 0.5m, Black, 30AWG, CA-N	EOL [HVM]
QSFP28	NA	100GE	980-9I629-00C00B	MCP1600-C00BE30N	Mellanox Passive Copper cable, ETH 100GbE, 100Gb/s, QSFP28, 0.75m, Black, 30AWG, CA-N	EOL [HVM]
QSFP28	NA	100GE	980-9I62C-00C01A	MCP1600-C01AE30N	Mellanox Passive Copper cable, ETH 100GbE, 100Gb/s, QSFP28, 1.5m, Black, 30AWG, CA-N	EOL [HVM]
QSFP28	NA	100GE	980-9I62H-00C02A	MCP1600-C02AE26N	Mellanox Passive Copper cable, ETH 100GbE, 100Gb/s, QSFP28, 2.5m, Black, 26AWG, CA-N	EOL [HVM]
QSFP28	NA	100GE	980-9I62I-00C02A	MCP1600-C02AE30L	Mellanox Passive Copper cable, ETH 100GbE, 100Gb/s, QSFP28,2.5m, Black, 30AWG, CA-L	EOL [HVM]
QSFP	EDR	100GE	980-9I62V-00C003	MCP1600-E003	Mellanox Passive Copper cable, IB EDR, up to 100Gb/s, QSFP, LSZH, 3m 26AWG	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I486-00C001	MCP7F00-A001R30N	Mellanox passive copper hybrid cable, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 1m, Colored, 30AWG, CA-N	EOL [HVM]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP28 to 4xSFP28	NA	100GE	980-9I48B-00C002	MCP7F00-A002R30N	Mellanox passive copper hybrid cable, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 2m, Colored, 30AWG, CA-N	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I48G-00C003	MCP7F00-A003R26N	Mellanox passive copper hybrid cable, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 3m, Colored, 26AWG, CA-N	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I48H-00C003	MCP7F00-A003R30L	Mellanox passive copper hybrid cable, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 3m, Colored, 30AWG, CA-L	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I48J-00C005	MCP7F00-A005R26L	Mellanox passive copper hybrid cable, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 5m, Colored, 26AWG, CA-L	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I48N-00C01A	MCP7F00-A01AR30N	Mellanox passive copper hybrid cable, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 1.5m, Colored, 30AWG, CA-N	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I48S-00C02A	MCP7F00-A02AR26N	Mellanox passive copper hybrid cable, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 2.5m, Colored, 26AWG, CA-N	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I48T-00C02A	MCP7F00-A02AR30L	Mellanox passive copper hybrid cable, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 2.5m, Colored, 30AWG, CA-L	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I48X-00C03A	MCP7F00-A03AR26L	Mellanox passive copper hybrid cable, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 3.5m, Colored, 26AWG, CA-L	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I99G-00C001	MCP7H00-G001R30N	Mellanox passive copper hybrid cable, ETH 100Gb/s to 2x50Gb/s, QSFP28 to 2xQSFP28, 1m, Colored, 30AWG, CA-N	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I99L-00C002	MCP7H00-G002R30N	Mellanox passive copper hybrid cable, ETH 100Gb/s to 2x50Gb/s, QSFP28 to 2xQSFP28, 2m, Colored, 30AWG, CA-N	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I99Q-00C003	MCP7H00-G003R26N	Mellanox passive copper hybrid cable, ETH 100Gb/s to 2x50Gb/s, QSFP28 to 2xQSFP28, 3m, Colored, 26AWG, CA-N	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I39R-00C003	MCP7H00-G003R30L	Mellanox passive copper hybrid cable, ETH 100Gb/s to 2x50Gb/s, QSFP28 to 2xQSFP28, 3m, Colored, 30AWG, CA-L	EOL [HVM]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP28 to 2xQSFP28	NA	100GE	980-9I99S-00C004	MCP7H00-G004R26L	Mellanox passive copper hybrid cable, ETH 100Gb/s to 2x50Gb/s, QSFP28 to 2xQSFP28, 4m, Colored, 26AWG, CA-L	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I99X-00C01A	MCP7H00-G01AR30N	Mellanox passive copper hybrid cable, ETH 100Gb/s to 2x50Gb/s, QSFP28 to 2xQSFP28, 1.5m, Colored, 30AWG, CA-N	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I395-00C02A	MCP7H00-G02AR30L	Mellanox passive copper hybrid cable, ETH 100Gb/s to 2x50Gb/s, QSFP28 to 2xQSFP28, 2.5m, Colored, 30AWG, CA-L	EOL [HVM]
QSFP28	NA	100GE	980-9I13S-00C003	MFA1A00-C003	Mellanox active fiber cable, ETH 100GbE, 100Gb/s, QSFP, LSZH, 3m	EOL [HVM]
QSFP28	NA	100GE	980-9I13X-00C005	MFA1A00-C005	Mellanox active fiber cable, ETH 100GbE, 100Gb/s, QSFP, LSZH, 5m	EOL [HVM]
QSFP28	NA	100GE	980-9I134-00C010	MFA1A00-C010	Mellanox active fiber cable, ETH 100GbE, 100Gb/s, QSFP, LSZH, 10m	EOL [HVM]
QSFP28	NA	100GE	980-9I13A-00C015	MFA1A00-C015	Mellanox active fiber cable, ETH 100GbE, 100Gb/s, QSFP, LSZH, 15m	EOL [HVM]
QSFP28	NA	100GE	980-9I13F-00C020	MFA1A00-C020	Mellanox active fiber cable, ETH 100GbE, 100Gb/s, QSFP, LSZH, 20m	EOL [HVM]
QSFP28	NA	100GE	980-9I13N-00C030	MFA1A00-C030	Mellanox active fiber cable, ETH 100GbE, 100Gb/s, QSFP, LSZH, 30m	EOL [HVM]
QSFP28	NA	100GE	980-9I130-00C050	MFA1A00-C050	Mellanox active fiber cable, ETH 100GbE, 100Gb/s, QSFP, LSZH, 50m	EOL [HVM]
QSFP28	NA	100GE	980-9I13B-00C100	MFA1A00-C100	Mellanox active fiber cable, ETH 100GbE, 100Gb/s, QSFP, LSZH, 100m	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I37H-00C003	MFA7A20-C003	Mellanox active fiber hybrid solution, ETH 100GbE to 2x50GbE, QSFP28 to 2xQSFP28, 3m	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I37I-00C005	MFA7A20-C005	Mellanox active fiber hybrid solution, ETH 100GbE to 2x50GbE, QSFP28 to 2xQSFP28, 5m	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I40J-00C010	MFA7A20-C010	Mellanox active fiber hybrid solution, ETH 100GbE to 2x50GbE, QSFP28 to 2xQSFP28, 10m	EOL [HVM]
QSFP28 to 2xQSFP28	NA	100GE	980-9I40K-00C020	MFA7A20-C020	Mellanox active fiber hybrid solution, ETH 100GbE to 2x50GbE, QSFP28 to 2xQSFP28, 20m	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I40N-00C003	MFA7A50-C003	Mellanox active fiber hybrid solution, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 3m	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I40O-00C005	MFA7A50-C005	Mellanox active fiber hybrid solution, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 5m	EOL [HVM]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP28 to 4xSFP28	NA	100GE	980-9I49P-00C010	MFA7A50-C010	Mellanox active fiber hybrid solution, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 10m	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I49Q-00C015	MFA7A50-C015	Mellanox active fiber hybrid solution, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 15m	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I49R-00C020	MFA7A50-C020	Mellanox active fiber hybrid solution, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 20m	EOL [HVM]
QSFP28 to 4xSFP28	NA	100GE	980-9I49S-00C030	MFA7A50-C030	Mellanox active fiber hybrid solution, ETH 100GbE to 4x25GbE, QSFP28 to 4xSFP28, 30m	EOL [HVM]
QSFP28	NA	100GE	980-9I149-00CS00	MMA1B00-C100D	Mellanox transceiver, 100GbE, QSFP28, MPO, 850nm, SR4, up to 100m, DDMI	HVM
QSFP	NA	100GE	980-9I17P-00CR00	MMA1L10-CR	Mellanox optical transceiver, 100GbE, 100Gb/s, QSFP28, LC-LC, 1310nm, LR4 up to 10km	LTB [HVM]
QSFP	NA	100GE	980-9I17Q-00CM00	MMA1L30-CM	Mellanox optical module, 100GbE, 100Gb/s, QSFP28, LC-LC, 1310nm, CWDM4, up to 2km	LTB [MP]
QSFP	NA	100GE	980-9I16X-00C000	MMS1C10-CM	Mellanox active optical module, 100Gb/s, QSFP, MPO, 1310nm, PSM4, up to 500m	EOL [MP]
QSFP28	NA	100GE	980-9I042-00C000	MMS1V70-CM	Mellanox transceiver, 100GbE, QSFP28, LC-LC, 1310nm, DR1	P-Rel
	NA	100GE	980-9I53X-00C000	SPQ-CE-ER-CDFL-M	40km 100G QSFP28 ER Optical Transceiver	EOL [P-Rel]

2.2.9.2.5 FDR / 56GbE Cables

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP	FDR	56GE	980-9I679-00L004	MC2207126-004	Mellanox passive copper cable, VPI, up to 56Gb/s, QSFP, 4m	EOL [HVM]
QSFP	FDR	56GE	980-9I67A-00L003	MC2207128-003	Mellanox passive copper cable, VPI, up to 56Gb/s, QSFP, 3m	EOL [HVM]
QSFP	FDR	56GE	980-9I67C-00L02A	MC2207128-0A2	Mellanox passive copper cable, VPI, up to 56Gb/s, QSFP, 2.5m	EOL [MP]
QSFP	FDR	56GE	980-9I67D-00L001	MC2207130-001	Mellanox passive copper cable, VPI, up to 56Gb/s, QSFP, 1m	EOL [HVM]
QSFP	FDR	56GE	980-9I67E-00L002	MC2207130-002	Mellanox passive copper cable, VPI, up to 56Gb/s, QSFP, 2m	EOL [HVM]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP	FDR	56GE	980-9I67F-00L00A	MC2207130-00A	Mellanox passive copper cable, VPI, up to 56Gb/s, QSFP, 0.5m	EOL [HVM]
QSFP	FDR	56GE	980-9I67G-00L01A	MC2207130-0A1	Mellanox passive copper cable, VPI, up to 56Gb/s, QSFP, 1.5m	EOL [HVM]
QSFP	FDR	56GE	980-9I15U-00L003	MC220731V-003	Mellanox active fiber cable, VPI, up to 56Gb/s, QSFP, 3m	EOL [HVM]
QSFP	FDR	56GE	980-9I15V-00L005	MC220731V-005	Mellanox active fiber cable, VPI, up to 56Gb/s, QSFP, 5m	EOL [HVM]
QSFP	FDR	56GE	980-9I15W-00L010	MC220731V-010	Mellanox active fiber cable, VPI, up to 56Gb/s, QSFP, 10m	EOL [HVM]
QSFP	FDR	56GE	980-9I15X-00L015	MC220731V-015	Mellanox active fiber cable, VPI, up to 56Gb/s, QSFP, 15m	EOL [HVM]
QSFP	FDR	56GE	980-9I15Y-00L020	MC220731V-020	Mellanox active fiber cable, VPI, up to 56Gb/s, QSFP, 20m	EOL [HVM]
QSFP	FDR	56GE	980-9I15Z-00L025	MC220731V-025	Mellanox active fiber cable, VPI, up to 56Gb/s, QSFP, 25m	EOL [HVM]
QSFP	FDR	56GE	980-9I150-00L030	MC220731V-030	Mellanox active fiber cable, VPI, up to 56Gb/s, QSFP, 30m	EOL [HVM]
QSFP	FDR	56GE	980-9I152-00L050	MC220731V-050	Mellanox active fiber cable, VPI, up to 56Gb/s, QSFP, 50m	EOL [HVM]
QSFP	FDR	56GE	980-9I154-00L100	MC220731V-100	Mellanox active fiber cable, VPI, up to 56Gb/s, QSFP, 100m	EOL [HVM]
QSFP	FDR	NA	980-9I17M-00FS00	MMA1B00-F030D	Mellanox transceiver, FDR, QSFP+, MPO, 850nm, SR4, up to 30m, DDMI	EOL [HVM]

2.2.9.2.6 40GbE Cables

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP	NA	40GE	980-9I666-00B004	MC2210126-004	Mellanox passive copper cable, ETH 40GbE, 40Gb/s, QSFP, 4m	EOL [HVM]
QSFP	NA	40GE	980-9I667-00B005	MC2210126-005	Mellanox passive copper cable, ETH 40GbE, 40Gb/s, QSFP, 5m	EOL [HVM]
QSFP	NA	40GE	980-9I668-00B003	MC2210128-003	Mellanox passive copper cable, ETH 40GbE, 40Gb/s, QSFP, 3m	EOL [HVM]
QSFP	NA	40GE	980-9I66A-00B001	MC2210130-001	Mellanox passive copper cable, ETH 40GbE, 40Gb/s, QSFP, 1m	EOL [HVM]
QSFP	NA	40GE	980-9I66C-00B002	MC2210130-002	Mellanox passive copper cable, ETH 40GbE, 40Gb/s, QSFP, 2m	EOL [HVM]
QSFP	NA	40GE	980-9I14D-00B003	MC2210310-003	Mellanox active fiber cable, ETH 40GbE, 40Gb/s, QSFP, 3m	EOL [MP]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP	NA	40GE	980-9I14E-00B005	MC2210310-005	Mellanox active fiber cable, ETH 40GbE, 40Gb/s, QSFP, 5m	EOL [MP]
QSFP	NA	40GE	980-9I14F-00B010	MC2210310-010	Mellanox active fiber cable, ETH 40GbE, 40Gb/s, QSFP, 10m	EOL [MP]
QSFP	NA	40GE	980-9I14G-00B015	MC2210310-015	Mellanox active fiber cable, ETH 40GbE, 40Gb/s, QSFP, 15m	EOL [MP]
QSFP	NA	40GE	980-9I14H-00B020	MC2210310-020	Mellanox active fiber cable, ETH 40GbE, 40Gb/s, QSFP, 20m	EOL [MP]
QSFP	NA	40GE	980-9I14I-00B030	MC2210310-030	Mellanox active fiber cable, ETH 40GbE, 40Gb/s, QSFP, 30m	EOL [MP]
QSFP	NA	40GE	980-9I14J-00B050	MC2210310-050	Mellanox active fiber cable, ETH 40GbE, 40Gb/s, QSFP, 50m	EOL [MP]
QSFP	NA	40GE	980-9I14K-00B100	MC2210310-100	Mellanox active fiber cable, ETH 40GbE, 40Gb/s, QSFP, 100m	EOL [MP]
Q24S	NA	40GE	980-9I64V-00B005	MC2609125-005	Mellanox passive copper hybrid cable, ETH 40GbE to 4x10GbE, QSFP to 4xSFP+, 5m	EOL [P-Rel]
Q24S	NA	40GE	980-9I64W-00B001	MC2609130-001	Mellanox passive copper hybrid cable, ETH 40GbE to 4x10GbE, QSFP to 4xSFP+, 1m	EOL [HVM]
Q24S	NA	40GE	980-9I64Y-00B003	MC2609130-003	Mellanox passive copper hybrid cable, ETH 40GbE to 4x10GbE, QSFP to 4xSFP+, 3m	EOL [HVM]
QSFP	NA	40GE	980-9I66U-00B001	MCP1700-B001E	Mellanox passive copper cable, ETH 40GbE, 40Gb/s, QSFP, 1m, Black Pulltab	EOL [HVM]
QSFP	NA	40GE	980-9I66V-00B002	MCP1700-B002E	Mellanox passive copper cable, ETH 40GbE, 40Gb/s, QSFP, 2m, Black Pulltab	EOL [HVM]
QSFP	NA	40GE	980-9I66W-00B003	MCP1700-B003E	Mellanox passive copper cable, ETH 40GbE, 40Gb/s, QSFP, 3m, Black Pulltab	EOL [HVM]
QSFP	NA	40GE	980-9I66X-00B01A	MCP1700-B01AE	Mellanox passive copper cable, ETH 40GbE, 40Gb/s, QSFP, 1.5m, Black Pulltab	EOL [MP]
QSFP	NA	40GE	980-9I66Y-00B02A	MCP1700-B02AE	Mellanox passive copper cable, ETH 40GbE, 40Gb/s, QSFP, 2.5m, Black Pulltab	EOL [MP]
Q24S	NA	40GE	980-9I640-00B002	MCP7904-X002A	Mellanox passive copper hybrid cable, ETH 40GbE to 4x10GbE, QSFP to 4xSFP+, 2m, Black Pulltab, customized label	EOL [HVM]
QSFP	NA	40GE	980-9I426-00BM00	MMA1B00-B150D	Mellanox transceiver, 40GbE, QSFP+, MPO, 850nm, SR4, up to 150m, DDMI	EOL [HVM]

2.2.9.2.7 25GbE Cables

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
QSFP28 to SFP28	NA	25GE	980-9I78I-00A000	MAM1Q00A-QSA28	Mellanox cable module, ETH 25GbE, 100Gb/s to 25Gb/s, QSFP28 to SFP28	LTB [HVM]
SFP28	NA	25GE	980-9I63L-00A001	MCP2M00-A001E30N	Mellanox Passive Copper cable, ETH, up to 25Gb/s, SFP28, 1m, Black, 30AWG, CA-N	EOL [HVM]
SFP28	NA	25GE	980-9I63O-00A002	MCP2M00-A002E30N	Mellanox Passive Copper cable, ETH, up to 25Gb/s, SFP28, 2m, Black, 30AWG, CA-N	EOL [HVM]
SFP28	NA	25GE	980-9I63R-00A003	MCP2M00-A003E26N	Mellanox Passive Copper cable, ETH, up to 25Gb/s, SFP28, 3m, Black, 26AWG, CA-N	EOL [HVM]
SFP28	NA	25GE	980-9I63S-00A003	MCP2M00-A003E30L	Mellanox Passive Copper cable, ETH, up to 25Gb/s, SFP28, 3m, Black, 30AWG, CA-L	EOL [HVM]
SFP+	NA	25GE	980-9I63T-00A004	MCP2M00-A004E26L	Mellanox Passive Copper cable, ETH, up to 25Gb/s, SFP28, 4m, Black, 26AWG, CA-L	EOL [HVM]
SFP+	NA	25GE	980-9I63V-00A005	MCP2M00-A005E26L	Mellanox Passive Copper cable, ETH, up to 25Gb/s, SFP28, 5m, Black, 26AWG, CA-L	EOL [HVM]
SFP28	NA	25GE	980-9I63X-00A00A	MCP2M00-A00AE30N	Mellanox Passive Copper cable, ETH, up to 25Gb/s, SFP28, 0.5m, Black, 30AWG, CA-N	EOL [HVM]
SFP28	NA	25GE	980-9I63Z-00A01A	MCP2M00-A01AE30N	Mellanox Passive Copper cable, ETH, up to 25Gb/s, SFP28, 1.5m, Black, 30AWG, CA-N	EOL [HVM]
SFP28	NA	25GE	980-9I631-00A02A	MCP2M00-A02AE26N	Mellanox Passive Copper cable, ETH, up to 25Gb/s, SFP28, 2.5m, Black, 26AWG, CA-N	EOL [HVM]
SFP28	NA	25GE	980-9I632-00A02A	MCP2M00-A02AE30L	Mellanox Passive Copper cable, ETH, up to 25Gb/s, SFP28, 2.5m, Black, 30AWG, CA-L	EOL [HVM]
SFP28	NA	25GE	980-9IA1T-00A003	MFA2P10-A003	Mellanox active optical cable 25GbE, SFP28, 3m	EOL [HVM]
SFP28	NA	25GE	980-9I53W-00A005	MFA2P10-A005	Mellanox active optical cable 25GbE, SFP28, 5m	EOL [HVM]
SFP28	NA	25GE	980-9I53Z-00A007	MFA2P10-A007	Mellanox active optical cable 25GbE, SFP28, 7m	EOL [HVM]
SFP28	NA	25GE	980-9I532-00A010	MFA2P10-A010	Mellanox active optical cable 25GbE, SFP28, 10m	EOL [HVM]
SFP28	NA	25GE	980-9I535-00A015	MFA2P10-A015	Mellanox active optical cable 25GbE, SFP28, 15m	EOL [HVM]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
SFP28	NA	25GE	980-9I536-00A020	MFA2P10-A020	Mellanox active optical cable 25GbE, SFP28, 20m	EOL [HVM]
SFP28	NA	25GE	980-9I539-00A030	MFA2P10-A030	Mellanox active optical cable 25GbE, SFP28, 30m	EOL [HVM]
SFP+	NA	25GE	980-9I094-00AR00	MMA2L20-AR	Mellanox optical transceiver, 25GbE, 25Gb/s, SFP28, LC-LC, 1310nm, LR up to 10km	EOL [MP]
SFP28	NA	25GE	980-9I595-00AM00	MMA2P00-AS	Mellanox transceiver, 25GbE, SFP28, LC-LC, 850nm, SR	HVM

2.2.9.2.8 10GbE Cables

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
Q2S	NA	10GE	980-9I71G-00J000	MAM1Q00A-QSA	Mellanox cable module, ETH 10GbE, 40Gb/s to 10Gb/s, QSFP to SFP+	LTB [HVM]
Q2S	NA	10GE	980-9I65P-00J005	MC2309124-005	Mellanox passive copper hybrid cable, ETH 10GbE, 10Gb/s, QSFP to SFP+, 5m	EOL [P-Rel]
Q2S	NA	10GE	980-9I65Q-00J007	MC2309124-007	Mellanox passive copper hybrid cable, ETH 10GbE, 10Gb/s, QSFP to SFP+, 7m	EOL [P-Rel]
Q2S	NA	10GE	980-9I65R-00J001	MC2309130-001	Mellanox passive copper hybrid cable, ETH 10GbE, 10Gb/s, QSFP to SFP+, 1m	EOL [HVM]
Q2S	NA	10GE	980-9I65S-00J002	MC2309130-002	Mellanox passive copper hybrid cable, ETH 10GbE, 10Gb/s, QSFP to SFP+, 2m	EOL [HVM]
Q2S	NA	10GE	980-9I65T-00J003	MC2309130-003	Mellanox passive copper hybrid cable, ETH 10GbE, 10Gb/s, QSFP to SFP+, 3m	EOL [HVM]
SFP+	NA	10GE	980-9I682-00J004	MC3309124-004	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 4m	EOL [HVM]
SFP+	NA	10GE	980-9I683-00J005	MC3309124-005	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 5m	EOL [HVM]
SFP+	NA	10GE	980-9I684-00J006	MC3309124-006	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 6m	EOL [HVM]
SFP+	NA	10GE	980-9I685-00J007	MC3309124-007	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 7m	EOL [HVM]
SFP+	NA	10GE	980-9I686-00J001	MC3309130-001	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 1m	EOL [HVM]
SFP+	NA	10GE	980-9I688-00J002	MC3309130-002	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 2m	EOL [HVM]

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Marketing Description	Lifecycle Phase
SFP+	NA	10GE	980-9I68B-00J003	MC3309130-003	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 3m	EOL [HVM]
SFP+	NA	10GE	980-9I68F-00J00A	MC3309130-00A	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 0.5m	EOL [HVM]
SFP+	NA	10GE	980-9I68G-00J01A	MC3309130-0A1	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 1.5m	EOL [HVM]
SFP+	NA	10GE	980-9I68H-00J02A	MC3309130-0A2	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 2.5m	EOL [HVM]
SFP+	NA	10GE	980-9I68C-00J003	MCP2100-X003B	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 3m, Blue Pulltab, Connector Label	EOL [HVM]
SFP+	NA	10GE	980-9I68F-00J002	MCP2104-X002B	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 2m, Black Pulltab, Connector Label	EOL [HVM]
SFP+	NA	10GE	980-9I68G-00J003	MCP2104-X003B	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 3m, Black Pulltab, Connector Label	EOL [HVM]
SFP+	NA	10GE	980-9I68I-00J02A	MCP2104-X02AB	Mellanox passive copper cable, ETH 10GbE, 10Gb/s, SFP+, 2.5m, Black Pulltab, Connector Label	EOL [HVM]
SFP+	NA	10GE	MFM1T02A-LR-F	MFM1T02A-LR-F	Mellanox optical module, ETH 10GbE, 10Gb/s, SFP+, LC-LC, 1310nm, LR up to 10km	HVM
SFP+	NA	10GE	MFM1T02A-SR-F	MFM1T02A-SR-F	Mellanox optical module, ETH 10GbE, 10Gb/s, SFP+, LC-LC, 850nm, SR up to 300m	HVM
SFP+	NA	10GE	MFM1T02A-SR-P	MFM1T02A-SR-P	Mellanox optical module, ETH 10GbE, 10Gb/s, SFP+, LC-LC, 850nm, SR up to 300m	HVM

2.2.9.2.9 1GbE Cables

Form Factor	IB Data Rate	Eth Data Rate	NVIDIA SKU	Legacy P/N	Description	LifeCycle Phase
SFP	NA	1GE	980-9I270-00IM00	MC3208011-SX	NVIDIA Optical module, ETH 1GbE, 1Gb/s, SFP, LC-LC, SX 850nm, up to 500m	EOL [P-Rel]
SFP	NA	1GE	980-9I251-00IS00	MC3208411-T	NVIDIA module, ETH 1GbE, 1Gb/s, SFP, Base-T, up to 100m	HVM

2.2.9.2.10 Supported 3rd Party Cables and Modules

> ⚠ Third-party devices that have not been qualified by NVIDIA may be used; however, please be aware that no performance guarantees are provided.
> Any issues that arise will require initiating a new feature request process for third-party support.

Form Factor	Eth Data Rate	Cable P/N	Description
SFP	10GE	1-2053783-2	PASSIVE COPPER CABLE ETH 10GBE SFP 3M
SFP	25GE	FTLF8536P4BCL	TRANSCIEVER 25GBE SFP SR
SFP	25GE	LTF8507-PC07	HISENSE ACTIVE FIBER CABLE, 25GBE
SFP	25GE	FTLF1436P3BCL	TRANSCIEVER 25GBE SFP LR
SFP	10GE	SFP-H10GB-CU5M	PASSIVE COPPER CABLE ETH 10GBE SFP 3M
SFP	10GE	QSFP-H40G-CU5M	PASSIVE COPPER CABLE ETH 10GBE SFP 5M
SFP	10GE	CAB-SFP-SFP-1M	PASSIVE COPPER CABLE ETH 10GBE SFP 1M
SFP	10GE	CAB-SFP-SFP-5M	PASSIVE COPPER CABLE ETH 10GBE SFP 5M
SFP	10GE	CAB-SFP-SFP-3M	PASSIVE COPPER CABLE ETH 10GBE SFP 3M
SFP+	10GE	SFP-10G-SR	CISCO 10GBASE SFP
SFP	10GE	SFP-H10GB-SU5M	MOLEX SFP-H10GB-SU5M
SFP	25GE	SFP-H25G-CU3M	CISCO 25GBASE-CR1 COPPER CABLE 3-METER NDCCGJ-C403
SFP	10GE	FTLX8574D3BCL	FTLX8571D3BCL-ME 10GBASE-LR
SFP+	10GE	FTLX1471D3BCL-ME	FTLX1471D3BCL-ME 10GBASE-LR
SFP	10GE	QSFP-4SFP10G-CU5M	CISCO QSFP-4SFP10G-CU5M
SFP+	10GE	FTLX8571D3BCL-ME	DELL FINISAR 10GB SFP 850NM OPTIC TRANSCEIVER
SFP	1GE	FTLF8519P3BTL-N1	1000BASE-SX AND 2G FIBRE CHANNEL (2GFC) 500M INDUSTRIAL TEMPERATURE SFP OPTICAL TRANSCEIVER
SFP+	1GE	FTLF1428P2BNV	8 GIGABIT ROHS COMPLIANT LONG-WAVELENGTH SFP+ TRANSCEIVER
SFP+	1GE	FTLF8528P2BCV	8.5 GB/S SHORT-WAVELENGTH SFP+ TRANSCEIVER

Form Factor	Eth Data Rate	Cable P/N	Description
SFP+	25GE	74752-9520	CISCO MOLEX SFP H10GB CX1 PASSIVE CU3M 74752 9520
SFP	1GE	FCLF8522P2BTL	CBL ASSY 4X25G ETH QSFP 1M
SFP	25GE	NDCCGJ-C403	CISCO 3M COPPER CABLE
SFP	1GE	FCLF8521P2BTL	1000BASE-T ROHS COMPLIANT COPPER SFP TRANSCEIVER
SFP	10GE	SFP-10GB-SR	CISCO-SUMITOMO 100GBE AOM
SFP	10GE	SFP-10G-T-A-ENC	10GBASE-T COPPER SFP+ FOR CAT6A CAT7 RJ 45 30M MAX
SFP+	10GE	FTLX1471D3BCL-EM	10GB/S 10KM SINGLE MODE DATACOM SFP+ TRANSCEIVER
SFP+	10GE	FTLX1672D3BCL	10GB/S, 40KM SINGLE MODE, MULTI-RATE SFP+ TRANSCEIVER
SFP	1GE	AFBR-5710PZ	AVAGO AFBR-5710PZ COMPATIBLE 1000BASE-SX SFP 850NM
SFP+	10GE	AFBR-703SDZ-MX1	AVAGO 10GB ETHERNET 850NM 10GBASE-SR
QSFP to 4xSFP	10GE---40GE	FCBN510QE2C03	CABLE 4SFP+-QSFP+ M-M 3M
SFP56	50GE	FTLF8556D1BCW	FIBRE OPTIC TRANSMITTERS, RECEIVERS, TRANSCEIVERS 850NM OXIDE VCSEL, 50GE, 53.125/25.78/10.3 GB/S TRANSCEIVER
SFP+	10GE	FTLX8574D3BCV	FIBRE OPTIC TRANSMITTERS, RECEIVERS, TRANSCEIVERS 10G 850-NM VCSEL 1G/10G DUAL-RATE
SFP+	10GE	MFM1T02A-SR	10GBASE-LR SFP+
SFP28	25GE	GSS-MDO250-007CA	25GE SFP28 ENHANCED AOC
SFP	1GE	RTXM191-450-H3C	1.25G-1310NM-40KM OPTICAL TRANSCEIVER SFP
SFP+	10GE	R0R41A	10GB BASE-T P15891-001 SFP+ RJ45 30M XCVR TRANSCEIVER
SFP+	10GE	R0R41B	10GBASE-T SFP+ RJ45 300M
SFP+	10GE	SP7051-HP	SFP+ 10GBASE-T
QSFP-DD	400GE	C-DQ8FNM005-N00	MELLANOX SELECT 400GBE QSFP-DD AOC 5M
QSFP-DD	400GE	C-DQ8FNM050-N00	MELLANOX SELECT 400GBE QSFP-DD AOC 5M
QSFP-DD	400GE	QDD-400G-SR8	400GBASE-SR8 QSFP-DD PAM4 850NM 100M DOM MTP/MPO-16 MMF OPTICAL TRANSCEIVER MODULE
QSFP-DD	400GE	DMQ8811A-EC05	QSFP-DD AOC 100M
SFP	1GE	FTLF1318P3BTL-IB	TRANSCEIVER MODULE ETHERNET, FIBRE CHANNEL 1.25GBPS 1310NM 3.3V LC DUPLEX PLUGGABLE, SFP
SFP	1GE	FTLF8519P3BNL-IB	TRANSCEIVER MODULE ETHERNET, FIBRE CHANNEL 1.25GBPS 1310NM 3.3V LC DUPLEX PLUGGABLE, SFP

Form Factor	Eth Data Rate	Cable P/N	Description
QSFP28	200GE	L6WQF102-SD-R	3M DAC, 200G TO 2X100G
SFP28	25GE	MMA2L20-AR-PLV	25G LR SFP28 TRANSCEIVER (SMF, 1310NM, 10KM, LC, DOM)
SFP28	25GE	MMA2P00-AS-PLV	25GBASE-SR SFP28 TRANSCEIVER (MMF, 850NM, 100M, LC, DOM)
SFP+	10GE	DM7053	COPPER 10G BASE-T
SFP	1GE	FTLF1318P3BTL	TRANSCEIVER MODULE ETHERNET, FIBRE CHANNEL 1.25GBPS 1310NM 3.3V LC DUPLEX PLUGGABLE, SFP
SFP+	10GE	GLC-T-10G-AO	CISCO GLC-T-10G COMPATIBLE TAA COMPLIANT 100,1000,10000BASE-TX SFP+ TRANSCEIVER (COPPER, 30M, RJ-45)
SFP+	10GE	EX-SFP-10GE-LR	1FP+ 10GBASE-LR 10 GIGABIT ETHERNET OPTICS, 1310NM FOR 10KM TRANSMISSION ON SMF, EX-SFP-10GE-LR-LU
SFP+	10GE	EX-SFP-10GE-SR	10GBASE-SR SFP+ TRANSCEIVER, 850NM, 300M, EX-SFP-10GE-SR-LU
QSFP-DD	200GE	DEF8504-2C06-MB3	QSFP-DD ACTIVE OPTICAL CABLE (AOC) TO 2XQSFP-28 ACTIVE OPTICAL CABLE BREAK-OUT
QSFP-DD to 4xQSFP56	100GE	NDYSV2-0003	400G TO 4X 100G QSFP DD - 4X QSFP56 CABLE ASSEMBLY, LINEAR ACTIVE, 30AWG, 3M, 56G / LANE, JACKET
QSFP-DD to 4xQSFP56	100GE	NDYSV2-0008	400G TO 4X 100G QSFP DD - 4X QSFP CABLE ASSEMBLY, LINEAR ACTIVE, 30AWG, 2.5M, 56G / LANE, JACKET
QSFP-DD to 2xQSFP56	200GE	NDYRYH-0003	400GGTO 2X200FG QSFP DD - 2X QSFP CABLE ASSEMBLY, PASSIVE, 27AWG, 3M, 56G / LANE, JACKET
QSFP-DD to 2xQSFP56	200GE	DME8811-EC07	400G-2X200G SPLIT 7M AOC CABLES (400G QSFP-DD BREAKING OUT TO 2X 200G QSFP56
QSFP-DD to 4xQSFP56	100GE	NDYSYF-0001	400G TO 4X100G QSFP DD - 4X QSFP56 CABLE ASSEMBLY, PASSIVE, 30AWG, 1M
QSFP-DD to 2xQSFP56	200GE	NDYRYF-0001	400G TO 2X200G, QSFP DD - 2X QSFP56 CABLE ASSEMBLY, PASSIVE, 30AWG, 1M
QSFP-DD to 2xQSFP56	200GE	NDYRYH-0002	400G TO 2X200G, QSFP DD - 2X QSFP56 CABLE ASSEMBLY, PASSIVE, 27AWG, 2M
SFP+	1GE	FCLF8521P2BTL-HP	1GBT SFP+ RJ-45 TRANSCEIVER MODULE
SFP56	50GE	FTLF8556E1BCL	50G SFP56 SR UNIVERSAL TRANSCEIVER, 850NM, 100M, LC-DUPLEX, MULTIMODE MMF
QSFP-DD to 8 SFP56	50GE	NDYNYH-0001	AMPHENOL QSFP DD - 8X SFP CABLE ASSEMBLY, PASSIVE, 27AWG, 1M, 56G / LANE, JACKET

Form Factor	Eth Data Rate	Cable P/N	Description
QSFP-DD to 8 SFP56	50GE	NDYNYH-0002	AMPHENOL QSFP DD - 8X SFP CABLE ASSEMBLY, PASSIVE, 27AWG, 2M, 56G / LANE, JACKET
QSFP-DD to 8 SFP56	50GE	NDYNYH-0003	AMPHENOL QSFP DD - 8X SFP CABLE ASSEMBLY, PASSIVE, 27AWG, 3M, 56G / LANE, JACKET
SFP	10GE	74752-9096	CISCO-MOLEX 10G SFP+ ACTIVE DAC, 5M
SFP	10GE	74752-9521	CISCO 10G SFP+ MODULE
QSFP-DD to 2xQSFP56	400GE	AAQD2QP2400C003	AOI 400G BREAKOUT TO 2X200G BREAKOUT AOC
QSFP56	400GE	ATRF-C020	HGTECH 200G QSFP56 AOC 20M
SFP28	25GE	ATRS-2005	HGTECH 25G SFP28 AOC 5M
SFP28	25GE	ATRS-2007	HGTECH 25G SFP28 AOC 7M
QSFP-DD to 2xQSFP56	400GE	C-DQF8FNMxxx-N00	INNOLIGHT 400G QSFP-DD TO 2X200G QSFP56 BREAKOUT AOC
SFP28	25GE	FCBG125SD1C05M	FINISAR SFP28 AOCS 5M
SFP28	25GE	FCBG125SD1C10M	FINISAR SFP28 AOCS 7M
QSFP-DD to 2xQSFP56	400GE	FCBN950QE1C05	FINISAR 400G QSFP-DD TO 2X200G QSFP56 BREAKOUT AOC 5M
QSFP-DD to 2xQSFP56	400GE	FCBN950QE1C20	FINISAR 400G QSFP-DD TO 2X200G QSFP56 BREAKOUT AOC 20M
SFP28	25GE	LTF8507-PC05	HISENSE 25G SFP28 AOCS 5M
QSFP-DD to 2xQSFP56	400GE	QDD-2Q200-CU3M	CISCO 400G QSFP-DD TO 2X200G BREAKOUT DAC 3M
QSFP-DD to 4xQSFP56	400GE	QDD-4ZQ100-CU1M	CISCO 400G QSFP-DD TO 4X100G BREAKOUT DAC 1M
SFP28	25GE	RTXM330-005	ACCELINK 25G SFP28 AOCS 5M
SFP28	25GE	RTXM330-007	ACCELINK 25G SFP28 AOCS 7M
QSFP56	200GE	RTXM500-301-F1	ACCELINK 200G QSFP56 SR4
QSFP-DD to 2xQSFP56	400GE	RTXM500-910	ACCELINK 400G QSFP-DD TO 2X200G QSFP56 BREAKOUT AOC 10M
SFP+	10GE	SFP-H10GB-CU1M	CISCO 10G SFP+ DAC
QSFP56	200GE	T-FX4FNS-N00	INNOLIGHT 200G QSFP56 SR4 MODULE
SFP28	25GE	AFBR-735ASMZ-HT1	BROADCOM DUAL RATE 10G/25G SR SFP28 MODULE

Form Factor	Eth Data Rate	Cable P/N	Description
QSFP56	200GE	QSFP-200-CU3M	200G QSFP56 TO QSFP56 PASSIVE COPPER CABLE, 3M

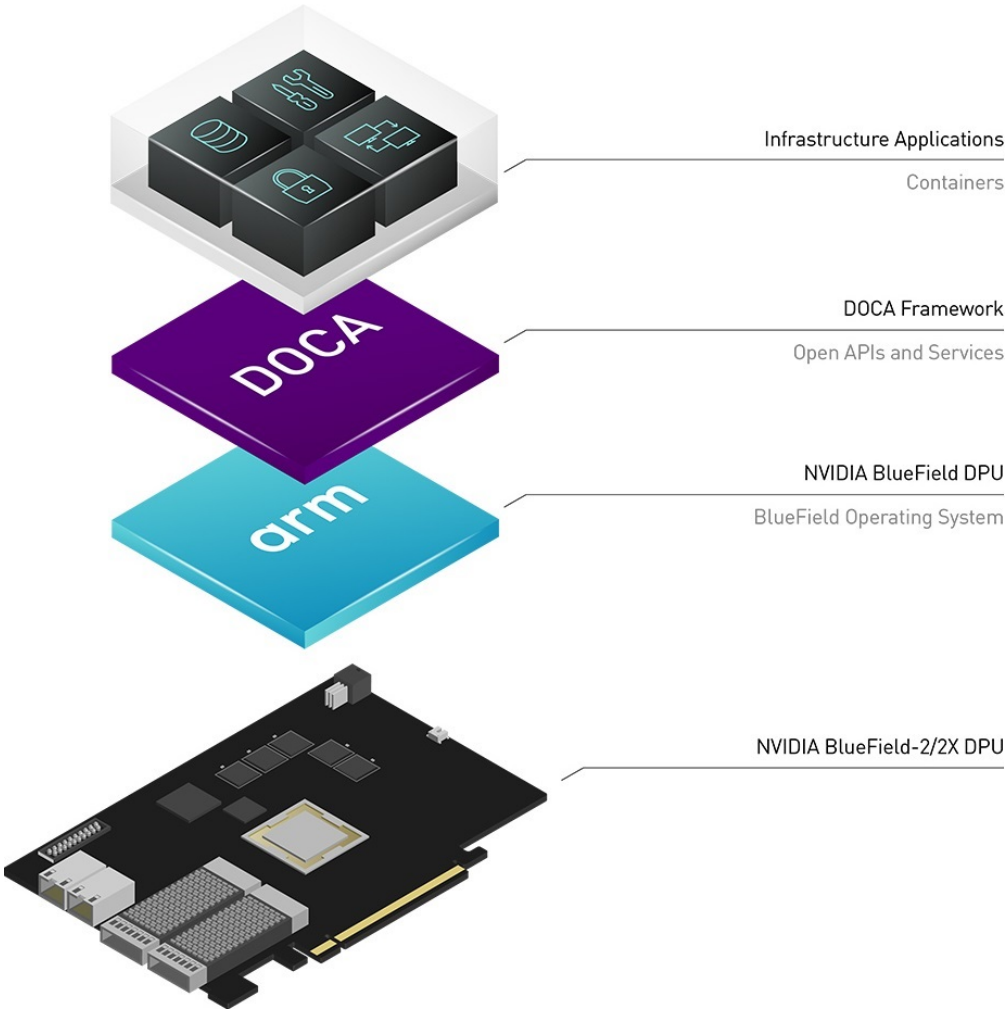
2.3 BlueField and DOCA User Types

This guide provides a quick introduction to the NVIDIA® BlueField® networking platform, its DOCA software components, and BlueField user types.

2.3.1 Introduction

The BlueField family of networking platforms includes data processing units (DPUs) and SuperNICs, and is optimized for traditional enterprise, high-performance computing (HPC), and modern cloud workloads, delivering a broad set of accelerated software-defined networking, storage, security, and management services. BlueField enables organizations to transform their IT infrastructures into state-of-the-art data centers that are accelerated, fully programmable, and armed with zero-trust security to prevent data breaches and cyber-attacks.

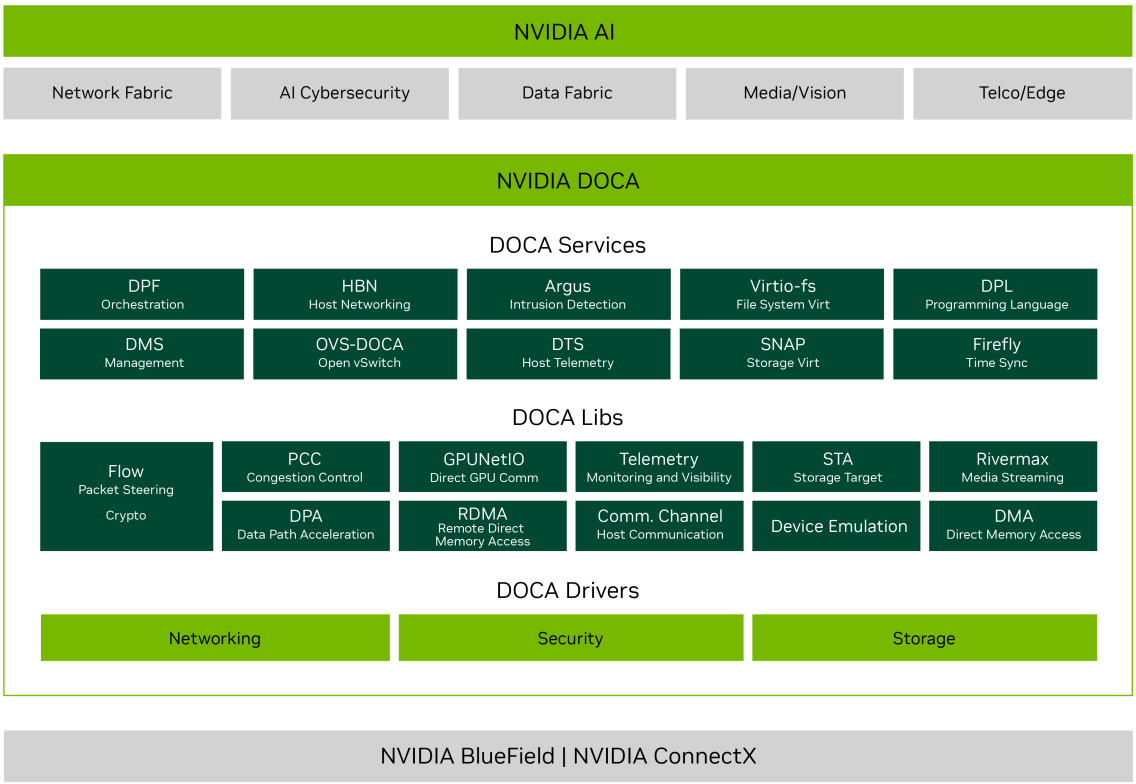
NVIDIA DOCA™ brings together a wide range of powerful APIs, libraries, and frameworks for programming and acceleration of the modern data center infrastructure. Like NVIDIA® CUDA® for GPUs, DOCA is a consistent and essential resource across all existing and future generations of BlueField products.



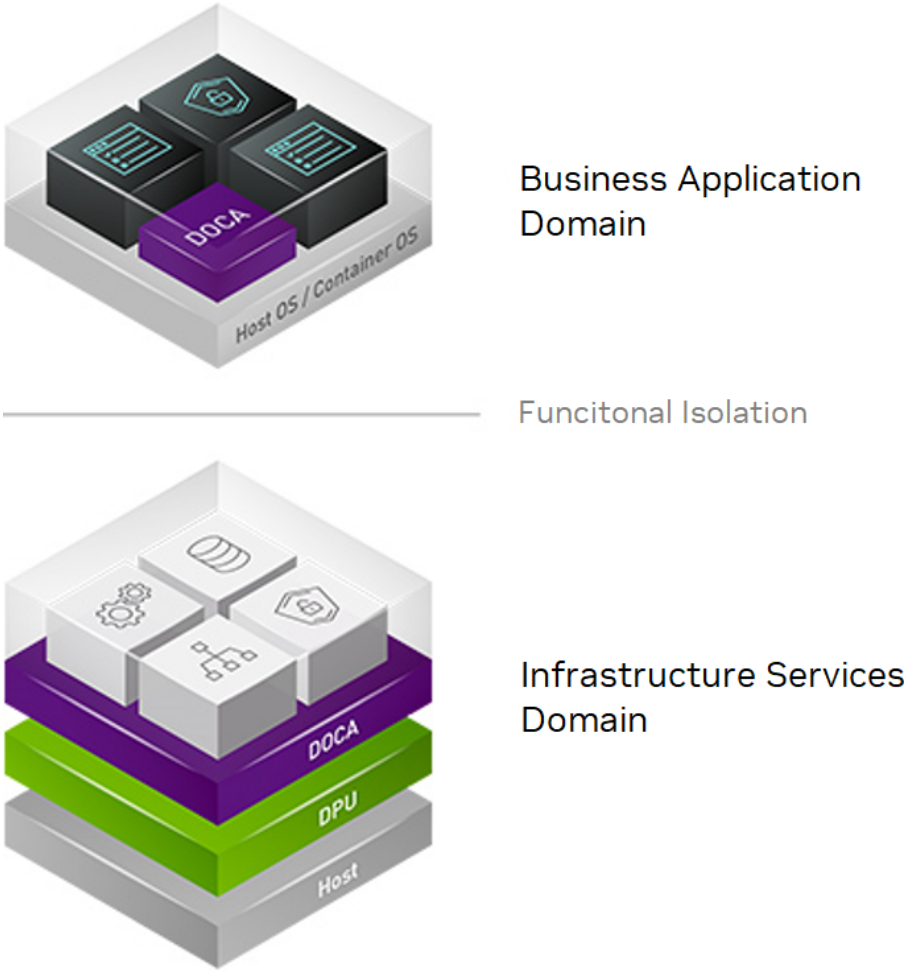
2.3.2 DOCA Components

DOCA software consists of a development and a runtime environment.

- DOCA-Devel provides industry-standard open APIs and frameworks, including Data Plane Development Kit (DPDK) and P4 for networking and security, and the Storage Performance Development Kit (SPDK) for storage. The frameworks simplify application offload with integrated NVIDIA acceleration packages. The Devel environment supports a range of operating systems and distributions and includes drivers, libraries, tools, documentation, and reference applications.



- DOCA runtime includes tools for provisioning, deploying, and orchestrating containerized services on BlueField Platforms in bulk across the data center.



2.3.3 BlueField Networking Platform User Types

2.3.3.1 BlueField Administrator

A BlueField administrator can be a system admin, an IT specialist, a security operations specialist, or anyone managing data center servers and their functionality. The admin would usually be interfacing with BlueField configuration and DOCA services and applications running on the BlueField Platform.

Common operations performed by the BlueField admin:

- Updating the BlueField image
- Running reference applications on the BlueField Platform
- Running DOCA services on the BlueField Platform

For more information, please visit BlueField Administrator Quick Start Guide.

2.3.3.2 DOCA Developer

A DOCA developer creates the services and applications that run on top of the BlueField Platform and usually interfaces with DOCA libraries and drivers to create the necessary workflow and functionality.

Common operations performed by the DOCA developer:

- Developing DOCA applications using DOCA libraries and drivers
- Compiling DOCA reference applications
- Using DOCA sample code to create a new workflow

For more information, please refer to the DOCA Developer Quick Start Guide.

2.4 DOCA EULA

2.4.1 End-User License Agreement

This NVIDIA DOCA License agreement (“Agreement) is a legal agreement between you and Mellanox Technologies, Ltd. ("NVIDIA") and governs the use of the NVIDIA DOCA software and related materials provided under this Agreement (“SOFTWARE”).

This Agreement can be accepted only by an adult of legal age of majority in the country in which the SOFTWARE is used. If you are entering this Agreement on behalf of a company or other legal entity, you represent that you have legal authority and “you” will mean the entity you represent.

By using the SOFTWARE, you affirm that you have reached the legal age of majority, you accept the terms of this Agreement, and you take legal and financial responsibility for the actions of your permitted users.

You agree to use the SOFTWARE only for purposes that are permitted by (a) this Agreement, and (b) any applicable law, regulation or generally accepted practices or guidelines in the relevant jurisdictions.

1. License. Subject to the terms of this Agreement, NVIDIA hereby grants you a non-exclusive, non-transferable license, without the right to sublicense (except as expressly provided in this Agreement) to:

1. Install and use the SOFTWARE,
2. Modify and create derivative works of sample or reference source code delivered in the SOFTWARE (“Derivatives”),
3. Use the provided SOFTWARE to develop your own implementation of the API, provided that such implementation does not modify NVIDIA’s APIs, and
4. Distribute the following portions of the SOFTWARE as incorporated in object code format into a software application, subject to the distribution requirements indicated in Section 2 of this Agreement: API headers, drivers, libraries and sample applications.

BlueField SNAP software and materials, if delivered to you under this Agreement, are licensed only for use in BlueField DPUs and SuperNICs (“Networking Platform”), Per Networking Platform. “Per Networking Platform” license means a license that allows concurrent authorized users to use the SOFTWARE in a single BlueField Networking Platform under the license, and in some cases the SKU or documentation will indicate the maximum number of concurrent authorized users or

virtual machines per Networking Platform. Notwithstanding contrary terms in Section 1 above, you may not use or copy BlueField SNAP software without the necessary licenses.

2. Distribution Requirements. These are the distribution requirements for you to exercise the grants above:

1. An application must have material additional functionality, beyond the included portions of the SOFTWARE.
2. The distributable portions of the Software shall only be accessed by your application.
3. The following notice shall be included in distributed modifications and derivative works of source code: “This software contains source code provided by Mellanox Technologies Ltd.”
4. Unless you have a separate agreement in writing with NVIDIA , any developer tools provided under this Agreement are delivered for your internal use only.
5. You agree to distribute the SOFTWARE subject to the terms at least as protective as the terms of this Agreement, including (without limitation) terms relating to the license grant, license restrictions and protection of NVIDIA’s intellectual property rights. Additionally, you agree that you will protect the privacy, security and legal rights of your application users.
6. You agree to notify NVIDIA in writing of any known or suspected distribution or use of the SOFTWARE not in compliance with the requirements of this Agreement, and to enforce the terms of your agreements with respect to the distributed portions of the SOFTWARE.

3. Authorized Users. You may allow employees and contractors of your entity or of your subsidiary(ies) to access and use the SOFTWARE from your secure network to perform work on your behalf. If you are an academic institution you may allow users enrolled or employed by the academic institution to access and use the SOFTWARE from your secure network. You are responsible for the compliance with the terms of this Agreement by your authorized users. Any act or omission that if committed by you would constitute a breach of this Agreement will be deemed to constitute a breach of this Agreement if committed by your authorized users.

4. Limitations. Your license to use the Software and Derivatives is restricted as stated in this Section 4 (“License Restrictions”). You will cooperate with NVIDIA and, upon NVIDIA’s written request, you will confirm in writing and provide reasonably requested information to verify your compliance with the terms of this Agreement.

1. The SOFTWARE is licensed for you to develop applications only for their use in systems with NVIDIA DPUs, NVIDIA SuperNICs or NVIDIA adapter products or related NVIDIA adapter products. Except as provided in this Agreement, you may not modify, reverse engineer, decompile or disassemble, or remove copyright or other proprietary notices from any portion of the SOFTWARE or copies of the SOFTWARE.
2. You may not distribute or disclose to third parties the output of the SOFTWARE where the output reveals functionality or performance data pertinent to NVIDIA hardware or software products, the results of benchmarking, competitive analysis, regression or performance data relating to the SOFTWARE without the prior written permission from NVIDIA .
3. Except as expressly provided in this Agreement, you may not copy, sell, rent, sublicense, transfer, distribute or otherwise make available to others (except as stated in Section 3 (“Authorized Users”), modify, or create derivative works of any portion of the SOFTWARE or Derivates. For clarity, unless you have an agreement with NVIDIA for this purpose you may not distribute or sublicense the SOFTWARE as a stand-alone product.
4. Unless you have an agreement with NVIDIA for this purpose, you may not indicate that an application created with the SOFTWARE is sponsored or endorsed by NVIDIA.

5. You may not bypass, disable, or circumvent any technical limitation, encryption, security, digital rights management or authentication mechanism in the SOFTWARE.
6. You may not use the SOFTWARE or Derivatives in any manner that would cause them to become subject to an open source software license subject to the terms in the Section 8 below ("Components Under Other Licenses").
7. You may not replace any NVIDIA software components in the Software that are governed by this Agreement with other software that implements NVIDIA APIs.
8. You may not reverse engineer, decompile or disassemble any portion of the output generated using Software elements for the purpose of translating such output artifacts to target a non-NVIDIA platform.
9. You may not use the SOFTWARE in or with any system or application where the use or failure of such system or application developed or deployed with SOFTWARE could result in injury, death or catastrophic damage ("Mission Critical Applications"). NVIDIA will not be liable to you or any third party, in whole or in part, for any claims or damages arising from uses in Mission Critical Applications.
10. You agree to defend, indemnify and hold harmless NVIDIA and its affiliates, and their respective employees, contractors, agents, officers and directors, from and against any and all claims, damages, obligations, losses, liabilities, costs or debt, fines, restitutions and expenses (including but not limited to attorney's fees and costs incident to establishing the right of indemnification) arising out of or related to your use of the SOFTWARE outside of the scope of this Agreement, or not in compliance with its terms.

5. Confidentiality. You agree that you will not use, nor authorize others to use, NVIDIA Confidential Information, except as necessary for the performance of this Agreement, and that you will not disclose NVIDIA Confidential Information to any third party, except to permitted users under this Agreement that have a need to know such Confidential Information for the purpose of this Agreement, provided that each such recipient is subject to a written agreement that includes confidentiality obligations consistent with these terms. You will protect the NVIDIA Confidential Information with at least the same degree of care that you use to protect your own similar confidential and proprietary information, but no less than a reasonable degree of care. "Confidential Information" means the Software including its features and functionality, output, and any results of benchmarking or other competitive analysis or regression or performance data relating to the Software.

6. Updates NVIDIA may, at its option, make available patches, workarounds or other updates to this SOFTWARE. Unless the updates are provided with their separate governing terms, they are deemed part of the SOFTWARE licensed to you as provided in this Agreement. You agree that the form and content of the SOFTWARE that NVIDIA provides may change without prior notice to you.

7. Pre-Release Versions. SOFTWARE versions identified as alpha, beta, preview, early access or otherwise as pre-release may not be fully functional, may contain errors or design flaws, and may have reduced or different security, privacy, availability, and reliability standards relative to commercial versions of NVIDIA software and materials. You may use a pre-release SOFTWARE version at your own risk, understanding that these versions are not intended for use in production or business-critical systems. NVIDIA may choose not to make available a commercial version of any pre-release SOFTWARE. NVIDIA may also choose to abandon development and terminate the availability of a pre-release SOFTWARE at any time without liability.

8. Components Under Other Licenses. The Software may include or be distributed with components provided with separate legal notices or terms that accompany the components, such as open source software licenses and other license. The components are subject to the applicable

other licenses, including any proprietary notices, disclaimers, requirements and extended use rights; except that this Agreement will prevail regarding the use of third-party open source software, unless a third-party open source software license requires its license terms to prevail. Open source software license means any software, data or documentation subject to any license identified as an open source license by the Open Source Initiative (http://opensource.org), Free Software Foundation (http://www.fsf.org) or other similar open source organization or listed by the Software Package Data Exchange (SPDX) Workgroup under the Linux Foundation (http://www.spdx.org).

9. Ownership.

9.1 NVIDIA Ownership. The Software, including all intellectual property rights, is and will remain the sole and exclusive property of NVIDIA or its licensors. Except as expressly granted in this Agreement, (i) NVIDIA reserves all rights, interests and remedies in connection with the Software and (ii) no other license or right is granted to you by implication, estoppel or otherwise.

9.2 Your Ownership. Subject to the rights of NVIDIA and its suppliers in the SOFTWARE, which continue to be licensed as stated in this Agreement, even when incorporated in your products or services and the extent permitted by applicable law, as between you and NVIDIA, you hold all rights, title and interest in and to your applications and your Derivatives you develop as permitted in this Agreement including their respective intellectual property rights.

10. Feedback. You may, but are not obligated to, provide suggestions, requests, fixes, modifications, enhancements or other feedback regarding or in connection with your use of the Software (collectively, "Feedback"). Feedback, even if designated as confidential by you, will not create any confidentiality obligation for NVIDIA or its affiliates. If you provide Feedback, you hereby grant NVIDIA , its affiliates and its designees a non-exclusive, perpetual, irrevocable, sublicensable, worldwide, royalty-free, fully paid-up and transferable license, under your intellectual property rights, to publicly perform, publicly display, reproduce, use, make, have made, sell, offer for sale, distribute (through multiple tiers of distribution), import, create derivative works of and otherwise commercialize and exploit the Feedback at NVIDIA's discretion. You will not give Feedback (i) that you have reason to believe is subject to any restriction that impairs the exercise of the grant stated in this section, such as third-party intellectual property rights or (ii) subject to license terms which seek to require any product incorporating or developed using such Feedback, or other intellectual property of NVIDIA or its affiliates, to be licensed to or otherwise shared with any third party.

11. DISCLAIMER OF WARRANTIES. THE SOFTWARE IS PROVIDED by NVIDIA AS-IS and with all faults. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW NVIDIA AND ITS AFFILIATES EXPRESSLY DISCLAIM ALL WARRANTIES and Representations, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING without limitation, WARRANTIES OF TITLE, MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR A PARTICULAR PURPOSE. Without limiting the foregoing, NVIDIA DOES NOT WARRANT THAT THE SOFTWARE WILL MEET YOUR REQUIREMENTS; THAT ANY DEFECTS OR ERRORS WILL BE CORRECTED; THAT ANY CERTAIN CONTENT WILL BE AVAILABLE; OR THAT THE SOFTWARE IS FREE OF VIRUSES OR OTHER HARMFUL COMPONENTS. NO INFORMATION OR ADVICE GIVEN BY NVIDIA WILL IN ANY WAY INCREASE THE SCOPE OF ANY WARRANTY EXPRESSLY PROVIDED IN THIS AGREEMENT.

12. LIMITATIONS OF LIABILITY.

12.1 Disclaimers. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, in no event will NVIDIA AND ITS AFFILIATES BE LIABLE FOR ANY (I) Indirect, SPECIAL, INCIDENTAL, PUNITIVE OR CONSEQUENTIAL DAMAGES, OR (II) Damages for the (A) COSTS OF PROCURING SUBSTITUTE PRODUCTS or (B) LOSS of PROFITS, Revenues, USE, DATA OR GOODWILL ARISING OUT OF OR

related to This Agreement WHETHER BASED UPON BREACH OF CONTRACT, TORT (INCLUDING NEGLIGENCE), Strict LIABILITY OR otherwise AND EVEN IF NVIDIA HAS PREVIOUSLY BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES and even if a Party's remedies fail their essential purpose.

12.2 Damages Cap. Additionally, to the maximum extent permitted by applicable law, NVIDIA'S AND ITS AFFILIATES TOTAL CUMULATIVE Aggregate LIABILITY for any and all liabilities, obligations or claims ARISING OUT OF THIS Agreement will not EXCEED US$10.00.

13. Your Privacy: Collection and Use of Information.

13.1 Privacy Policy. Please review the NVIDIA Privacy Policy, located at https://www.nvidia.com/en-us/about-nvidia/privacy-policy, which explains NVIDIA's policy for collecting and using data, as well as visit the NVIDIA Privacy Center, located at https://www.nvidia.com/en-us/privacy-center, to manage your consent and privacy preferences.

13.2 Collection Purposes. NVIDIA may collect certain personal information, such as your name and email address or those of your authorized users, and other information necessary to authenticate and enable you or your authorized users' access to the Software. Where appropriate you will disclose to, and obtain any necessary consent from, your authorized users to allow NVIDIA to collect such information.

13.3 Third Party Privacy Practices. The Software may contain links to third party websites and services. NVIDIA encourages you to review the privacy statements on those sites and services that you choose to visit to understand how they may collect, use and share your data. NVIDIA is not responsible for the privacy statements or practices of third-party sites or services.

14. Termination. This Agreement will terminate automatically without notice from NVIDIA if you fail to comply with any of the terms in this Agreement or if you commence or participate in any legal proceeding against NVIDIA with respect to the SOFTWARE. Additionally, NVIDIA may terminate this Agreement with prior written notice to you, if NVIDIA decides to no longer provide the SOFTWARE in a country or, in NVIDIA's sole discretion, the continued use of it is no longer commercially viable or creates liabilities for NVIDIA . Upon any termination of this Agreement, you must stop using and destroy all copies of the SOFTWARE. Upon written request, you will certify in writing that you have complied with your commitments under this section. Your prior distributions in accordance with this Agreement are not affected by the termination of this Agreement. All provisions of this Agreement will survive termination, except for the license granted to you.

15. Governing Law and Jurisdiction. This Agreement will be governed in all respects by the laws of the United States and of the State of Delaware, without regard to the conflicts of laws principles or the United Nations Convention on Contracts for the International Sale of Goods. The state or federal courts residing in Santa Clara County, California shall have exclusive jurisdiction over any dispute or claim arising out of this Agreement and the parties irrevocably consent to personal jurisdiction in those courts; except that, either party may apply for injunctive remedies or an equivalent type of urgent legal relief in any jurisdiction.

16. General.

16.1 No Assignment. This Agreement and your rights and obligations thereunder may not be assigned by you by any means or operation of law without NVIDIA's prior written consent and any attempt to do so is null and void. NVIDIA may assign, delegate, or transfer its rights and obligations under this Agreement by any means or operation of law.

16.2 Trade and Compliance. You agree to comply with all applicable export, import, trade and economic sanctions laws and regulations, including U.S. Export Administration Regulations and Office of Foreign Assets Control regulations. Any violation of such laws by you will void any warranty for the associated products and technologies. You confirm (a) your understanding that export or reexport of certain NVIDIA products or technologies may require a license or other approval from appropriate authorities and (b) that you will not export or reexport any products or technology, directly or indirectly, without first obtaining any required license or other approval from appropriate authorities, (i) to any countries that are subject to any U.S. or local export restrictions (currently including, but not necessarily limited to, Belarus, Cuba, Iran, North Korea, Russia, Syria, the Region of Crimea, Donetsk People's Republic Region and Luhansk People's Republic Region); (ii) to any end user who you know or have reason to know will utilize them in the design, development or production of nuclear, chemical or biological weapons, missiles, rocket systems, unmanned air vehicles capable of a maximum range of at least 300 kilometers, regardless of payload, or intended for military end-use, or any weapons of mass destruction; (iii) to any end-user who has been prohibited from participating in the U.S. or local export transactions by any governing authority; or (iv) to any known military or military-intelligence end-user or for any known military or military-intelligence end-use in accordance with U.S. trade compliance laws and regulations. Use of the Software under this Agreement must be consistent with NVIDIA's pdf (nvidia.com).

16.3. Government Rights. The SOFTWARE, documentation and technology ("Protected Items") are "Commercial products" as this term is defined at 48 C.F.R. 2.101, consisting of "commercial computer software" and "commercial computer software documentation" as such terms are used in, respectively, 48 C.F.R. 12.212 and 48 C.F.R. 227.7202 & 252.227-7014(a)(1). Before any Protected Items are supplied to the U.S. Government, you will (i) inform the U.S. Government in writing that the Protected Items are and must be treated as commercial computer software and commercial computer software documentation developed at private expense; (ii) inform the U.S. Government that the Protected Items are provided subject to the terms of the Agreement; and (iii) mark the Protected Items as commercial computer software and commercial computer software documentation developed at private expense. In no event will you permit the U.S. Government to acquire rights in Protected Items beyond those specified in 48 C.F.R. 52.227-19(b)(1)-(2) or 252.227-7013(c) except as expressly approved by NVIDIA in writing.

16.4. Notices. Please direct your legal notices or other correspondence to NVIDIA Corporation, 2788 San Tomas Expressway, Santa Clara, California 95051, United States of America, Attention: Legal Department with a copy emailed to legalnotices@nvidia.com. If NVIDIA needs to contact you about the Software, you consent to receive the notices by email and agree that such notices will satisfy any legal communication requirements.

16.5 Force Majeure. Neither party will be liable during any period where an event or circumstance prevents or delays that party from performing its obligations under this Agreement and that event or circumstance: (i) is not within the reasonable control of that party and is not the result of that party's negligence, and (ii) cannot be overcome or avoided by that party using reasonably diligent efforts.

16.6 Severability and Amendment. If a court of competent jurisdiction rules that a provision of this Agreement is unenforceable, that provision will be deemed modified to the extent necessary to make it enforceable and the remainder of this Agreement will continue in full force and effect. Any amendment to this Agreement must be in writing and signed by authorized representatives of both parties.

16.7 Construction. The headings in the Agreement are included solely for convenience and are not intended to affect the meaning or interpretation of the Agreement. As required by the context of the Agreement, the singular of a term includes the plural and vice versa.

16.8. Entire Agreement. Regarding the subject matter of this Agreement, the parties agree that (i) this Agreement constitutes the entire and exclusive agreement between the parties and supersedes all prior or contemporaneous communications and (ii) and any additional or different terms or conditions contained in purchase orders, order acknowledgements, invoices or otherwise are null and void.

17. Licensing. If the distribution terms in this Agreement are not suitable for your organization, or for any questions regarding this Agreement, please contact NVIDIA at DOCA_License@nvidia.com.

Last updated: April 10, 2025

2.4.2 DOCA Legal Notices and 3rd Party Licenses

The following table provides links to the legal notices, licenses, and third-party attribution documents for the software components included in this NVIDIA DOCA release.

Software Components	Version	3rd Party Components and Licenses
DOCA SDK	3.5.0	• License • 3rd Party Notice • 3rd Party Unify Notice
SoC OS Linux Ubuntu 24.04 Distro	3.3.0-202_26.01_ubuntu-24.04_64k	• Package
DOCA-Host	3.5.0	• License • 3rd Party Notice • 3rd Party Unify Notice
DOCA Platform Foundation	26.8	• License • 3rd Party Notice • 3rd Party Unify Notice
DOCA VPC OVN	26.8	• License • 3rd Party Notice • 3rd Party Unify Notice
Firmware	xx.50.1002	• License • 3rd Party Notice • 3rd Party Unify Notice • HCA Firmware EULA
MFT Linux	4.37.0-154	• License • 3rd Party Notice • 3rd Party Unify Notice
BlueField ATF	4.16.0	• License • 3rd Party Notice • 3rd Party Unify Notice
BlueField EDK2	4.16.0	• License • 3rd Party Notice • 3rd Party Unify Notice

Software Components	Version	3rd Party Components and Licenses
BlueField OP-TEE	4.16.0	• License • 3rd Party Notice • 3rd Party Unify Notice
BlueField RShim	4.16.0	• License • 3rd Party Notice • 3rd Party Unify Notice
BMC	26.07	• License • 3rd Party Notice • 3rd Party Unify Notice
OpenSM	5.26.0	• 3rd Party Notice • 3rd Party Unify Notice
OpenSM MLNX	5.26.1	• License • 3rd Party Notice • 3rd Party Unify Notice
DPCP	1.1.65-1	• License • 3rd Party Notice • 3rd Party Unify Notice
VMA	9.8.84	• License • 3rd Party Notice • 3rd Party Unify Notice
XLIO	3.71.1	• License • 3rd Party Notice • 3rd Party Unify Notice
NVIDIA SHARP	3.16.3	• License • 3rd Party Notice • 3rd Party Unify Notice
Sosreport	3.4.0	• License • 3rd Party Notice • 3rd Party Unify Notice
CollectX	1.26.3-1	• License bin • License DTS • License UFMT • 3rd Party Notice bin • 3rd Party Notice DTS • 3rd Party Notice UFMT • 3rd Party Unify Notice bin • 3rd Party Unify Notice DTS • 3rd Party Unify Notice UFMT
Open vSwitch	3.5.0048	• License • 3rd Party Notice • 3rd Party Unify Notice
DAL2OVS	3.5.0	• License • 3rd Party Notice • 3rd Party Unify Notice
XPlane	3.5.0	• License • 3rd Party Notice • 3rd Party Unify Notice

Software Components	Version	3rd Party Components and Licenses
ibutils2	2.26.0	• License • 3rd Party Notice • 3rd Party Unify Notice
DOCA Binary	3.5	• License • 3rd Party Notice • 3rd Party Unify Notice
DOCA Perftest	3.5.0	• License • 3rd Party Notice • 3rd Party Unify Notice • License (Binary) • 3rd Party Notice (Binary) • 3rd Party Unify Notice (Binary)
HPC-X (Core)	2.51	• License • 3rd Party Notice • 3rd Party Unify Notice
HPC-X HCOLL	4.8.3236	• License • 3rd Party Notice • 3rd Party Unify Notice
HPC-X MPI Tests	2.51	• License • 3rd Party Notice • 3rd Party Unify Notice
HPC-X NCCL RDMA SHARP	2.7	• 3rd Party Notice • 3rd Party Unify Notice
HPC-X NCCL SPCX Plugin	1.5.0	• 3rd Party Notice • 3rd Party Unify Notice
HPC-X OMPI4	2.51	• License • 3rd Party Notice • 3rd Party Unify Notice
HPC-X OMPI5	5.0.10	• License • 3rd Party Notice • 3rd Party Unify Notice
HPC-X UCC	1.9.0	• License • 3rd Party Notice • 3rd Party Unify Notice
HPC-X UCX	1.22.0	• License • 3rd Party Notice • 3rd Party Unify Notice
NVQual	3.5.0	• License • 3rd Party Notice • 3rd Party Unify Notice
Rivermax	1.91	• License • 3rd Party Notice • 3rd Party Unify Notice

3 Quick Start for BlueField Developers

This section contains the following pages:

- DOCA Developer Quick Start Guide

3.1 DOCA Developer Quick Start Guide

This guide details the basic steps to bring up the NVIDIA DOCA development environment and to build and run the DOCA reference applications provided along with the DOCA software framework package.

3.1.1 Introduction

NVIDIA DOCA brings together a wide range of powerful APIs, libraries, and frameworks for programming and accelerating modern data center infrastructures. Like NVIDIA® CUDA® for GPUs, DOCA is a consistent and essential resource across all existing and future generations of BlueField DPU and SuperNIC products.

This document is intended for those wishing to develop applications using the DOCA framework.

> ⚠ Not sure which installation type to use? To expand on different DOCA user types and the relevant installation for each, see BlueField and DOCA User Types.

3.1.2 Install BlueField Networking Platform

Install the BlueField networking platform into your host according to the installation instructions in the BlueField's hardware user guide. The steps include installing BlueField into the PCIe slot and properly securing it in the chassis. Make sure your host OS is listed under the "Supported Host OS and Features per DOCA-Host Installation Profile" section.

3.1.3 Install DOCA Software Package

A detailed step-by-step process for downloading and installing the required development software on both the host and BlueField can be found in the DOCA Installation Guide for Linux.

During installation, you must change the default password, **ubuntu**, to access the NVIDIA® BlueField® networking platform.

3.1.4 Access BlueField

After a successful installation, on the host, the RShim driver exposes a virtual Ethernet device called **tmfifo_net0**.

1. Configure the host side of the **tmfifo_net0** with a static IP to enable IPv4-based communication to the BlueField OS according to the instructions in section "Virtual

Ethernet Interface" of the "Host-side Interface Configuration" page in the *NVIDIA BlueField DPU BSP* document.

2. Log into BlueField's Ubuntu-based OS by running the following command from the host:

```
host# ssh ubuntu@192.168.100.2
```

Use the BlueField networking platform password you defined during the installation process.

At this stage DOCA is installed on BlueField and the host server.

3.1.5 Run Reference DOCA Application

DOCA package assets (e.g., references, tools) are located on Bluefield and on the host under **/opt/mellanox/doca/** .

The DOCA package includes a set of reference applications to facilitate developer on-boarding. Please refer to the DOCA Reference Applications and DOCA Programming Guide for more information.

To run the DOCA Secure Channel reference application which demonstrates accelerated and secure message transmission between the host and BlueField over the Comm Channel interface:

1. Run the application as server on the BlueField networking platform using the following command (all parameters are available in the DOCA Secure Channel Application Guide):

```
# /opt/mellanox/doca/applications/secure_channel/bin/doca_secure_channel -s 256 -n 10 -p 03:00.0 -r 3b:00.0
```

2. Run the application as client on the host using the following command (all parameters are available in the DOCA Secure Channel Application Guide):

```
# /opt/mellanox/doca/applications/secure_channel/bin/doca_secure_channel -s 256 -n 10 -p 3b:00.0
```

3.1.6 More Information

To learn more about NVIDIA BlueField networking platforms, refer to the NVIDIA BlueField Hardware Manuals.

For questions, comments, and feedback, please contact us at doca-feedback@nvidia.com.

4 Installation and Setup

This section contains the following page:

- DOCA Profiles
- BlueField Modes of Operation
- DOCA Installation Guide for Linux
- DOCA Developer Guide

4.1 DOCA Profiles

This document introduces and compares the different NVIDIA DOCA-Host installation profiles, helping users select the most suitable configuration for their workloads and environments.

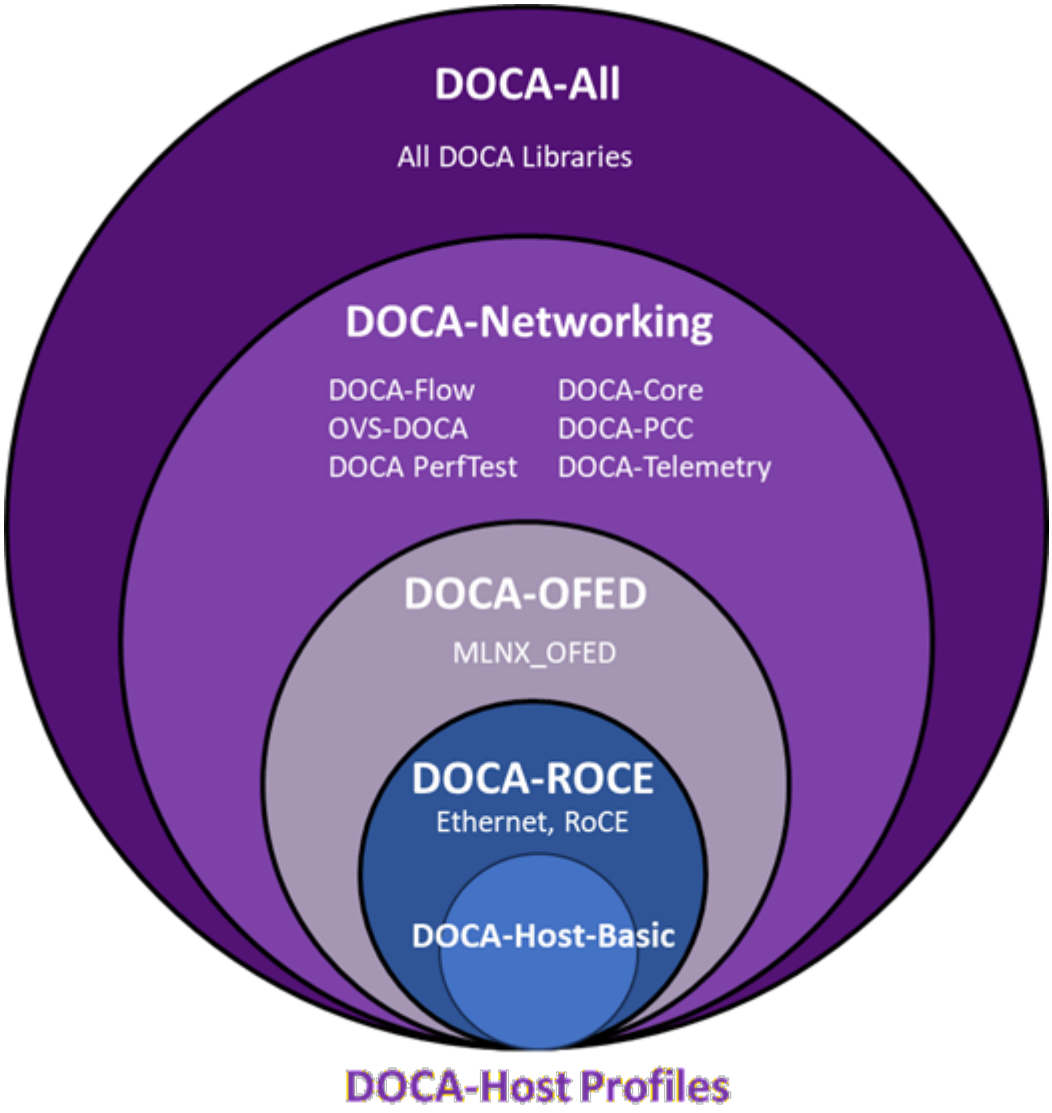
4.1.1 Introduction

NVIDIA DOCA can be installed on host machines to support a range of NVIDIA® BlueField® DPUs and NVIDIA® ConnectX® SmartNICs. To meet the needs of different deployment scenarios, DOCA offers installation profiles, which are validated packages that streamline setup based on the desired functionality.

The following are the available DOCA profiles:

- **doca-all**
- **doca-networking**
- **doca-ofed**
- **doca-roce**
- doca-host-basic

DOCA functionality is limited by the specific device capabilities.



4.1.2 Which Profile to Install?

The functionality of DOCA-Host is limited by the device capabilities (e.g., ConnectX devices cannot utilize DOCA libraries such as DPA, even if **doca-all** is installed on the host).

For BlueField devices:

- NVIDIA recommends the **doca-all** profile
- If you require the smallest installation package for networking-only purposes, use **doca-networking**
- For MLNX_OFED-like installations, use **doca-ofed** (no additional DOCA functionality)
- For RoCE functionality only, install **doca-roce**
- doca-host-basic will provide the common basic networking functionality, similar to doca-roce but without MFT and FW-Updater

For ConnectX devices:

- NVIDIA recommends the **doca-networking** profile
- For deployments with both BlueField and ConnectX devices, use **doca-all**. This is also a good, scalable option for deployments that do not currently use either BlueField or ConnectX devices but eventually might.
- For MLNX_OFED-like installation use **doca-ofed** (no additional DOCA functionality)
- For RoCE functionality only, install **doca-roce**
- For common networking functionality without the added Tools, install doca-host-basic

4.1.3 Profile Details

4.1.3.1 doca-all

Purpose	Install the full DOCA suite, including all libraries, drivers, and tools
Includes	• All DOCA libraries and drivers • MLNX_OFED • DOCA Core • MLNX-DPDK • OVS-DOCA • DOCA Flow
Recommended for	• Full-featured BlueField deployments • Users needing access to all DOCA functionalities • Future-proof mixed BlueField/ConnectX environments

⚠ Some components may require additional manual installation after the default package is installed.

4.1.3.2 doca-networking

Purpose	Networking-specific installation without full DOCA library overhead
Includes	• MLNX_OFED • DOCA Core • DOCA-PCC • OVS-DOCA • DOCA Flow • DOCA Perftest
Recommended for	• Users focused on accelerated networking • BlueField or ConnectX environments where only networking functionality is required

4.1.3.3 doca-ofed

Purpose	Driver-only installation equivalent to MLNX_OFED
Includes	• MLNX_OFED drivers and tools

Recommended for	• Minimalist setups with no DOCA user-space components • Compatibility-focused deployments

4.1.3.4 doca-roce

Purpose	Lightweight RDMA-over-Ethernet (RoCE) functionality
Includes	• rdma-core • ofed-scripts • mlnx-tools • mlnx-ofa_kernel • perftest
Recommended for	• RoCE-only applications • Low-footprint installations

4.1.3.5 doca-host-basic

Purpose	Lightweight Ethernet only functionality
Includes	• rdma-core • mlnx-tools • mlnx-ofa_kernel • perftest • mlnx-ethtool • mlnx-iproute2
Recommended for	• Low-footprint installations with basic Ethernet networking

4.1.4 Supported Devices

All profiles support the following NVIDIA hardware:

- BlueField-3
- BlueField-2
- ConnectX-8
- ConnectX-7
- ConnectX-6 DX / LX / standard
- ConnectX-5
- ConnectX-4 LX

DOCA functionality is limited by the specific device capabilities. For example, ConnectX devices cannot use certain DOCA libraries, such as DPA.

4.1.5 Profile Installation Notes

Each DOCA-Host profile is supported on a subset of operating systems. Refer to Supported Host OS per DOCA-Host Installation Profile for device-specific compatibility information. To install a profile, follow the installation steps in DOCA-Host Installation and Upgrade.

4.2 BlueField Modes of Operation

This page details the different operational modes for NVIDIA® BlueField® networking platforms (DPU, Zero-trust, and NIC modes) and provides instructions for configuring and switching between them.

4.2.1 Introduction

BlueField devices feature the following modes of operation:

Mode of Operation	External Host Trust Level	Description	Default on SKUs	Can be Configured to All Other Modes?
NIC Mode	Host-trusted	The Arm cores of BlueField are inactive, and the device functions as an NVIDIA® ConnectX® network adapter.	SuperNIC SKUs default mode	Yes
DPU Mode	Host-trusted	The Arm cores of BlueField are active, and the embedded Arm system runs services that manage the NIC resources and data path.	DPU SKUs default mode	Yes
	Zero Trust (restricted)	The Arm cores of BlueField are active, and the embedded Arm system runs services to manage the NIC resources and data path while enforcing restrictions on the external host (host isolation).	-	Yes

We don't have a way to export this macro.

4.2.1.1 NIC Mode

In NIC Mode, BlueField operates as a ConnectX network adapter for the external host. For BlueField-3, the Arm cores are inactive, while for BlueField-2, the Arm cores are active but non-functional.

Operating in NIC Mode on BlueField-3 reduces power consumption, improves network performance, and minimizes the host memory footprint.

BlueField-3 SuperNIC SKUs are shipped in NIC Mode by default.

⚠ Multi-host is not supported when BlueField is operating in NIC Mode.

4.2.1.2 DPU Mode

In this operation mode, Arm cores active, known also as embedded CPU function ownership (ECPF) mode, is the default mode for the BlueField DPU family of SKUs.

In DPU Mode, the NIC resources and functionality are owned and controlled by the embedded Arm subsystem. All network communication to the host flows through a virtual switch control plane hosted on the Arm cores that manages all networking traffic coming and going from the host.

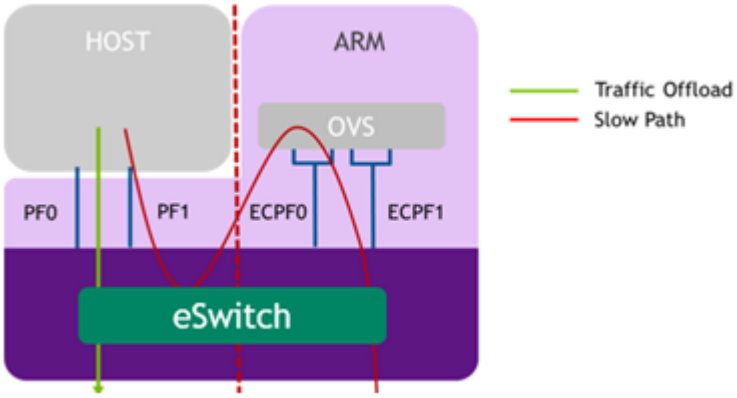
While working in this mode, the BlueField is the trusted function managed by the data center and host administrator for provisioning, management and orchestration (eg. load network drivers, reset an interface, bring an interface up and down, update the firmware, change the mode of operation on BlueField, etc).

⚠ BlueField-2 DPU and BlueField-3 DPU SKUs are shipped in DPU Mode by default.

⚠ Socket Direct is not supported when BlueField is operating in DPU Mode.

4.2.1.2.1 DPU Mode Architecture

In DPU Mode, the BlueField DPU provides the host system with access to network functions, while the host's capabilities are restricted and managed by the Arm processor within the BlueField.



4.2.1.2.1.1 Traffic Management and Paths

The Embedded Control and Processing Framework (ECPF) controls the NIC's embedded switch (eswitch). All network traffic between the host interface and the network initially passes through the BlueField's Arm processor via Representors. This path, where traffic is processed by the Arm processor, is referred to as the "slow path".

To improve performance, the Arm processor can define rules in the eswitch through the ECPF, allowing packets to bypass the Arm processor and be processed directly by the eswitch. This is known as the "fast path", which reduces latency and increases throughput by offloading traffic processing from the Arm to the eswitch.

A virtual switch running on the Arm processor may integrate both slow path and fast path functionalities by processing and classifying only the first packet of a new flow. For subsequent packets in the flow, the virtual switch defines eswitch rules, enabling fast path processing for the remainder of the traffic.

4.2.1.2.1.2 Initialization Process

At startup, network access to the host is initially blocked. This restriction remains until the virtual switch running on the Arm processor loads the default out-of-box rules to manage the ECPF on the BlueField. Once these rules are loaded, network traffic to the host is automatically enabled.

> ⚠ The driver on the host system can only be loaded after the driver on the BlueField has been loaded and has completed NIC configuration. Additionally, all Interface Configuration Memory (ICM) is allocated by the ECPF and resides in the BlueField's memory.

4.2.1.2.2 InfiniBand in DPU Mode

In DPU Mode, when operating with an InfiniBand network, OpenSM must be executed from the BlueField Arm side rather than the host side. Similarly, InfiniBand management tools such as **sminfo**, **ibdev2netdev**, and **ibnetdiscover** can only be used from the BlueField Arm side and are not accessible from the host side.

4.2.1.3 DPU Mode in Zero Trust

Zero Trust, also known as Restricted Mode, is a specialized variation of DPU Mode that enhances security by preventing the host system administrator from accessing BlueField from the host side. Once Zero Trust mode is enabled, the BlueField must be fully controlled by the data center administrator via the Arm cores or the BMC connection, rather than through the host.

This mode enforces security and isolation by restricting the host from performing operations that could compromise BlueField. The following operations can be restricted individually in Zero Trust mode:

- Port ownership – The host cannot assign itself as the port owner
- Hardware counters – The host is denied access to hardware counters
- Tracer functionality – The tracer functionality is blocked
- RShim interface – The RShim interface is disabled
- Firmware flash – firmware flashing from the host is restricted

Zero Trust mode ensures a robust security boundary between the host and BlueField, making it an ideal configuration for environments requiring strict control and isolation.

4.2.2 Moving Between Operation Modes

Transitioning between operation modes can be achieved through various configuration interfaces as presented in the following table.

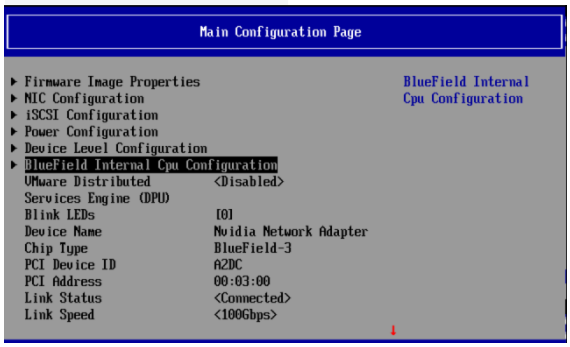
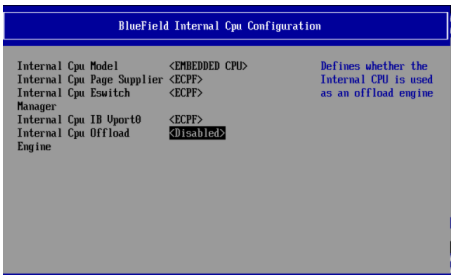
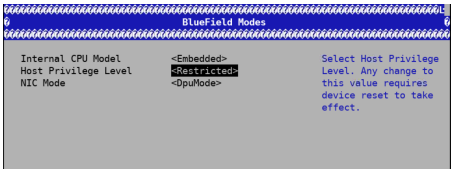
> ⚠ Some configuration interfaces may be locked out when operating in Zero Trust (Restricted) Mode.

Interface	From	To	Configuration option available by default for this interface?	Configuration option can be locked-outfor this interface?
External Host command line	DPU Mode	NIC Mode	Yes	Yes
	NIC Mode	DPU Mode	Yes	Yes
	DPU Mode	DPU Mode with Zero Trust	No	Always blocked
	DPU Mode with Zero Trust	DPU Mode	No	Always blocked
	DPU Mode with Zero Trust	NIC Mode	Not supported	N/A
Host UEFI Menu	DPU Mode	NIC Mode	Yes	Yes
	NIC Mode	DPU Mode	Yes	Yes
	DPU Mode	DPU Mode with Zero Trust	No	Always blocked
	DPU Mode with Zero Trust	DPU Mode	No	Always blocked
	DPU Mode with Zero Trust	NIC Mode	Not supported	N/A
Arm OS command line	DPU Mode	NIC Mode	Yes	No
	NIC Mode	DPU Mode	N/A	N/A
	DPU Mode	DPU Mode with Zero Trust	Yes	No
	DPU Mode with Zero Trust	DPU Mode	Yes	No
	DPU Mode with Zero Trust	NIC Mode	Not supported	N/A
Arm UEFI Menu	DPU Mode	NIC Mode	Yes	No
	NIC Mode	DPU Mode	Yes	No

Interface	From	To	Configuration option available by default for this interface?	Configuration option can be locked-outfor this interface?
	DPU Mode	DPU Mode with Zero Trust	No	No
	DPU Mode with Zero Trust	DPU Mode	No	No
	DPU Mode with Zero Trust	NIC Mode	Not supported	N/A
DPU-BMC Redfish	DPU Mode	NIC Mode	Yes	No
	NIC Mode	DPU Mode	Yes	No
	DPU Mode	DPU Mode with Zero Trust	No	No
	DPU Mode with Zero Trust	DPU Mode	No	No
	DPU Mode with Zero Trust	NIC Mode	Not supported	N/A
Platform-BMC NC-SI OEM commands	DPU Mode	NIC Mode	Yes	No
	NIC Mode	DPU Mode	Yes	No
	DPU Mode	DPU Mode with Zero Trust	No	No
	DPU Mode with Zero Trust	DPU Mode	No	No
	DPU Mode with Zero Trust	NIC Mode	Not supported	N/A

We don't have a way to export this macro.

4.2.2.1 Identifying Which Mode BlueField is Currently Operating In

Interface	Command	Response
Using external host command line	`host> sudo mlxconfig -d /dev/mst/<device> q INTERNAL_CPU_OFFLOAD_ENGINE`	Output format: `RO INTERNAL_CPU_OFFLOAD_ENGINE ENABLED(0)` • **ENABLED(0)** means BlueField is running in DPU Mode • **DISABLED(1)** means BlueField is running in NIC Mode • **RO** is marked for read only, and indicates DPU Mode with Zero Trust
Using external host UEFI (HII) menu	Navigate to specific device and select **BlueField Internal Cpu Configuration**. 	**Internal Cpu Offload Engine** indicates the mode BlueField is currently operating in: • Disabled – BlueField is operating in NIC Mode • Enabled – BlueField is operating in DPU Mode 
Using Arm UEFI (GUI) menu	Access the Arm UEFI menu by pressing the Esc button twice on the console and navigate to **Device Manager** → **System Configuration** → **BlueField Modes**.	The **NIC Mode** menu indicates the mode BlueField is currently operating in: • **NicMode** – BlueField is operating in NIC Mode • **DpuMode** – BlueField is operating in DPU Mode 

<table>
<tr><th>Interface</th><th>Command</th><th>Response</th></tr>
<tr><td>Using DPU BMC Redfish</td><td>

```
curl -k -u root:'<PASSWORD>' -X GET https://
<bmc_ip>/redfish/v1/Systems/Bluefield/Oem/Nvidia
```

</td><td>Read Mode field:

```
{
...
"Mode": <"DpuMode"> or <"NicMode">
...
}
```

"Mode": NicMode – BlueField is operating in NIC Mode
"Mode":DpuMode – BlueField is operating in DPU Mode</td></tr>
<tr><td>Using platform BMC NC-SI OEM commands</td><td>Get Command = 0x13, Parameter = 0x33
<table>
<tr><th>Offset</th><th>31:24</th><th>23:16</th><th>15:8</th><th>7:0</th></tr>
<tr><td>0:15</td><td colspan="4">NC-SI OEM Header (OEM Command)</td></tr>
<tr><td>16:19</td><td colspan="4">Mellanox Manufacture ID (IANA) = 0x8119</td></tr>
<tr><td>20:23</td><td>Command rev=0x00</td><td>MLNX Cmd ID= 0x13</td><td>Parameter=0x33</td><td>Reserved</td></tr>
<tr><td>24:27</td><td colspan="4">Checksum 31:0</td></tr>
</table></td><td>Response format:
<table>
<tr><th>Offset</th><th>31:24</th><th>23:16</th><th>15:8</th><th>7:0</th></tr>
<tr><td>0:15</td><td colspan="4">NC-SI OEM Header (OEM Command)</td></tr>
<tr><td>16:19</td><td colspan="2">Response Code</td><td colspan="2">Reason Code</td></tr>
<tr><td>20:23</td><td colspan="4">Mellanox Manufacture ID (IANA) = 0x8119</td></tr>
<tr><td>24:27</td><td>Command rev=0x00</td><td>MLNX Cmd ID= 0x12</td><td>Parameter=0x33</td><td>Reserved</td></tr>
<tr><td>28:31</td><td colspan="3"></td><td>Offload engine</td></tr>
<tr><td>32:35</td><td colspan="4">Checksum 31:0</td></tr>
</table>
Response field:
<table>
<tr><th>Field</th><th>Size</th><th>Offset in NC-SI Command</th><th>Description</th></tr>
<tr><td>Offload engine</td><td>1 bit</td><td>31.0</td><td>0x0: Enabled - DPU Mode 0x1: Disabled - NIC Mode</td></tr>
</table></td></tr>
</table>

4.2.2.2 Changing Mode

For the configuration to take effect, Arm and NIC components must undergo a reset. Power cycle is recommended.

4.2.2.2.1 Using External Host Command Line

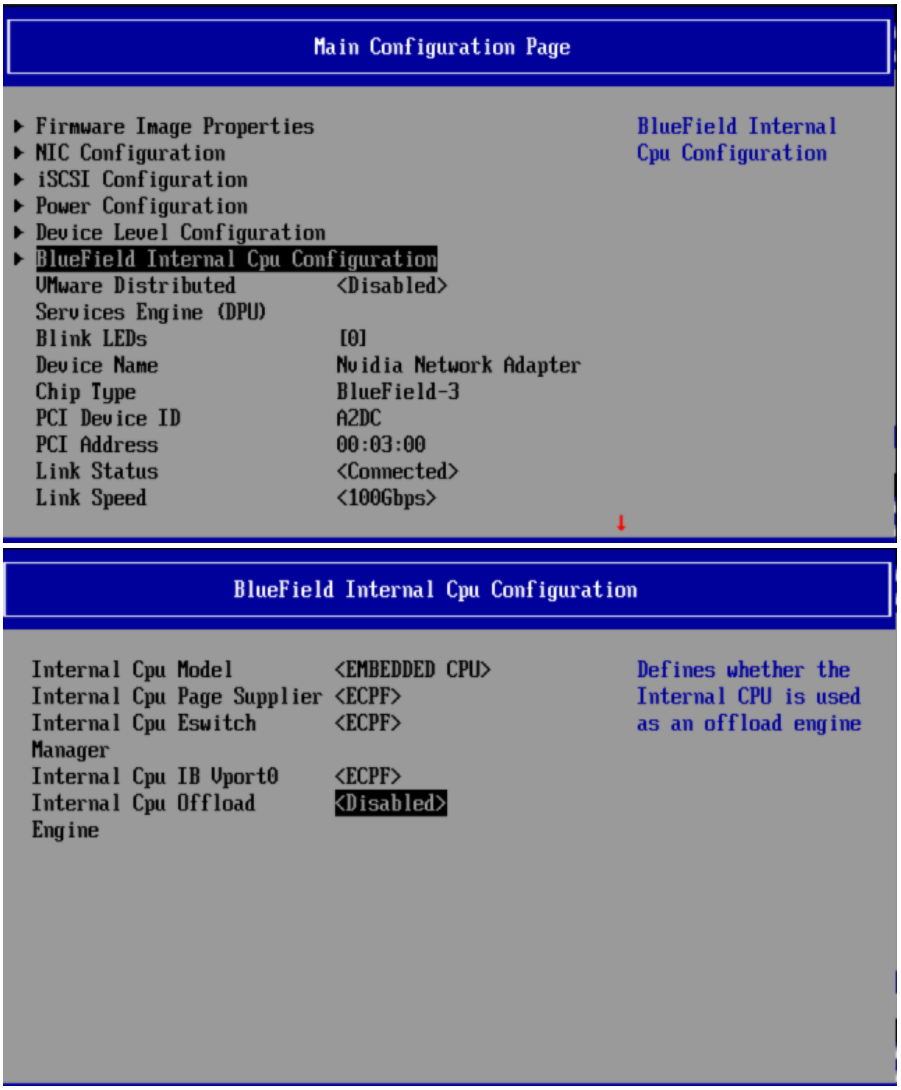
From Mode	To Mode	Command

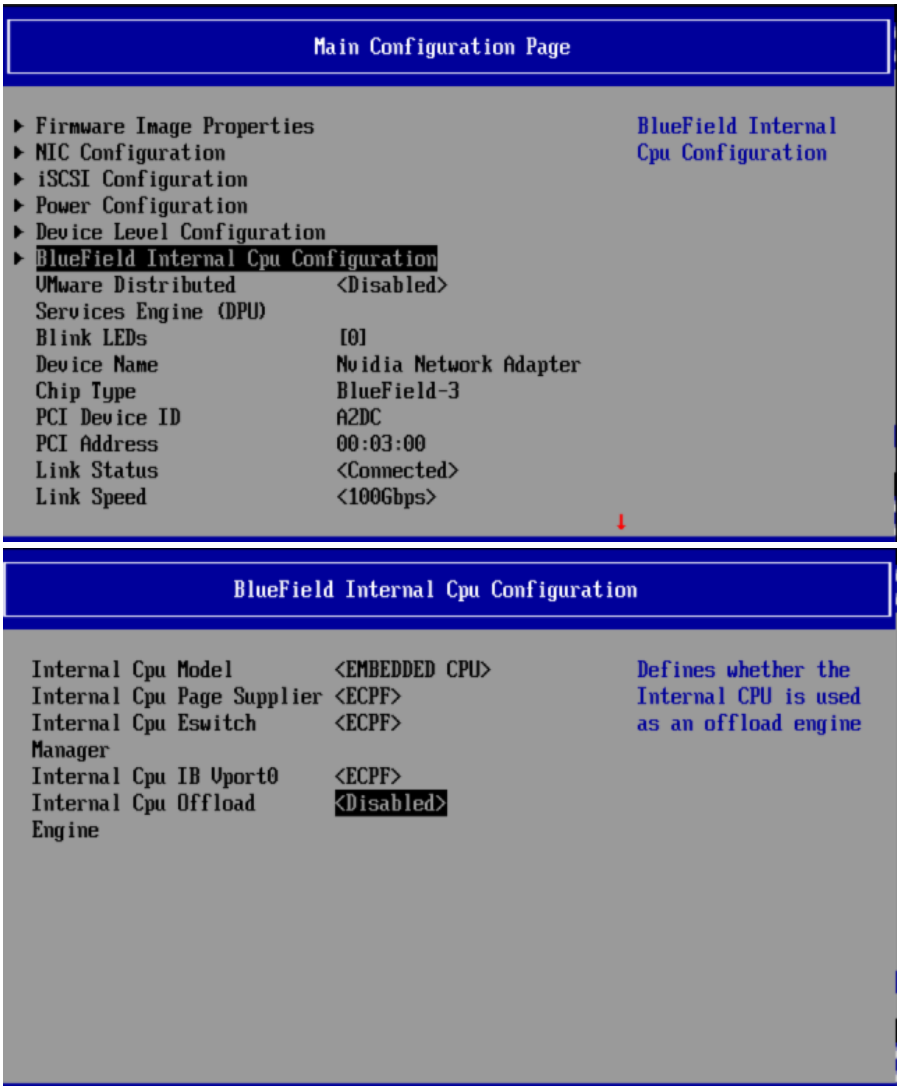
DPU Mode	NIC Mode	For BlueField-3:
		`host> sudo mlxconfig -d /dev/mst/<device> s INTERNAL_CPU_OFFLOAD_ENGINE=1`
		For BlueField-2:
		`host> sudo mlxconfig -d /dev/mst/<device> s INTERNAL_CPU_PAGE_SUPPLIER=1 INTERNAL_CPU_ESWITCH_MANAGER=1 INTERNAL_CPU_IB_VPORT0=1 INTERNAL_CPU_OFFLOAD_ENGINE=1`

NIC Mode	DPU Mode	For BlueField-3:
		`host> sudo mlxconfig -d /dev/mst/<device> s INTERNAL_CPU_OFFLOAD_ENGINE=0`
		For BlueField-2:
		`host> sudo mlxconfig -d /dev/mst/<device> s INTERNAL_CPU_PAGE_SUPPLIER=0 INTERNAL_CPU_ESWITCH_MANAGER=0 INTERNAL_CPU_IB_VPORT0=0 INTERNAL_CPU_OFFLOAD_ENGINE=0`

DPU Mode	DPU Mode with Zero Trust	N/A
DPU Mode with Zero Trust	DPU Mode	N/A
DPU Mode with Zero Trust	NIC Mode	Not supported. Move from DPU Mode with Zero Trust to DPU Mode first, and then from DPU Mode to NIC Mode. • Perform a system-level reset when moving from Zero Trust to DPU Mode before configuring NIC Mode. • Ensure to disable Zero Trust configuration before transitioning to NIC Mode. • Operating in NIC Mode with Zero Trust (Restricted) configuration is not supported and may lead to undefined behavior.

4.2.2.2.2 Using External Host UEFI Menu

From Mode	To Mode	Command
DPU Mode	NIC Mode	 Select **BlueField Internal Cpu Configuration**. To enable NIC Mode, set **Internal Cpu Offload Engine** to **Disabled**.

From Mode	To Mode	Command
NIC Mode	DPU Mode	 Select **BlueField Internal Cpu Configuration**. To enable DPU Mode, set **Internal Cpu Offload Engine** to **Enabled**.
DPU Mode	DPU Mode with Zero Trust	N/A
DPU Mode with Zero Trust	DPU Mode	N/A

From Mode	To Mode	Command
DPU Mode with Zero Trust	NIC Mode	Not supported. Move from DPU Mode with Zero Trust to DPU Mode first, and then from DPU Mode to NIC Mode. • Perform a system-level reset when moving from Zero Trust to DPU Mode before configuring NIC Mode. • Ensure to disable Zero Trust configuration before transitioning to NIC Mode. • Operating in NIC Mode with Zero Trust (Restricted) configuration is not supported and may lead to undefined behavior.

4.2.2.2.3 Using Arm OS Command Line

From Mode	To Mode	Command

DPU Mode	NIC Mode	For BlueField-3:
		`bf> sudo mlxconfig -d /dev/mst/<device> s INTERNAL_CPU_OFFLOAD_ENGINE=1`
		For BlueField-2:
		`bf> sudo mlxconfig -d /dev/mst/<device> s INTERNAL_CPU_PAGE_SUPPLIER=1 INTERNAL_CPU_ESWITCH_MANAGER=1 INTERNAL_CPU_IB_VPORT0=1 INTERNAL_CPU_OFFLOAD_ENGINE=1`

NIC Mode	DPU Mode	Not Applicable (Arm OS is not available)
DPU Mode	DPU Mode with Zero Trust	`bf> $ sudo mlxprivhost -d /dev/mst/<device> r --disable_rshim --disable_tracer --disable_counter_rd --disable_port_owner`
DPU Mode with Zero Trust	DPU Mode	`bf> $ sudo mlxprivhost -d /dev/mst/<device> p`
DPU Mode with Zero Trust	NIC Mode	Not supported. Move from DPU Mode with Zero Trust to DPU Mode first, and then from DPU Mode to NIC Mode. • Perform a system-level reset when moving from Zero Trust to DPU Mode before configuring NIC Mode. • Ensure to disable Zero Trust configuration before transitioning to NIC Mode. • Operating in NIC Mode with Zero Trust (Restricted) configuration is not supported and may lead to undefined behavior.

4.2.2.2.4 Using Arm UEFI Menu

From Mode	To Mode	Command
DPU Mode	NIC Mode	1. Access the Arm UEFI menu by pressing the Esc button twice on the console. 2. Select **Device Manager** → **System Configuration** → **BlueField Modes**. 3. Set the **NIC Mode** option to **NicMode** to enable NIC Mode. 4. Exit **BlueField Modes** and **System Configuration** and make sure to save the settings. 5. Exit the UEFI setup using the "reset" option. The configuration is not yet applied and BlueField is expected to boot in DPU Mode. 6. Issue power cycle to apply new configuration.
NIC Mode	DPU Mode	1. Access the Arm UEFI menu by pressing the Esc button twice on the console. 2. Select **Device Manager** → **System Configuration** → **BlueField Modes**. 3. Set the **NIC Mode** option to **DpuMode** to enable DPU Mode. 4. Exit **BlueField Modes** and **System Configuration** and make sure to save the settings. 5. Exit the UEFI setup using the "reset" option. The configuration is not yet applied and BlueField is expected to boot in NIC Mode. 6. Issue power cycle to apply new configuration.
DPU Mode	DPU Mode with Zero Trust	Roadmap
DPU Mode with Zero Trust	DPU Mode	Roadmap

From Mode	To Mode	Command
DPU Mode with Zero Trust	NIC Mode	Not supported. Move from DPU Mode with Zero Trust to DPU Mode first, and then from DPU Mode to NIC Mode. • Perform a system-level reset when moving from Zero Trust to DPU Mode before configuring NIC Mode. • Ensure to disable Zero Trust configuration before transitioning to NIC Mode. • Operating in NIC Mode with Zero Trust (Restricted) configuration is not supported and may lead to undefined behavior.

4.2.2.2.5 Using DPU-BMC Redfish

From Mode	To Mode	Command
DPU Mode	NIC Mode	`curl -k -u root:'password' -H 'content-type: application/json' -d '{"Mode": "NicMode"}' -X POST https://bmc_ip/redfish/v1/Systems/Bluefield/Oem/Nvidia/Actions/Mode.Set` Two consecutive Arm reboots are required to apply configuration.
NIC Mode	DPU Mode	`curl -k -u root:'password' -H 'content-type: application/json' -d '{"Mode": "DpuMode"}' -X POST https://bmc_ip/redfish/v1/Systems/Bluefield/Oem/Nvidia/Actions/Mode.Set` Two consecutive Arm reboots are required to apply configuration.
DPU Mode	DPU Mode with Zero Trust	`curl -k -u root:'password' -H 'content-type: application/json' -d '{"PrivilegeMode": "Restricted"}' -X PATCH https://<bmc_ip>/redfish/v1/Chassis/Card1/NetworkAdapters/NvidiaNetworkAdapter/Oem/Nvidia/HostPrivilegeConfig/Settings` Power cycle is required to apply configuration.
DPU Mode with Zero Trust	DPU Mode	`curl -k -u root:'password' -H 'content-type: application/json' -d '{"PrivilegeMode": "Privileged"}' -X PATCH https://<bmc_ip>/redfish/v1/Chassis/Card1/NetworkAdapters/NvidiaNetworkAdapter/Oem/Nvidia/HostPrivilegeConfig/Settings` Power cycle is required to apply configuration.
DPU Mode with Zero Trust	NIC Mode	Not supported. Move from DPU Mode with Zero Trust to DPU Mode first, and then from DPU Mode to NIC Mode. • Perform a system-level reset when moving from Zero Trust to DPU Mode before configuring NIC Mode. • Ensure to disable Zero Trust configuration before transitioning to NIC Mode. • Operating in NIC Mode with Zero Trust (Restricted) configuration is not supported and may lead to undefined behavior.

4.2.2.2.6 Using Platform-BMC NC-SI OEM Commands

<table>
<tr><th>From Mode</th><th>To Mode</th><th>Command</th></tr>
<tr><td>DPU Mode</td><td>NIC Mode</td><td>

Command = 0x12, Parameter = 0x33

<table>
<tr><th>Offset</th><th>31:24</th><th>23:16</th><th>15:8</th><th>7:0</th></tr>
<tr><td>0:15</td><td colspan="4">NC-SI OEM Header (OEM Command)</td></tr>
<tr><td>16:19</td><td colspan="4">Mellanox Manufacture ID (IANA) = 0x8119</td></tr>
<tr><td>20:23</td><td>Command rev=0x00</td><td>MLNX Cmd ID= 0x12</td><td>Parameter=0x33</td><td>Reserved</td></tr>
<tr><td>24:27</td><td colspan="3"></td><td>Offload engine</td></tr>
<tr><td>28:31</td><td colspan="4">Checksum 31:0</td></tr>
</table>

<table>
<tr><th>Field</th><th>Size</th><th>Offset in NC-SI Command</th><th>Description</th></tr>
<tr><td>Offload engine</td><td>1 bit</td><td>27.0</td><td>0x0: Enabled – DPU Mode
0x1: Disabled – NIC Mode</td></tr>
</table>

For NIC Mode, set the offload engine bit to 0x1.

</td></tr>
<tr><td>NIC Mode</td><td>DPU Mode</td><td>

Command = 0x12, Parameter = 0x33

<table>
<tr><th>bytes/bits</th><th>31:24</th><th>23:16</th><th>15:8</th><th>7:0</th></tr>
<tr><td>0:15</td><td colspan="4">NC-SI OEM Header (OEM Command)</td></tr>
<tr><td>16:19</td><td colspan="4">Mellanox Manufacture ID (IANA) = 0x8119</td></tr>
<tr><td>20:23</td><td>Command rev=0x00</td><td>MLNX Cmd ID= 0x12</td><td>Parameter=0x33</td><td>Reserved</td></tr>
<tr><td>24:27</td><td colspan="3"></td><td>Offload engine</td></tr>
<tr><td>28:31</td><td colspan="4">Checksum 31:0</td></tr>
</table>

<table>
<tr><th>Field</th><th>Size</th><th>Offset in NC-SI Command</th><th>Description</th></tr>
<tr><td>Offload engine</td><td>1 bit</td><td>27.0</td><td>0x0: Enabled - DPU Mode
0x1: Disabled - NIC Mode</td></tr>
</table>

For DPU Mode, set the offload engine bit to 0x0.

</td></tr>
</table>

From Mode	To Mode	Command
DPU Mode	DPU Mode with Zero Trust	Roadmap
DPU Mode with Zero Trust	DPU Mode	Roadmap
DPU Mode with Zero Trust	NIC Mode	Not supported. Move from DPU Mode with Zero Trust to DPU Mode first, and then from DPU Mode to NIC Mode. • Perform a system-level reset when moving from Zero Trust to DPU Mode before configuring NIC Mode. • Ensure to disable Zero Trust configuration before transitioning to NIC Mode. • Operating in NIC Mode with Zero Trust (Restricted) configuration is not supported and may lead to undefined behavior.

Separated Host Mode is obsolete for BlueField DPU/SuperNIC SKUs and should not be used, even if it remains visible in some configuration menus or options.

Separated Host Mode is available only for BlueField controller SKUs, where the BlueField Arm OS is the sole CPU/OS in the chassis.

4.3 DOCA Installation Guide for Linux

Follow the steps in this section to set up NVIDIA DOCA in your Linux environment.

4.3.1 Introduction

To deploy the BF-Bundle software on the BlueField networking platform (DPU or SuperNIC), you must:

1. Install or upgrade DOCA on the host side.
2. Install or upgrade BF-Bundle on the BlueField device.

4.3.2 Supported Networking Platforms

DOCA supports the following networking platforms:

- BlueField: refer to the BlueField hardware manuals for version-specific support information
- ConnectX: refer to the ConnectX hardware manuals for version-specific DOCA-Host support information

4.3.3 Hardware Prerequisites

For BlueField platform users, this guide assumes that a BlueField device has been installed in a server according to the instructions in your DPU's hardware user guide.

4.3.4 DOCA Packages

See information in the DOCA Release Notes.

4.3.5 Supported Host OS per DOCA-Host Installation Profile

Refer to Supported Host OS per DOCA-Host Installation Profile in the release notes to for OS-specific profile support information.

4.3.6 Setting Up Build Environment for Developers

For full instructions about setting up a development environment, refer to the DOCA Developer Guide.

4.3.7 Available BF-Bundle Installation and Upgrade Methods

Methods	Description	Purpose	Scope	Tool	Capability	Links
BFB image	BlueField bootstream (BFB) image for OS deployment on the BlueField DPUs	Production	All OSs	BFB installation (**install.sh**)	Installs OS, Kernel, and pre-built binaries	BF-Bundle Installation and Upgrade
Standard Linux tools	Enables upgrading DOCA components using standard Linux tools (e.g., **apt update** and **yum update**). This process utilizes native package manager repositories to upgrade DPUs without the need for a full installation.	Day 2/ Production	All OSs	Based on Linux tools (apt/ yum)	Updates components that include changes only	
ISO	ISO image	Installation	Ubuntu 22.04	PXE	Installs OS, Kernel, and pre-built binaries	Deploying BlueField Software Using ISO with PXE

Methods	Description	Purpose	Scope	Tool	Capability	Links
Bfb-build	Meant for BlueField users wishing to build their own BFB image for OS deployment on BlueField DPUs	To create custom BFB with special kernel and compile additional packages	All OSs	bfb-build	Installs OS, Kernel, and pre-built binaries	Bfb-build github
BF-FW bundle	A firmware-only BFB image which includes UEFI, NIC firmware, and BMC firmware	Upgrade only firmware components	Only firmware	BFB installation (`install.sh`)	Upgrade firmware components only	NVIDIA DOCA Downloads BF-FW Bundle

4.3.8 DOCA-Host Installation and Upgrade

The following options are available for installing or upgrading DOCA-Host:

Method	Option Name	Description
Installation	DOCA local repo package	This option installs DOCA components
Upgrade	DOCA online repo	This option upgrades DOCA components using online repo from the Internet

Find the relevant DOCA installation files for your host in the NVIDIA DOCA Downloads page.

This page provides basic instructions for setting up DOCA on a standard system. For device-specific setups, refer to the respective device documentation, as additional steps may be required.

4.3.8.1 Uninstalling Software from Host

If DOCA-Host version 2.6.0 or greater is installed you can refer to "DOCA-Host Installation and Upgrade#Upgrading Software on Host" section.

Full uninstallation is required before installing DOCA-Host:

Deb-based

```
host# for f in $( dpkg --list | grep -E 'doca|flexio|dpa-gdbserver|dpa-stats|dpa-resource-mgmt|dpaeumgmt|
dpdk-community' | awk '{print $2}' ); do echo $f ; sudo apt remove --purge $f -y ; done
host# sudo /usr/sbin/ofed_uninstall.sh --force
host# sudo apt-get autoremove
```

RPM-based

```
host# for f in $(rpm -qa | grep -i doca ) ; do sudo yum -y remove $f; done
host# sudo /usr/sbin/ofed_uninstall.sh --force
host# sudo yum autoremove
host# sudo yum makecache
```

Installation and upgrade requirements:

- Installed kernel headers that match the version of the currently running kernel.

  If the **build** directory exists in under **/lib/modules/$(uname -r)/build**, then the kernel headers are installed.

- The same gcc version that the currently running kernel was built with.

4.3.8.2 Installing Software on Host

1. Install DOCA local repo package for host:
   - For deb-based distributions:
   - i. Download the DOCA host repo from the NVIDIA DOCA Downloads page.
     ii. Install the deb repo:

     ```
     host# sudo dpkg -i <repo_file>.deb
     ```

     iii. Run the update command:

     ```
     host# apt-get update
     ```

     iv. Install any of the profiles supported for your OS and architecture. Refer to "Supported Host OS per DOCA-Host Installation Profile" (e.g., **doca-all**) for more information.

     ```
     host# sudo apt install -y doca-all
     ```

     v. Update the firmware:

     ```
     host# sudo apt install -y mlnx-fw-updater
     ```

   - For RPM-based distributions:

     Required

     DKMS ≥ 3.2 must be installed on the system as a package.

     i. Download the DOCA host repo from the NVIDIA DOCA Downloads page.
     ii. Install the rpm repo:

     ```
     host# sudo rpm -Uvh <repo_file>.rpm
     ```

     iii. For non-SLES systems, run:

```
host# sudo yum makecache
```

iv. For SLES systems, run the following to verify the package signing:

```
host# zypper --gpg-auto-import-keys refresh
```

> Continue the procedure using **zypper** instead of **yum**.

v. Install any of the profiles supported for your OS and architecture (e.g., doca-all):

```
host# yum/zypper install -y doca-all
```

vi. Update the firmware:

```
host# yum/zypper install -y mlnx-fw-updater
```

> **Required**
>
> If you are running on a secure-boot-enabled OS, please refer to "Kernel Module Signing" before continuing.

2. Load the drivers:

```
host# sudo /etc/init.d/openibd restart
```

3. Initialize MST:

```
host# sudo mst restart
```

4.3.8.3 Upgrading Software on Host

To update existing DOCA host packages, follow these steps:

1. Install the newer version of the DOCA host repository as detailed in section "DOCA-Host Installation and Upgrade#Installing Software on Host".
2. Upgrade the DOCA packages:
   - To upgrade from DOCA version 2.5.x, all DOCA and OFED related packages should be removed. For detailed instructions on how to uninstall these packages, please refer to the uninstall section "DOCA-Host Installation and Upgrade#Uninstalling Software from Host".
   - To upgrade from DOCA 2.6.0 or later, use the following command:
     - For DEB-based distributions:

```
host# apt install <doca-profile>
```

     - For RPM-based distributions:

```
host# yum upgrade <doca-profile>
```

- For SLES systems:

```
host# zypper update <doca-profile>
```

- Before upgrading **mlnx-fw-updater**, make sure to restart MST first:

```
host# mst restart
```

4.3.8.4 Package and Kernel Module Signing

This section highlights the signing methods used to ensure the authenticity and integrity of the packages and kernel modules.

4.3.8.4.1 Package Signing

RPM packages are signed with a GPG key, enabling users to confirm that the packages come from a trusted NVIDIA source and have not been altered.

During installation, package managers automatically fetch and verify the signature while updating their metadata cache.

> For more details, see DOCA-Host Installation and Upgrade#Installing Software on Host.

4.3.8.4.2 Kernel Module Signing

With the integration of DKMS (Dynamic Kernel Module Support), DOCA-Host no longer includes pre-built, Nvidia-signed kernel drivers. Instead, the DOCA-Host installation process uses DKMS to automatically build and sign the kernel modules using a locally generated key pair.

To load these drivers on a secure-boot-enabled OS, the DKMS-generated public key must be enrolled in the system's key database. This enrollment is performed after the DOCA-Host installation is complete:

1. Import the public key to the Machine Owner Key (MOK) list using the **mokutil** utility:

```
# mokutil --import /var/lib/dkms/mok.pub
```

2. You are prompted to create and confirm a password for this one-time enrollment request.
3. Reboot the system.
4. During the boot process, follow the on-screen prompts from the MOK utility to enroll the key. You will need the password created in step 2.

4.3.8.5 DOCA Extra Package and doca-kernel-support

doca-kernel-support is a script intended to rebuild kernel modules included with DOCA-host. It builds a single package (i.e., **rpm** or **deb**) that contains a repository of packages (i.e., **dnf** or **apt**).

> Rebuilt kernel modules with **doca-kernel-support** are not signed by default. To sign them, follow the note in step 2 below.

DOCA-host provides binary builds of kernel modules for some specific kernels. This script rebuilds kernel modules included with DOCA-host for a custom kernel version and creates an RPM or DEB package that holds all of those rebuilt modules for easy install.

> **doca-kernel-support** does not support customized or unofficial kernels.

> **doca-kernel-support** is no longer required when using non-default kernels. DKMS rebuilds kernel modules automatically during the regular installation process.

To run **doca-kernel-support**:

1. Run:

   ```
   host# apt/yum install -y doca-extra
   ```

2. Execute the **doca-kernel-support** script which rebuilds and installs the DOCA-host kernel modules with the running kernel:

   ```
   host# /opt/mellanox/doca/tools/doca-kernel-support
   ```

   > To sign the kernel modules, run this command instead:
   >
   > ```
   > host# MODULE_SIGN_PUB_KEY=/path/to/pub_key.der MODULE_SIGN_PRIV_KEY=/path/to/priv_key.priv
   > WITH_MOD_SIGN=1 /opt/mellanox/doca/tools/doca-kernel-support
   > ```

   The output may end with **rpm** or **deb** with the following general format:

   ```
   doca-kernel-support: Built single package: <doca-kernel-repo>
   doca-kernel-support: Done
   ```

   Output example:

   ```
   doca-kernel-support: Built single package: /tmp/DOCA.EuUfkWfV7Z/doca-kernel-
   repo-2.9.0-1.kver.5.14.0.356.el9.x86.64.x86_64.rpm
   doca-kernel-support: Done
   ```

   > **doca-kernel-support** does not support customized or unofficial kernels.

3. Install the generated meta-package on the desired DOCA profile:

   > The examples provided install DOCA kernel repos (RPM/DEB) with **doca-ofed** profile, but other profiles may be installed if they are supported.

   - For SLES:

```
host# rpm -Uvh <doca-kernel-repo>
host# zypper refresh
host# zypper install <doca-userspace-metapackage>
host# zypper modifyrepo --disable doca
host# zypper install doca-kernel-KERNEL_VERSION     # For all the meta-package, not just this
 single package
host# zypper modifyrepo --enable doca
```

Example:

```
host# rpm -Uvh <doca-kernel-repo>
host# zypper refresh
host# zypper install doca-ofed-userspace
host# zypper modifyrepo --disable doca
host# zypper install doca-kernel-5.14.21.150500.53.default     # For all the meta-package, not just
this single package
host# zypper modifyrepo --enable doca
```

- For RPM-based distributions:

```
host# rpm -Uvh <doca-kernel-repo>
host# dnf makecache
host# dnf install <doca-userspace-metapackage>
host# dnf install --disablerepo=doca doca-kernel-KERNEL_VERSION     # For all the meta-package, not
just this single package
```

Example:

```
host# rpm -Uvh /tmp/DOCA.EuUfkWfV7Z/doca-kernel-repo-2.9.0-1.kver.5.14.0.356.el9.x86.64.x86_64.rpm
host# dnf makecache
host# dnf install doca-ofed-userspace
host# dnf install --disablerepo=doca doca-kernel-5.14.0.356.el9.x86.64.x86_64.noarch
```

- For DEB-based distributions:

```
host# dpkg -i <doca-kernel-repo>
host# apt update
host# apt install <doca-profile>
```

Example:

```
host# dpkg -i /tmp/DOCA.J8JYxEmffD/doca-kernel-repo-2.9.0-6.4.0.mlnx_2.9.0_amd64.deb
host# apt update
host# apt install doca-ofed
```

4. (Optional) Retrieve installed packages and their versions as part of DOCA Host installation:

```
host# /opt/mellanox/doca/tools/doca-info

Versions:
- DOCA Base MLNX_OFED_LINUX-24.07-0.5.5.0
- MFT 4.29.0-127

UEFI\ATF versions:
- mst_device: mt41692_pciconf0
     UEFI Version: 4.7.0-42-g13081ae
     ATF Version: 4.7.0-25-g5569834

Firmware (Current):
- BlueField-3 32.41.1000

DOCA:
- doca-all 2.8.0-0.0.4
- doca-apsh-config 2.8.0079-1
- doca-bench 2.8.0079-1
…

DOCA Dependencies:
…
- flexio 24.07.2300
- mlnx-dpdk 22.11.0-2407.0.10


OFED:
…
- rdma-core 2407mlnx52-1.2407055
…
- ucx 1.17.0-1.2407055
…
```

If BlueField has a BF-Bundle version older than 2.7.0 installed on it, the UEFI/ATF versions would appear as N/A. If your version is 2.7.0 or higher and still see N/A, then perform driver restart on the host:

```
/etc/init.d/openibd restart
```

4.3.8.6 Storage Installation

1. Before installation, search for the required package using your system's package manager:

```
# apt/zypper/dnf/yum search mlnx-nvme
# apt/zypper/dnf/yum search mlnx-nfsrdma
```

2. Once the correct package name is identified, install it using:

```
# apt/zypper/dnf/yum install <full-package-name>
```

Installation examples:

- For SLES:

```
# zypper search mlnx-nvme
# zypper install mlnx-nvme-kmp-default
```

- For Debian-based systems:

```
# apt search mlnx-nvme
# apt search mlnx-nfsrdma
# apt install mlnx-nvme-dkms
# apt install mlnx-nfsrdma-dkms
```

4.3.8.6.1 VMA Installation

VMA is compatible with the latest DOCA-Host drivers, firmware, and libraries.

For users looking to install or upgrade VMA, DOCA provides **doca-libvma**, a meta-package that installs the driver stack, libvma, and all necessary dependencies.

To install **doca-libvma**, follow the instructions under section "DOCA-Host Installation and Upgrade#Installing Software on Host" and substitute **doca-all** with **doca-libvma** in the following command:

```
# Original command
sudo <apt/yum/zypper> install -y doca-all
# Updated command
sudo <apt/yum/zypper> install -y doca-libvma
```

For the list of systems required to work with VMA, refer to the VMA documentation.

4.3.8.6.2 XLIO Installation

XLIO is compatible with the latest DOCA-Host drivers, firmware, and libraries.

For users looking to install or upgrade XLIO, DOCA **doca-libxlio**, a meta-package that installs the driver stack, libxlio, and all necessary dependencies.

To install **doca-libxlio**, follow the instructions under section "DOCA-Host Installation and Upgrade#Installing Software on Host" while substituting **doca-all** with **doca-libxlio** in the following command:

```
# Original command
sudo <apt/yum/zypper> install -y doca-all
# Updated command
sudo <apt/yum/zypper> install -y doca-libxlio
```

For the list of systems required to work with XLIO, please see the XLIO documentation.

4.3.8.7 Installing Proprietary Packages

The installation procedure does not install proprietary packages. Those packages are installed upon request.

List of close-source proprietary packages:

- Clusterkit
- DPCP
- hcoll
- sharp
- ibutils2
- opensm

Currently, the only way to install these packages is by using an already-built RPM or DEB file from a similar primary OS.

The following table maps community OSs which are most similar to primary OSs:

Community OS	Most Similar Primary OS
Alma 8.5	RHEL 8.5
Anolis OS 8.4	RHEL 8.5
CentOS Stream 8	RHEL 8
CentOS Stream 9	RHEL 9
EulerOS-V2.0.SP10	EulerOS-V2.0.SP11
Fedora 35	RHEL 8.5
OpenEuler-20.03.SP1	OpenEuler20 SP3
OpenSUSE 15.3	SLES15 SP3
Photon OS 3.0	RHEL 7.9
UOS-V20-1040d	Debian 10.8

4.3.8.7.1 Download and Install MFT and Firmware

- Download MFT

- Download firmware

4.3.8.8 DOCA-Host Installation and DKMS Management Guide

4.3.8.8.1 Introduction

DOCA-Host is a host software stack that provides kernel drivers, user space libraries, and management tools for NVIDIA networking products, including ConnectX and BlueField adapters. It ensures high performance, scalability, and compatibility across various operating systems and applications.

The DOCA-Host stack utilizes DKMS (Dynamic Kernel Module Support) to build and install kernel modules directly from source on the host machine.

The changes in architecture include the following points:

- Source-based build – The DOCA repository no longer contains prebuilt kernel drivers; instead, modules are compiled during installation via DKMS.
- Repository structure – NVIDIA no longer publishes repositories for minor OS releases (e.g., RHEL 9.3). Users must now utilize repositories published per Major OS release (e.g., RHEL 9.x), allowing DKMS to handle kernel differences.
- Module signing – Because modules are built locally, they are not signed by NVIDIA.

4.3.8.8.2 Understanding DKMS

DKMS is a framework that enables the automatic rebuilding and reinstallation of Linux kernel modules whenever the host server's kernel version is updated.

DKMS offers the following benefits:

- DKMS creates a trigger that compiles drivers against new kernel headers immediately upon update installation.
- This eliminates the need for specific binaries for every minor OS kernel update or manual source recompilation after system updates.

4.3.8.8.3 Supported Platforms and Profiles

The following operating systems support the DOCA-Host DKMS implementation:

- RHEL (via **doca-all**)
- Ubuntu (via **doca-all**)
- Oracle (via **doca-all**)
- Debian (via **doca-all**)
- SLES (via **doca-ofed**)

This architecture applies to all DOCA-Host profiles that include kernel components:

- **doca-roce**
- **doca-ofed**
- **doca-networking**
- **doca-all**

⚠ User-space–only metapackages are not affected by these changes

4.3.8.8.4 Installation Procedure

Select the repository corresponding to the Major OS version. For example, installations on RHEL 9.4 or RHEL 9.6 must use the RHEL 9 repository.

The following steps outline the installation process using Rocky Linux 9.6 as an example target, utilizing the RHEL 9 repository:

1. If DOCA-Host or MLNX-OFED packages are currently installed, they must be removed prior to installation to prevent conflicts.

```
host# for f in $(rpm -qa | grep -i doca ) ; do sudo yum -y remove $f; done
host# sudo /usr/sbin/ofed_uninstall.sh --force
host# sudo yum autoremove
host# sudo yum makecache
```

2. You must install DKMS and the kernel headers matching your currently running kernel.
   a. Enable EPEL (if required for `dkms` availability):

```
sudo dnf install -y epel-release
```

   b. Install build tools and headers:

```
sudo dnf install -y dkms gcc make perl mokutil kernel-devel-$(uname -r) kernel-headers-$(uname -r)
```

3. Download and install the Major Version repository RPM (e.g., RHEL 9):

```
sudo rpm -Uvh <DOCA_HOST_REPO_RPM_FOR_RHEL9>
sudo dnf makecache
```

4. Install the desired stack (e.g., `doca-all`) and the firmware updater:

```
sudo dnf install -y doca-all mlnx-fw-updater
```

4.3.8.8.5 Secure Boot and Package Signing

Standard installation packages (RPM/DEB) are signed to verify they originate from NVIDIA.

Kernel modules built via DKMS are not NVIDIA-signed. For systems with Secure Boot enabled, DKMS utilizes a locally generated key pair to sign the built modules.

To ensure the modules load correctly on Secure Boot systems, the public key must be enrolled via the Machine Owner Key (MOK) utility:

1. Import the DKMS public key:

```
sudo mokutil --import /var/lib/dkms/mok.pub
```

2. Reboot the server:

```
sudo reboot
```

3. Complete the enrollment process via the MOK menu during the system boot sequence.

4.3.9 BF-Bundle Installation and Upgrade

Users have three options for installing or upgrading DOCA on BlueField DPU or SuperNIC:

Method	Option Name	Description
Installation	BF-bundle DOCA image	This option overwrites the entire boot partition with an Ubuntu 22.04 installation and updates BlueField and NIC firmware
Upgrade	Standard Linux Tools	This option upgrades DOCA components without overwriting the boot partition. Use this option to preserve configurations or files on BlueField itself
Installation	ISO DOCA image	This option overwrites the entire boot partition with an Ubuntu 22.04 installation and updates BlueField and NIC firmware

Find the DOCA installation files for BlueField on the NVIDIA DOCA Downloads page.

4.3.9.1 Installing DOCA-Host

Make sure to update DOCA on the host side before installing the BF-bundle on the BlueField.

4.3.9.2 Installing Software on BlueField Using BF-Bundle

The ATF will not boot 150W BlueField-3 platforms if the ATX +12V power supply is not connected. This requirement ensures the proper operation of the BlueField device. For detailed instructions on connecting the external power supply connector, refer to the NVIDIA BlueField-3 Networking Platform User Guide.

Users can install DOCA on BlueField DPU or SuperNIC by upgrading the full DOCA image on the BlueField device. This process overwrites the entire boot partition with an Ubuntu 22.04 installation and updates BlueField and NIC firmware. It also sets up the OVS bridge.

Upgrading DOCA components on BlueField networking platforms, such as DPUs or SuperNICs, using standard Linux tools will install the new **doca-eula** package at **/etc/NVIDIA/doca-license/DOCA-EULA.txt**. By installing this package, you accept the NVIDIA DOCA SDK end-user license agreement.

4.3.9.2.1 Option 1 – No Pre-defined Password

To change the default Ubuntu password during the BFB bundle installation, proceed to Option 2.

To install a BFB, use the following command:

```
host# sudo bfb-install --rshim rshim<N> --bfb <image_path.bfb>
```

Where:

- **<N>** – the appropriate RShim device identifier. If you have only one BlueField device, use **rshim0** .
- **<image_path.bfb>** – Replace this with the path to the BFB image file.

To list the RShim devices present on your system, run the following command:

```
host# ls -la /dev/ | grep rshim
```

⚠ By default, **bfb-install** will clear the RShim log in **/dev/rshim<N>/misc** and save it as **tmp/bfb-install-rshim[N].log** instead. To preserve the RShim log in **/dev/rshim<N>/misc** , provide the **--keep-log** argument to the **bfb-install** command line.

4.3.9.2.2 Option 2 – Set Pre-defined Password

Ubuntu users can set a unique password for the **ubuntu** user in the **bf.cfg** configuration file, which will be applied automatically at the end of the BlueField BFB bundle installation.

To do this, follow these steps:

1. Generate the hashed password:

   ```
   host# openssl passwd -1
   Password:
   Verifying - Password:
   $1$3B0RIrfX$TlHry93NFUJzg3Nya00rE1
   ```

2. Add the password hash to the **bf.cfg** configuration file:

   ```
   host# echo ubuntu_PASSWORD='$1$3B0RIrfX$TlHry93NFUJzg3Nya00rE1' > bf.cfg
   ```

3. Use the **--config** flag to specify the configuration file when running the installation command:

   ```
   host# sudo bfb-install --rshim rshim<N> --bfb <image_path.bfb> --config bf.cfg
   ```

   ⚠ If **--config** is not used, then upon first login to the BlueField device, users will be prompted to update the default **ubuntu** password.

   ⚠ Optionally, to upgrade the BlueField integrated BMC firmware using BFB bundle, please provide the current BMC root credentials in a **bf.cfg** file, as shown in the following:

   ```
   BMC_PASSWORD="<root password>"
   BMC_USER="root"
   ```

```
BMC_REBOOT="yes"
```

Unless previously changed, the default BMC root password is **OpenBmc**.

The following is an example output of Ubuntu-22.04 BFB bundle installation:

```
host# sudo bfb-install --rshim rshim0 --bfb bf-bundle-2.7.0_24.04_ubuntu-22.04_prod.bfb --config bf.cfg
Pushing bfb 1.41GiB 0:02:02 [11.7MiB/s]
[           <=>                                 ]
Collecting BlueField booting status. Press Ctrl+C to stop
 INFO[PSC]: PSC BL1 START
 INFO[BL2]: start
 INFO[BL2]: boot mode (rshim)
 INFO[BL2]: VDDQ: 1120 mV
 INFO[BL2]: DDR POST passed
 INFO[BL2]: UEFI loaded
 INFO[BL31]: start
 INFO[BL31]: lifecycle GA Secured
 INFO[BL31]: VDD: 850 mV
 INFO[BL31]: runtime
 INFO[BL31]: MB ping success
 INFO[UEFI]: eMMC init
 INFO[UEFI]: eMMC probed
 INFO[UEFI]: UPVS valid
 INFO[UEFI]: PMI: updates started
 INFO[UEFI]: PMI: total updates: 1
 INFO[UEFI]: PMI: updates completed, status 0
 INFO[UEFI]: PCIe enum start
 INFO[UEFI]: PCIe enum end
 INFO[UEFI]: UEFI Secure Boot
 INFO[UEFI]: PK configured
 INFO[UEFI]: Redfish enabled
 INFO[UEFI]: exit Boot Service
 INFO[MISC]: Found bf.cfg
 INFO[MISC]: Ubuntu installation started
 INFO[MISC]: Installing OS image
 INFO[MISC]: Changing the default password for user ubuntu
 INFO[MISC]: Ubuntu installation completed
 INFO[MISC]: Updating NIC firmware...
 INFO[MISC]: NIC firmware update done
 INFO[MISC]: Installation finished
```

4. To verify that the BlueField device has completed booting, wait 90 seconds after the installation process finishes and run the following command:

```
host# sudo cat /dev/rshim<N>/misc
...
 INFO[MISC]: Linux up
 INFO[MISC]: DPU is ready
```

5. Retrieve installed packages and their versions as part of BF-Bundle installation:
    a. Log in to BlueField.
    b. Run the following command:

```
bf# sudo bf-info
```

Example output:

```
Firmware:
- ATF: v2.2(release):4.11.0-31-g3cc9f6506
- UEFI: 4.11.0-44-gb67dfb4a53
- BSP: 4.11.0.13582
- NIC Firmware: 32.45.0350
- BMC Firmware: 25.04-3
- CEC Firmware: 00.02.0195.0000

Drivers:
- mlnx-dpdk:  'MLNX_DPDK 22.11.2504.0.2'
- Kernel: 5.15.0-1064-bluefield

Tools:
- MFT: 4.32.0-94
- mstflint: 4.29.0-1

Storage:
- mlnx-libsnap 1.6.0-1
- spdk 23.01.5-24
- virtio-net-controller 24.10.15-1

DOCA:
- doca-apsh-config 2.9.0064-1
- libdoca-sdk-urom-dev 2.9.0064-1
...
```

```
FlexIO:
- flexio-samples 24.10.2447
- flexio-sdk 24.10.2447
...
SoC Platform:
- mlxbf-gige-modules 1.0-0.kver.6.1.0-11-arm64
- sdhci-of-dwcmshc-modules 1.0-0.kver.6.1.0-11-arm64
...
OFED:
rdma-core 2410mlnx54-1.2410051
ucx 1.18.0-1.2410051
...
```

6. Run the following command to configure the **tmfifo_net0** interface over IPv4 for SSH access to the BlueField Arm OS:

```
host# ifconfig tmfifo_net0 192.168.100.1/24
```

Establish an SSH connection to the BlueField Arm OS with 192.168.100.2 (preconfigured default) using the following command:

```
host# ssh ubuntu@192.168.100.2
```

4.3.9.3 Upgrading BlueField Using Standard Linux Tools

This procedure allows for upgrading DOCA components on BlueField networking platforms (DPUs or SuperNICs) using standard Linux tools (e.g., **apt update** and **yum update**). This method uses native package manager repositories to update components without requiring a full reinstallation.

Important!

Although this method allows you to update specific DOCA components, the combinations resulting from selective updates are not validated by NVIDIA. NVIDIA only validates the full installation of the BF-Bundle package.

Upgrading DOCA components on BlueField networking platforms using standard Linux tools will install the new **doca-eula** package located at **/etc/NVIDIA/doca-license/DOCA-EULA.txt**. By installing this package, you accept the NVIDIA DOCA SDK end-user license agreement.

This process provides the following capabilities and benefits:

- Selective updates: Only updates components that include modifications.
- Flexible upgrade paths: You must choose the upgrade option that best fits your deployment needs:
  - DOCA software only: Updates DOCA drivers, libraries, services, and reference applications without upgrading the hardware firmware.
  - DOCA software and firmware: Updates all DOCA software components alongside the BSP (UEFI/ATF), NIC firmware, and BMC components, while maintaining existing configurations.

> Firmware may also be updated independently through other supported methods, such as via the out-of-band (OOB) DPU-BMC in a deferred flow.

- Preserves environment: Does not impact user binaries or upgrade non-Ubuntu OS kernels.
- Post-upgrade activation:
  - If the NIC firmware was not updated, perform a BlueField Arm reset (software reset/reboot).
  - If the NIC firmware was updated, perform a firmware reset (**mlxfwreset**) or a graceful shutdown and power cycle.

4.3.9.3.1 Upgrading DEB-based Distributions

Follow this procedure to upgrade DOCA components on DEB-based BlueField environments, such as Ubuntu, using the APT package manager.

4.3.9.3.1.1 Repository Setup

1. Export **DOCA_REPO** with the relevant URL:

```
<bf> $ export DOCA_REPO="<URL>"
```

   - Latest GA: **https://linux.mellanox.com/public/repo/doca/latest/ubuntu22.04/dpu-arm64**
   - Latest-3.2-LTS: **https://linux.mellanox.com/public/repo/doca/latest-3.2-LTS/ubuntu22.04/dpu-arm64**
   - Latest-2.9-LTS: **https://linux.mellanox.com/public/repo/doca/latest-2.9-LTS/ubuntu22.04/dpu-arm64**
   - Latest-2.5-LTS: **https://linux.mellanox.com/public/repo/doca/latest-2.5-LTS/ubuntu22.04/dpu-arm64**

2. Add the GPG key to the APT trusted keyring:

```
<bf> $ curl $DOCA_REPO/nvidia-doca-debian-gpg-public-key.asc | gpg --dearmor > /etc/apt/trusted.gpg.d/
nvidia-doca-debian-gpg-public-key.gpg
```

3. Add the DOCA online repository and update the index:

```
<bf> $ echo "deb [signed-by=/etc/apt/trusted.gpg.d/nvidia-doca-debian-gpg-public-key.gpg] $DOCA_REPO ./"
 > /etc/apt/sources.list.d/doca.list
```

4. Update the index:

```
<bf> $ apt update
```

4.3.9.3.1.2 Package Upgrade Selection

Choose one of the following options to upgrade your packages.

Option 1: Upgrade BlueField FW Bundle

This command upgrades the ATF-UEFI, NIC firmware, and BMC firmware only.

```
<bf> $ apt install bf-fwbundle
```

Option 2: Update Specific DOCA Components

To skip the firmware update (e.g., when updating via DPU-BMC or Redfish), remove the old firmware packages:

- If your DOCA version is <3.4.0:

```
<bf> $ apt remove bf2-bmc-fw-signed bf3-bmc-fw-signed bf3-bmc-gi-signed bf3-bmc-nic-fw* bf2-cec-fw-signed bf3-cec-fw-signed
```

- If your DOCA version is ≥3.4.0:

```
<bf> $ apt remove bf2-bmc-fw-signed bf3-bmc-fw-signed bf3-bmc-gi-signed bf3-bmc-nic-fw* bf2-cec-fw-signed bf3-cec-fw-signed mlxbf-bootimages-signed
```

1. Perform the OS upgrade:
   - For non-Ubuntu OS, run the standard upgrade command:

```
<bf> $ apt upgrade
```

   - For Ubuntu OS, check the current kernel version (**uname -r**). If it matches the target upgrade version, simply run **apt upgrade** . If it does not match, you must transition the kernel manually:
     i. Install the new kernel:

```
<bf> $ apt install linux-image-<new> linux-modules-<new> linux-modules-extra-<new> linux-headers-<new>
```

     ii. Remove the old kernel:

```
<bf> $ apt purge linux-image-<old> linux-modules-<old> linux-modules-extra-<old> linux-headers-<old>
```

     iii. Cleanup and refresh GRUB:

```
<bf> $ apt autoremove --purge && update-grub
```

     iv. Ensure sshpass is installed (required for component activation):

```
<bf> $ command -v sshpass >/dev/null 2>&1 || { apt-get update && apt-get install -y sshpass; }
```

     v. Perform the final system upgrade:

```
<bf> $ apt upgrade
```

4.3.9.3.1.3 Firmware Configuration and Activation

Skip this phase if you do not wish to update firmware.

1. Configure BMC parameters by adding the BMC user and password to **/etc/bf-upgrade.conf** (format is identical to **bf.cfg**):

```
<bf> $ cat > /etc/bf-upgrade.conf << EOF
BMC_USER="<username>"
BMC_PASSWORD="<password>"
WITH_NIC_FW_UPDATE="yes"
BMC_REBOOT="yes"
EOF
```

> **BMC_REBOOT="yes"** causes a warm boot to the BMC, temporarily losing connectivity.

2. Activate the components:

```
<bf> $ dpu-bmc-upgrade
```

> The upgrade process takes up to 20 minutes. The script performs an **fwreset** if supported.

4.3.9.3.1.4 Apply Changes

For the upgrade to take effect, perform a BlueField system reboot to apply the new changes, NIC firmware, and UEFI/ATF.

> This step triggers immediate reboot of the BlueField Arm cores.

4.3.9.3.2 Upgrading RPM-based Distributions

Follow this procedure to upgrade DOCA components on RPM-based BlueField environments using the DNF or YUM package managers.

4.3.9.3.2.1 Repository Setup

1. Export **DOCA_REPO** with the relevant URL:

```
<bf> $ export DOCA_REPO="<URL>"
```

   - Latest GA: **https://linux.mellanox.com/public/repo/doca/latest/openeuler22.03sp3/dpu-arm64/**
   - Latest-3.2-LTS: **https://linux.mellanox.com/public/repo/doca/latest-3.2-LTS/openeuler22.03sp3/dpu-arm64/**
   - Latest-2.9-LTS: **https://linux.mellanox.com/public/repo/doca/latest-2.9-LTS/openeuler22.03sp3/dpu-arm64/**
   - Latest-2.5-LTS: **https://linux.mellanox.com/public/repo/doca/latest-2.5-LTS/anolis8.6/dpu-arm64/**

2. Add the DOCA online repository:

```
<bf> $ echo "[doca]
name=DOCA Online Repo
baseurl=$DOCA_REPO
enabled=1
gpgcheck=0
repo_gpgcheck=1
gpgkey=$DOCA_REPO/nvidia-doca-rpm-gpg-public-key.asc
priority=10
cost=10" > /etc/yum.repos.d/doca.repo
```

3. Update the index:

```
<bf> $ yum makecache
```

4.3.9.3.2.2 Package Upgrade Selection

Choose one of the following options to upgrade your packages.

Option 1: Upgrade BlueField FW Bundle

This command upgrades the ATF-UEFI, NIC firmware, and BMC firmware only.

```
<bf> $ dnf install bf-fwbundle
```

Option 2: Update Specific DOCA Components

To skip the firmware update (e.g., when updating via DPU-BMC or Redfish), remove the old firmware packages:

- If your DOCA version is <3.4.0:

```
<bf> $ dnf remove bf2-bmc-fw-signed bf3-bmc-fw-signed bf3-bmc-gi-signed bf3-bmc-nic-fw* bf2-cec-fw-signed bf3-cec-fw-signed
```

- If your DOCA version is ≥3.4.0:

```
<bf> $ dnf remove bf2-bmc-fw-signed bf3-bmc-fw-signed bf3-bmc-gi-signed bf3-bmc-nic-fw* bf2-cec-fw-signed bf3-cec-fw-signed mlxbf-bootimages-signed
```

1. Remove the deprecated `hcoll` package:

```
<bf> $ rpm -e --nodeps hcoll
```

2. Manage kernel version locks: Ensure the correct kernel is applied.
   - For non-Anolis OS:
     i. Install the version lock plugin:

```
<bf> $ dnf install -y 'dnf-command(versionlock)'
```

     ii. Clear existing kernel locks:

```
<bf> $ dnf versionlock delete kernel*
```

   - For Anolis OS:
     i. Install the version lock plugin:

```
<bf> $ dnf install -y 'dnf-command(versionlock)'
```

ii. Add specific kernel locks:

```
<bf> $ dnf versionlock add \
> kernel-5.10.134-13.1.an8.aarch64 \
> kernel-core-5.10.134-13.1.an8.aarch64 \
> kernel-rpm-macros-129-1.0.2.an8.noarch \
> kernel-headers-5.10.134-13.1.an8.aarch64 \
> kernel-modules-5.10.134-13.1.an8.aarch64 \
> kernel-modules-extra-5.10.134-13.1.an8.aarch64 \
> kernel-devel-5.10.134-13.1.an8.aarch64 \
> anolis-release-8.6-4.an8.aarch64 \
> anolis-repos-8.6-4.an8.aarch64 \
> anolis-gpg-keys-8.6-4.an8.noarch
```

3. Install firmware packages:

```
<bf> $ dnf install -y bf2-bmc-fw-signed bf3-bmc-fw-signed bf3-bmc-gi-signed bf3-bmc-nic-fw* bf2-cec-fw-signed bf3-cec-fw-signed
```

4. Ensure **sshpass** is installed (required for component activation):

```
<bf> $ command -v sshpass >/dev/null 2>&1 || { dnf install -y sshpass || yum install -y sshpass; }
```

5. Perform the OS upgrade:
   - For non-Anolis OS:
     i. Run the standard upgrade command:

```
<bf> $ dnf upgrade --nobest
```

     ii. Lock the kernel packages to the current version:

```
<bf> $ dnf versionlock kernel*
```

     iii. Update the GRUB configuration to ensure the correct default boot entry:

```
<bf> $ sed -i 's/^GRUB_DEFAULT=.*/GRUB_DEFAULT=0/' /etc/default/grub
```

     iv. Rebuild the GRUB configuration file (replace **<OS_NAME>** with your OS directory name, e.g., **rocky**, **openEuler**):

```
<bf> $ grub2-mkconfig -o /boot/efi/EFI/<OS_NAME>/grub.cfg
```

   - For Anolis OS:
     i. Remove the deprecated gRPC package:

```
<bf> $ dnf -y remove doca-grpc
```

     ii. Remove OpenMPI and OpenSM dependencies:

```
<bf> $ rpm -e --nodeps openmpi opensm opensm-devel opensm-static opensm-libs
```

     iii. Upgrade while excluding specific trace and daemon packages:

```
<bf> $ dnf --exclude='libtraceevent*,rasdaemon*' upgrade
```

     iv. Verify if an **.rpmsave** GRUB backup exists:

```
<bf> $ ls /boot/efi/EFI/anolis/grub.cfg.rpmsave
```

v. If the file exists, rename it to restore the correct configuration:

```
<bf> $ mv /boot/efi/EFI/anolis/grub.cfg.rpmsave /boot/efi/EFI/anolis/grub.cfg
```

4.3.9.3.2.3 Firmware Configuration and Activation

⚠ Skip this phase if you do not wish to update firmware.

1. Configure BMC parameters by adding the BMC user and password to **/etc/bf-upgrade.conf** (format is identical to **bf.cfg**):

```
<bf> $ cat > /etc/bf-upgrade.conf << EOF
BMC_USER="<username>"
BMC_PASSWORD="<password>"
WITH_NIC_FW_UPDATE="yes"
BMC_REBOOT="yes"
EOF
```

⚠ **BMC_REBOOT="yes"** causes a warm boot to the BMC, temporarily losing connectivity.

2. Activate the components:

```
<bf> $ dpu-bmc-upgrade
```

⚠ The upgrade process takes up to 20 minutes. The script performs an **fwreset** if supported.

4.3.9.3.2.4 Apply Changes

For the upgrade to take effect, perform a BlueField system reboot to apply the new changes, NIC firmware, and UEFI/ATF.

⚠ This step triggers an immediate reboot of the BlueField Arm cores.

4.3.9.4 Installing Multiple BlueField Devices with DOCA Installer Tool

The DOCA Installer is a utility for installing or upgrading one or more BlueField DPUs in parallel. It supports both **bf-bundle** and **bf-fw-bundle** image types and is compatible with multiple BlueField operation modes, including NIC mode, DPU mode, or both on the same host. Customization is supported through INI or YAML configuration files.

4.3.9.4.1 Prerequisites

Make sure to update DOCA on the host side before installing **doca-installer**.

DEB-based	1. Download and install the DOCA host repo from the NVIDIA DOCA Downloads page. 2. Install doca-installer using the following command: `host# apt-get install doca-installer`
RPM-based	1. Download the DOCA host repo from the NVIDIA DOCA Downloads page. 2. Install doca-installer using the following command: `host# sudo yum install doca-installer`

4.3.9.4.2 DOCA Installer Operation Modes

The installer supports the following execution modes:

- Global run (applies to all devices):

```
host# doca-installer -b <image_path.bfb> -c <path-to-config>
```

- Per PSID group (targets devices sharing the same PSID):

```
host# doca-installer -b <image_path.bfb> --psid <PSID> -c <path-to-config>
```

> To exclude devices when using global or per-PSID runs, use **--exclude rshim<X>**.

- Per RShim device (targets a specific RShim interface):

```
host# doca-installer -b <image_path.bfb> --rshim rshim<N> -c <path-to-config>
```

Where:

- **-c** or **--config-file** – defines a configuration via INI or YAML files

> To upgrade BMC, CEC, or Golden Image firmware, include **BMC_USER** and **BMC_PASSWORD** in the **bf.cfg** or **bf.yaml** file.

- **<path-to-config>** – Path to the INI or YAML configuration file
- **<image_path.bfb>** – Path to the BFB image file
- **<PSID>** – Shared PSID value for the target group

> Devices that do not match the provided PSID will not be upgraded.

- **<N>** – RShim device index (e.g., **0** for a single-device setup)

⚠ The **doca-installer** script attempts to load the new NIC firmware automatically, but if that fails, it will print out the following message:

```
INFO: At least one device requires a host power cycle for new NIC Firmware to be loaded.
```

⚠ If the script discovers that one or more of the devices did not have **BMC_REBOOT** set to **'yes'** as part of the configuration file, it will print out the following message:

```
WARNING: BMC_REBOOT is not configured for all devices or is not set to 'yes'.
BMC firmware updates will only take effect after a BMC reboot.
```

4.3.9.4.3 Configuration File Options

4.3.9.4.3.1 YAML Configuration File

The YAML format supports global and per-device customization using structured keys:

- **global_configs** – Shared parameters for all devices (e.g., credentials)
- **rshim** – Device-specific configurations
  - Each key under **rshim** (e.g., **rshim1**, **rshim2**) defines variables and methods for that device
- **variables** – Parameters that control the installation process (e.g., **WITH_NIC_FW_UPDATE**, **BMC_REBOOT**)
- **methods** – Custom shell commands run during installation. You can use YAML anchors (**&**, ***) to reuse methods across devices.

⚠ When a value is defined both globally and per-device, the per-device value takes precedence.

Per-Device Configuration Only Example

```
rshim:
  rshim1:
    variables:
      BMC_USER: 'root'
      BMC_PASSWORD: '<root password>'
    methods:
      bfb_modify_os:
        command: |
          sed -i -e "s@192.168.100.2@192.168.101.2@" /mnt/var/lib/cloud/seed/nocloud-net/network-config
          sed -i -e "s@192.168.100.1@192.168.101.1@" /mnt/var/lib/cloud/seed/nocloud-net/network-config
```

Global Configuration Only Example

```
global_configs:
  variables:
    BMC_USER: 'root'
    BMC_PASSWORD: '<root password>'
```

Global and Per-Device Configuration Example

```
global_configs:
  variables:
    BMC_USER: 'root'
    BMC_PASSWORD: '<root password>'

rshim:
  rshim1:
    variables:
      WITH_NIC_FW_UPDATE: 'no'
  rshim2:
    variables:
      BMC_PASSWORD: '<rshim2 password>'
```

Reusing Methods with Anchors Example

```
global_configs:
  variables:
    BMC_USER: 'root'
    BMC_PASSWORD: '<root password>'

rshim:
  rshim1:
    variables:
      BMC_REBOOT: 'yes'
    methods:
      bfb_post_install: &default_post_install
        command: |
          echo "Running post-install on rshim1"

  rshim2:
    methods:
      bfb_post_install: *default_post_install
```

4.3.9.4.3.2 INI Configuration File

Per-Device Configuration ini Copy Edit Example

```
RSHIM0_BMC_USER="root"
RSHIM0_BMC_PASSWORD="<password>"

RSHIM1_BMC_USER="root"
RSHIM1_BMC_PASSWORD="<password>"
```

Shared and Per-Device Parameters Example

```
# Shared across all devices
BMC_USER="root"
BMC_PASSWORD="<password>"
NEW_BMC_PASSWORD="<new password>"

# Device-specific (optional)
RSHIM0_NET_RSHIM_MAC=00:1a:ca:ff:ff:01
RSHIM1_NET_RSHIM_MAC=00:1a:ca:ff:ff:03
```

4.3.9.4.4 Logs and Debuggability

The DOCA Installer tool provides extended logging and improved debuggability compared to the legacy **bfb-install** tool.

For each execution of the **doca-installer**, a unique log directory is created under:

```
/var/log/doca_installer_logs/doca_installer_<timestamp>
```

Each device will have its own directory with logs related only to that device. These directories contain detailed diagnostic files, such as:

- **rshim** diagnostics
- **lspci**

- `bfb-install` output
- `rshim misc` installation record output
- `mlxconfig` output
- `mlxprivhost` output
- `mstdump` output (if applicable)
- `flint`
- Register reads

The main directory itself holds key summary logs, including:

- Full execution log of the `doca-installer` tool
- System services
- `bfb-install` output
- Run result file saved in YAML

DOCA Installer Standalone Flags4.3.9.4.5

The `doca-installer` tool introduces enhanced capabilities for comparing target image firmware versions with those currently running on devices. This allows for safer, more controlled upgrade decisions.

4.3.9.4.5.1 Fetch Firmware Versions from Image

Use the `--show-target-fw` flag to display firmware versions embedded in the provided BFB image:

```
host# doca-installer -b <image_path.bfb> --show-target-fw
```

Example output:

```
|======================|
| Tool version | 1.3.3 |
|======================|

Extracting firmware versions from /root/bf-bundle-3.2.0-87_25.10_ubuntu-24.04_prod.bfb

Target Firmware Versions of the provided image:
============================================================
| Component    | BlueField-2         | BlueField-3          |
|==============|=====================|======================|
| DOCA         | 3.2.0104            | 3.2.0104             |
|--------------|---------------------|----------------------|
| BSP          | 4.13.0.13792        | 4.13.0.13792         |
|--------------|---------------------|----------------------|
| ATF          | 4.13.0-19-g5fcb...  | 4.13.0-19-g5fcb...   |
|--------------|---------------------|----------------------|
| UEFI         | 4.13.0-25-g6cc8...  | 4.13.0-25-g6cc8...   |
|--------------|---------------------|----------------------|
| NIC_FW       | 24.47.0412          | 32.47.0412           |
|--------------|---------------------|----------------------|
| BMC_FW       | BF-25.10-7          | BF-25.10-7           |
|--------------|---------------------|----------------------|
| CEC_FW       | 4.15                | 00.02.0195.0000_n02  |
|--------------|---------------------|----------------------|
| DPU_GI       | Not Supported       | 4.13.0.13792         |
|--------------|---------------------|----------------------|
| NIC_FW_GI    | Not Supported       | 32_45_1020           |
============================================================
```

4.3.9.4.5.2 Fetch Firmware Versions from Running Devices

Use the `--show-running-fw` flag to display firmware versions currently installed on each detected BlueField device:

```
host# doca-installer --show-running-fw
```

Example output:

```
Current firmware versions:

4 BF DPUs:
=============================================================================================================================
=============================================================================================
| PSID          | Part Num            | Serial Num    | PCI Addr      | RSHIM Dev | Mode | Privileged | Pre Upgrade
ATF       | Pre Upgrade UEFI       | Pre Upgrade NIC-FW | Pre Upgrade BMC | Pre Upgrade CEC         |
|---------------|---------------------|---------------|---------------|-----------|------|------------|--------------
---------|------------------------|--------------------|-----------------|-------------------------|
| MT_0000000884 | 900-9D3B6-00CV-AA0  | MT2421601NTJ | 0000:17:00.0 | rshim0    | NIC  | ENABLED    |
4.13.0-18-g210482eb2  | 4.13.0-24-g31a2ae28c5  | 32.47.0412         | BF-25.10-6      | 00.02.0195.0000_n02     |
| MT_0000000884 | 900-9D3B6-00CV-AA0  | MT2421601NJM | 0000:65:00.0 | rshim1    | DPU  | ENABLED    |
4.13.0-18-g210482eb2  | 4.13.0-24-g31a2ae28c5  | 32.47.0412         | BF-25.10-6      | 00.02.0195.0000_n02     |
| MT_0000000884 | 900-9D3B6-00CV-AA0  | MT2421901NJM | 0000:72:00.0 | rshim2    | DPU  | ENABLED    |
4.13.0-18-g210482eb2  | 4.13.0-24-g31a2ae28c5  | 32.47.0412         | BF-25.10-6      | 00.02.0195.0000_n02     |
| MT_0000000884 | 900-9D3B6-00CV-AA0  | MT2423600ALQ | 0000:ca:00.0 | rshim3    | NIC  | ENABLED    |
4.13.0-18-g210482eb2  | 4.13.0-24-g31a2ae28c5  | 32.47.0412         | BF-25.10-6      | 00.02.0195.0000_n02     |
=============================================================================================================================
=============================================================================================
```

4.3.9.4.5.3 Compare Image vs. Running Firmware Versions

Use the `--compare` option to compare current firmware versions on each device with those contained in the image:

```
host# doca-installer -b <image_path.bfb> --compare
```

Example output:

```
Firmware Version Comparison:
=============================================================================================================
Device: 0000:17:00.0 (PSID: MT_0000000884, Type: BF3)
=============================================================================
| Component    | Current Version (Running) | Image Version (Offline)       |
|==============|===========================|===============================|
| ATF          | 4.13.0-18-g210482         | 4.13.0-18-g210482eb2          |
|--------------|---------------------------|-------------------------------|
| UEFI         | 4.13.0-24-g31a2ae         | 4.13.0-24-g31a2ae28c5         |
|--------------|---------------------------|-------------------------------|
| NIC_FW       | 32.47.0412                | 32.47.0412                    |
|--------------|---------------------------|-------------------------------|
| BMC_FW       | BF-25.10-6                | BF-25.10-6                    |
|--------------|---------------------------|-------------------------------|
| CEC_FW       | 00.02.0195.0000_n02       | 00.02.0195.0000_n02           |
=============================================================================

...

Device: 0000:ca:00.0 (PSID: MT_0000000884, Type: BF3)
=============================================================================
| Component    | Current Version (Running) | Image Version (Offline)       |
|==============|===========================|===============================|
| ATF          | 4.13.0-18-g210482         | 4.13.0-18-g210482eb2          |
|--------------|---------------------------|-------------------------------|
| UEFI         | 4.13.0-24-g31a2ae         | 4.13.0-24-g31a2ae28c5         |
|--------------|---------------------------|-------------------------------|
| NIC_FW       | 32.47.0412                | 32.47.0412                    |
|--------------|---------------------------|-------------------------------|
| BMC_FW       | BF-25.10-6                | BF-25.10-6                    |
|--------------|---------------------------|-------------------------------|
| CEC_FW       | 00.02.0195.0000_n02       | 00.02.0195.0000_n02           |
=============================================================================
```

4.3.9.4.5.4 Fetch DPU Information

Use the `--show-doca-info` option to fetch `bf-info` content from the running DPUs. This feature is supported for devices running in DPU mode only.

> ⚠ This feature requires DOCA 3.2.0 or newer.

```
host# doca-installer --show-doca-info
```

Example output:

```
Device 1: 0000:ca:00.0
-----------------------------------------------
PSID: MT_0000000884
Device Type: BF3
MST Address: /dev/mst/mt41692_pciconf2


Firmware Components:
  ATF             : v2.2(release):4.13.0-19-g5fcb148df
  UEFI            : 4.13.0-25-g6cc8b2a0fa
  BSP             : 4.13.0.13792
  NIC_Firmware    : 32.47.0412
  BMC_Firmware    : 25.10-7
  CEC_Firmware    : 00.02.0195.0000

DOCA Components:
  doca-apsh-config 3.2.0104-1
  doca-bench 3.2.0104-1
  doca-caps 3.2.0104-1
  doca-comm-channel-admin 3.2.0104-1
  doca-dms 3.2.0104-1
  doca-eula 1.0.0-1
  doca-flow-tune 3.2.0104-1
  doca-openvswitch-common 3.2.0041-1
  doca-openvswitch-dev 3.2.0041-1
  doca-openvswitch-ipsec 3.2.0041-1
  doca-openvswitch-switch 3.2.0041-1
  doca-pcc-counters 3.2.0104-1
  .
  .
  .

FlexIO Components:
  flexio-samples 25.10.3034
  flexio-sdk 25.10.3034
  .
  .
  .

SoC Platform:
  mmc-utils 0+git20220624.d7b343fd-1ubuntu1

OFED Components:
  ibutils2 2.1.1-0.22300.MLNX20251015.g12cc06356.2510102
  ibverbs-providers:arm64 2510.0.9-1
  ibverbs-utils 2510.0.9-1
  infiniband-diags 2510.0.9-1
  libibmad-dev:arm64 2510.0.9-1
  libibmad5:arm64 2510.0.9-1
  libibnetdisc5:arm64 2510.0.9-1
  libibumad-dev:arm64 2510.0.9-1
  rdma-core 2510.0.9-1
  .
  .
  .

System Drivers:
  mlnx_dpdk       : 'MLNX_DPDK 22.11.2510.1.0'
  Kernel          : 6.8.0-1012-bluefield

System Tools:
  MFT             : 4.34.0-140
  mstflint        : 4.32.0-1

Storage Information:
  virtio-net-controller 25.10.4-1
```

4.3.9.4.6 BlueField-2 Limitation

BlueField-2 is supported with the following features only:

- Parallel installation (single, global, per-PSID, exclude)
- Show target firmware
- Debuggability (extensive logs collection)

Due to firmware limitation, **doca-installer** will not be able to present the versions of ATF, UEFI, BMC and CEC:

```
3 BF DPUs:
================================================================================================================================================
================================================================================
| PSID          | Part Num        | Serial Num   | PCI Addr     | RSHIM Dev | Mode | Privileged | Pre Upgrade ATF |
Pre Upgrade UEFI | Pre Upgrade NIC-FW | Pre Upgrade BMC | Pre Upgrade CEC |
|---------------|-----------------|--------------|--------------|-----------|------|------------|-----------------|
-----------------|--------------------|-----------------|-----------------|
| MT_0000000375 | MBF2M516A-CECOT | MT2128X10584 | 0000:17:00.0 | rshim1    | DPU  | ENABLED    | N/A             |
N/A              | 24.47.0412         | N/A             | N/A             |
| MT_0000000733 | MBF2M516C-CECOT | MT2231X19273 | 0000:65:00.0 | rshim2    | DPU  | ENABLED    | N/A             |
N/A              | 24.47.0412         | N/A             | N/A             |
| MT_0000000733 | MBF2M516C-CECOT | MT2231X19257 | 0000:83:00.0 | rshim3    | DPU  | ENABLED    | N/A             |
N/A              | 24.47.0412         | N/A             | N/A             |
================================================================================================================================================
================================================================================
```

4.3.9.5 Building Custom BFB Installation Image

To build a customized BlueField OS image, use the BFB build environment. For more details, refer to the bfb-build project in in GitHub.

To boot a customized BlueField OS image on a UEFI secure-boot-enabled BlueField (the default secure boot setting), the following conditions must be met:

- The OS image must be signed with a key already present in the UEFI DB (e.g., the Microsoft key)
- If the above is not possible, UEFI secure boot must be disabled to allow booting of an unsigned or differently signed image

For detailed instructions on managing UEFI secure boot, refer to the "Secure Boot" page in the BlueField Platform SW Documentation.

4.3.10 Additional SDKs for DOCA

4.3.10.1 Installing CUDA on NVIDIA Converged Accelerator

NVIDIA® CUDA® is a parallel computing platform and programming model designed by NVIDIA for general-purpose GPU computing. The following steps detail how to set up CUDA on your environment.

> ⚠ These instructions assume that a BFB image has already been installed on your system.

To install CUDA on your converged accelerator:

1. Download and install the latest NVIDIA Data Center GPU driver.
2. Download and install CUDA.

   > ℹ The CUDA version tested to work with DOCA SDK is 11.8.0.

Downloading CUDA includes the latest NVIDIA Data Center GPU driver and CUDA toolkit. For more information about CUDA and driver compatibility, refer to the NVIDIA CUDA Toolkit Release Notes.

4.3.10.1.1 Configuring Operation Mode

There are two modes that the NVIDIA Converged Accelerator may operate in:

- Standard mode (default) – the BlueField and the GPU operate separately
- BlueField-X mode – the GPU is exposed to BlueField and is no longer visible on the host

To verify which mode the system is operating in, run:

```
host# sudo mst start
host# sudo mlxconfig -d <device-id> q PCI_DOWNSTREAM_PORT_OWNER[4]
```

- Standard mode output:

```
Device #1:
[…]
Configurations:                                 Next Boot
         PCI_DOWNSTREAM_PORT_OWNER[4]           DEVICE_DEFAULT(0)
```

To configure standard mode, run:

```
host# mlxconfig -d <device-id> s PCI_DOWNSTREAM_PORT_OWNER[4]=0x0
```

- BlueField-X mode output:

```
Device #1:
[…]
Configurations:                                 Next Boot
         PCI_DOWNSTREAM_PORT_OWNER[4]           EMBEDDED_CPU(15)
```

To configure BlueField-X mode, run:

```
host# mlxconfig -d <device-id> s PCI_DOWNSTREAM_PORT_OWNER[4]=0xF
```

Power cycle is required for configuration to take effect. For power cycle the host run:

```
host# ipmitool power cycle
```

4.3.10.1.2 Downloading and Installing CUDA Toolkit and Driver

This section details the necessary steps to set up CUDA on your environment. It assumes that a BFB image has already been installed on your environment.

1. Install CUDA by visiting the CUDA Toolkit Downloads webpage.

Select the Linux distribution and version relevant for your environment.

This section shows the native compilation option either on x86 or aarch64 hosts.

2. Test that the driver installation completed successfully. Run:

```
dpu# nvidia-smi

Tue Apr  5 13:37:59 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 510.47.03    Driver Version: 510.47.03    CUDA Version: 11.8     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA BF A10       Off  | 00000000:06:00.0 Off |                    0 |
|  0%   43C    P0    N/A / 225W |      0MiB / 23028MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

3. Verify that the installation completed successfully.
   a. Download CUDA samples repo. Run:

```
dpu# git clone https://github.com/NVIDIA/cuda-samples.git
```

   b. Build and run vectorAdd CUDA sample. Run:

```
dpu# cd cuda-samples/Samples/0_Introduction/vectorAdd
dpu# make
dpu# ./vectorAdd
```

If the **vectorAdd** sample works as expected, it should output "**Test Passed**".

If it seems that the GPU is slow or stuck, stop execution and run:

```
dpu# sudo setpci -v -d ::0302 800.L=201 # CPL_VC0 = 32
```

4.3.10.1.3 GPUDirect RDMA

For information on GPUDirect RMDA and more, refer to DOCA GPUNetIO documentation.

4.3.10.2 Installing Rivermax on BlueField

NVIDIA Rivermax offers a unique IP-based solution for any media and data streaming use case.

This section provides the steps to install Rivermax assuming that a BFB image has already been installed on your environment.

4.3.10.2.1 Downloading Rivermax Driver

1. Navigate to the NVIDIA Rivermax SDK product page.
2. Register to be able to download the driver package using the JOIN button at the top of the page.
3. Download the appropriate driver package according to your BFB under the "Linux" subsection. For example, for Ubuntu 22.04 BFB, download **rivermax_ubuntu2204_<version>.tar.gz**.

4.3.10.2.2 Installing Rivermax Driver

1. Copy the **.tgz** file to BlueField:

```
host# sudo scp -r rivermax_ubuntu2204_<version>.tar.gz ubuntu@192.168.100.2:/tmp/
```

2. Extract the Rivermax file:

```
dpu# sudo tar xzf rivermax_ubuntu2204_<version>.tar.gz
```

3. Install the Rivermax driver package:

```
dpu# cd <rivermax-version>/Ubuntu.22.04/deb-dist/aarch64/
dpu# sudo dpkg -i rivermax_<version>.deb
```

4.3.10.2.3 Installing Rivermax Libraries from DOCA

Rivermax libraries are compatibles with DOCA components and can be found inside the **doca-dpu-repo**.

1. Unpack the doca-dpu-repo:

```
dpu# sudo dpkg -i doca-dpu-repo-ubuntu2204-local_<version>_arm64.deb
```

2. Run apt update:

```
dpu# sudo apt-get update
```

3. Install the Rivermax libraries:

```
dpu# sudo apt install doca-sdk-rmax
dpu# sudo apt install libdoca-sdk-rmax-dev
```

For additional details and guidelines, please visit the NVIDIA Rivermax SDK product page.

4.4 DOCA Developer Guide

This guide details the recommended steps to set up an NVIDIA DOCA development environment.

4.4.1 Introduction

This guide is intended for software developers aiming to modify existing NVIDIA DOCA applications or develop their own DOCA-based software.

Instructions for installing DOCA on the NVIDIA® BlueField® Networking Platform (i.e., DPU or SuperNIC) can be found in the DOCA Installation Guide for Linux.

This guide focuses on the recommended flow for developing DOCA-based software, and will address the following scenarios:

- BlueField is accessible and can be used during the development and testing process
- BlueField is inaccessible, and the development happens on the host or on a different server

It is recommended to follow the instructions for the first scenario, leveraging BlueField during the development and testing process.

This guide recommends using DOCA's development container during the development process on BlueField Platforms or on the host. Deploying development containers allows multiple developers to work simultaneously on the same device (host or BlueField Platform) in an isolated manner and even across multiple different DOCA SDK versions. This can allow multiple developers to work on the BlueField Platform itself, for example, without needing to have a dedicated BlueField per developer.

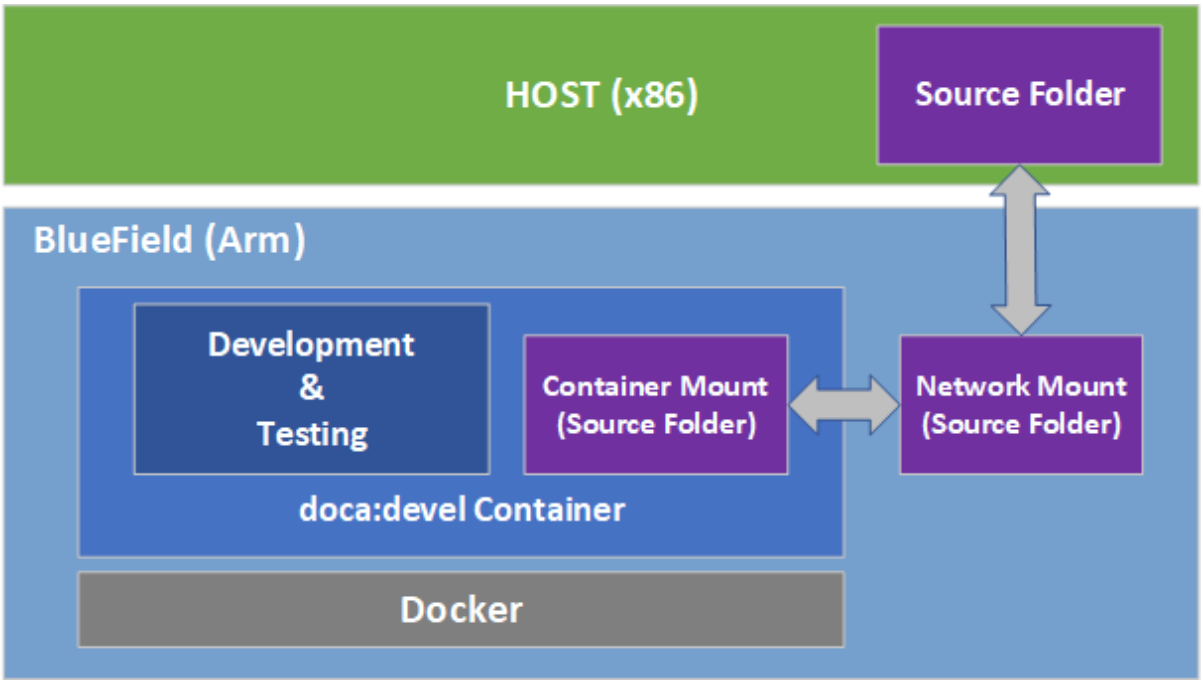
Another benefit of this container-based approach is that the development container allows developers to create and test their DOCA-based software in a user-friendly environment that comes pre-shipped with a set of handy development tools. The development container is focused on improving the development experience and is designed for that purpose, whereas the BlueField software is meant to be an efficient runtime environment for DOCA products.

For questions, comments, and feedback, please contact us at doca-feedback@nvidia.com.

4.4.2 Developing Using BlueField Networking Platform

4.4.2.1 Setup

DOCA's base image containers include a DOCA development container for the BlueField (`doca:devel`) which can be found on NGC. It is recommended to deploy this container on top of BlueField when preparing a development setup.



The recommended approach for working using DOCA's development container on top of the BlueField, is by using docker, which is already included in the supplied BFB image.

1. Make sure the docker service is started. Run:

```
sudo systemctl daemon-reload
sudo systemctl start docker
```

2. Pull the container image:
   a. Visit the NGC page of the DOCA base image.
   b. Under the "Tags" menu, select the desired development tag for BlueField.
   c. The container tag for the docker **pull** command is copied to your clipboard once selected. Example docker **pull** command using the selected tag:

```
sudo docker pull nvcr.io/nvidia/doca/doca:1.5.1-devel
```

3. Once loaded locally, you may find the image's ID using the following command:

```
sudo docker images
```

   Example output:

```
REPOSITORY                  TAG           IMAGE ID       CREATED         SIZE
nvcr.io/nvidia/doca/doca    1.5.1-devel   931bd576eb49   10 months ago   1.49GB
```

4. Run the docker image:

```
sudo docker run -v <source-code-folder>:/doca_devel -v /dev/hugepages:/dev/hugepages --privileged --
net=host -it <image-name/ID>
```

For example, to map a source folder named **my_sources** into the same container tag from the example above, the command should look like this:

```
sudo docker run -v my_sources:/doca_devel -v /dev/hugepages:/dev/hugepages --privileged --net=host -it
nvcr.io/nvidia/doca/doca:1.5.1-devel
```

After running the command, you get a shell inside the container where you can build your project using the regular build commands:

- From the container's perspective, the mounted folder will be named **/doca_devel**

  > ⚠ Make sure to map a folder with write privileges to **everyone**. Otherwise, the docker would not be able to write the output files to it.

- **--net=host** ensures the container has network access, including visibility to SFs and VFs as allocated on BlueField
- **-v /dev/hugepages:/dev/hugepages** ensures that allocated huge pages are accessible to the container

4.4.2.2 Development

It is recommended to do the development within the **doca:devel** container. That said, some developers prefer different integrated development environments (IDEs) or development tools, and sometimes prefer working using a graphical IDE until it is time to compile the code. As such, the recommendation is to mount a network share to BlueField (refer to DOCA DPU CLI for more information) and to the container.

> ⚠ Having the same code folder accessible from the IDE and the container helps prevent edge cases where the compilation fails due to a typo in the code, but the typo is only fixed locally within the container and not propagated to the main source folder.

4.4.2.3 Testing

The container is marked as "privileged", hence it can directly access the hardware capabilities of the BlueField Platform. This means that once the tested program compiles successfully, it can be directly tested from within the container without the need to copy it to BlueField and running it there.

4.4.2.4 Publishing

Once the program passes the testing phase, it should be prepared for deployment. While some proof-of-concept (POC) programs are just copied "as-is" in their binary form, most deployments will probably be in the form of a package (**.deb** / **.rpm**) or a container.

Construction of the binary package can be done as-is inside the current **doca:devel** container, or as part of a CI pipeline that will leverage the same development container as part of it.

For the construction of a container to ship the developed software, it is recommended to use a multi-staged build that ships the software on top of the runtime-oriented DOCA base images:

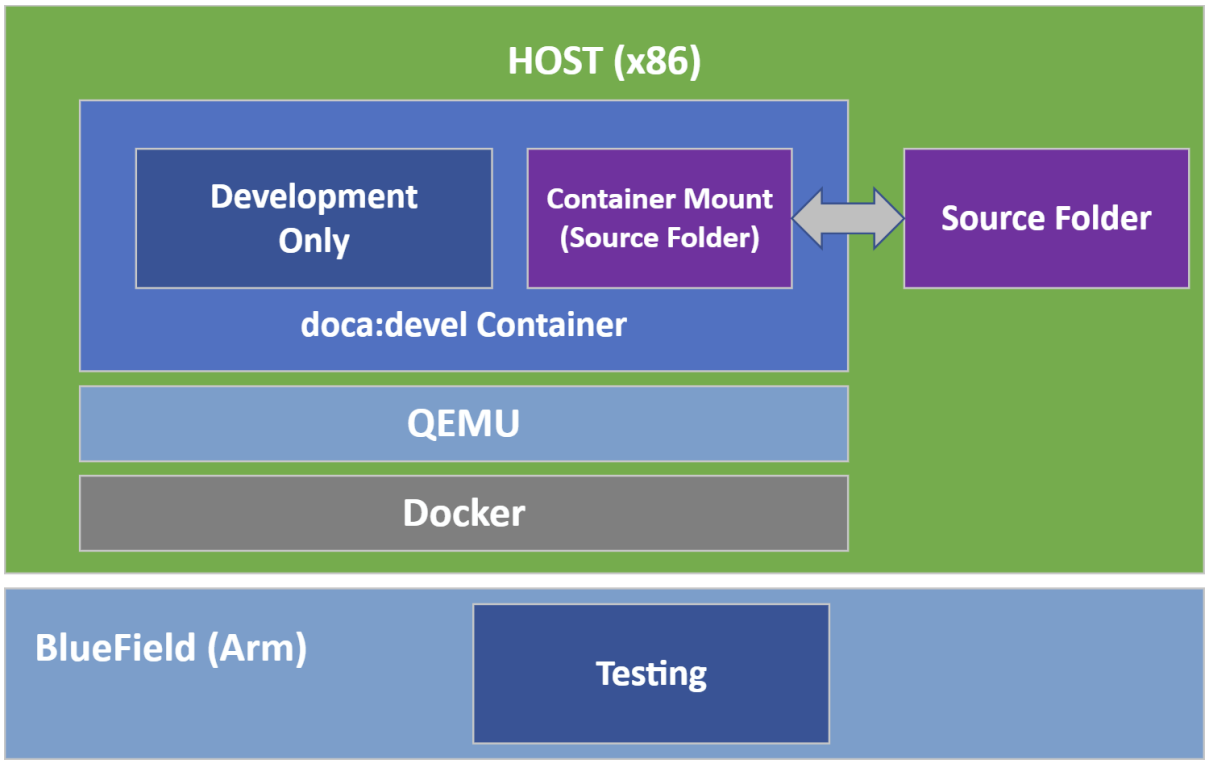
- `doca:base-rt` – slim DOCA runtime environment
- `doca:full-rt` – full DOCA runtime environment similar to the BlueField image

The runtime DOCA base images, alongside more details about their structure, can be found under the same NGC page that hosts the `doca:devel` image.

For a multi-staged build, it is recommended to compile the software inside the `doca:devel` container, and later copy it to one of the runtime container images. All relevant images must be pulled directly from NGC (using `docker pull`) to the container registry of BlueField.

4.4.3 Developing Without BlueField Networking Platform

If the development process needs to be done without access to a BlueField Platform, the recommendation is to use a QEMU-based deployment of a container on top of a regular x86 server. The development container for the host will be the same `doca:devel` image we mentioned previously.



4.4.3.1 Setup

1. Make sure Docker is installed on your host. Run:

   ```
   docker version
   ```

   If it is not installed, visit the official Install Docker Engine webpage for installation instructions.

2. Install QEMU on the host.

> ⚠ This step is for x86 hosts only. If you are working on an aarch64 host, move to the next step.

Host OS	Command
Ubuntu	`sudo apt-get install qemu binfmt-support qemu-user-static` `sudo docker run --rm --privileged multiarch/qemu-user-static --reset -p yes`
CentOS/RHEL 7.x	`sudo yum install epel-release` `sudo yum install qemu-system-arm`
CentOS 8.0/8.2	`sudo yum install epel-release` `sudo yum install qemu-kvm`
Fedora	`sudo yum install qemu-system-aarch64`

3. If you are using CentOS or Fedora on the host, verify if **qemu-aarch64.conf** Run:

```
cat /etc/binfmt.d/qemu-aarch64.conf
```

If it is missing, run:

```
echo ":qemu-aarch64:M::
\x7fELF\x02\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\xb7:\xff\xff\xff\xff\xff\xff\xff\xfc\xff\xf
f\xff\xff\xff\xff\xff\xfe\xff\xff:/usr/bin/qemu-aarch64-static:" > /etc/binfmt.d/qemu-aarch64.conf
```

4. If you are using CentOS or Fedora on the host, restart system binfmt. Run:

```
$ sudo systemctl restart systemd-binfmt
```

5. To load and execute the development container, refer to the "Setup" section discussing the same docker-based deployment on the BlueField side.

> ⚠ The **doca:devel** container supports multiple architectures. Therefore, Docker by default attempts to pull the one matching that of the current machine (i.e., **amd64** for the host and **arm64** for BlueField). Pulling the **arm64** container from the x86 host can be done by adding the flag **--platform=linux/arm64** :
>
> ```
> sudo docker pull --platform=linux/arm64 nvcr.io/nvidia/doca/doca:1.5.1-devel
> ```

4.4.3.2 Development

Much like the development phase using BlueField, it is recommended to develop within the container running on top of QEMU.

4.4.3.3 Testing

While the compilation can be performed on top of the container, testing the compiled software must be done on top of a BlueField Platform. This is because the QEMU environment emulates an aarch64 architecture, but it does not emulate the hardware devices present on the BlueField Platform. Therefore, the tested program will not be able to access the devices needed for its successful execution, thus mandating that the testing is done on top of a physical BlueField.

> ⚠ Make sure that the DOCA version used for compilation is the same as the version installed on BlueField used for testing.

4.4.3.4 Publishing

The publishing process is identical to the publishing process when using BlueField.

5 DOCA Programming Guide

The DOCA Programming Guide is intended for developers wishing to utilize DOCA SDK to develop application on top of the NVIDIA® BlueField® DPUs and SuperNICs.

- DOCA Programming Overview is important to read for new DOCA developers to understand the architecture and main building blocks most applications will rely on.
- DOCA Development Best Practices outlines common development pitfalls and capabilities to speed up application development, qualification, and productization.
- DOCA Libraries describes in details how to use each DOCA library, its APIs, and different aspects related to that library. Users may choose to only read the pages concerning DOCA libraries required for their application.
- DOCA Utils includes modules that may be used by application developers to speed up their development process (e.g., DOCA Arg Parser which simplifies the creation of a command-line interface for your application).
- DOCA Drivers describes additional frameworks used within DOCA.

For questions, comments, and feedback, please contact us at doca-feedback@nvidia.com.

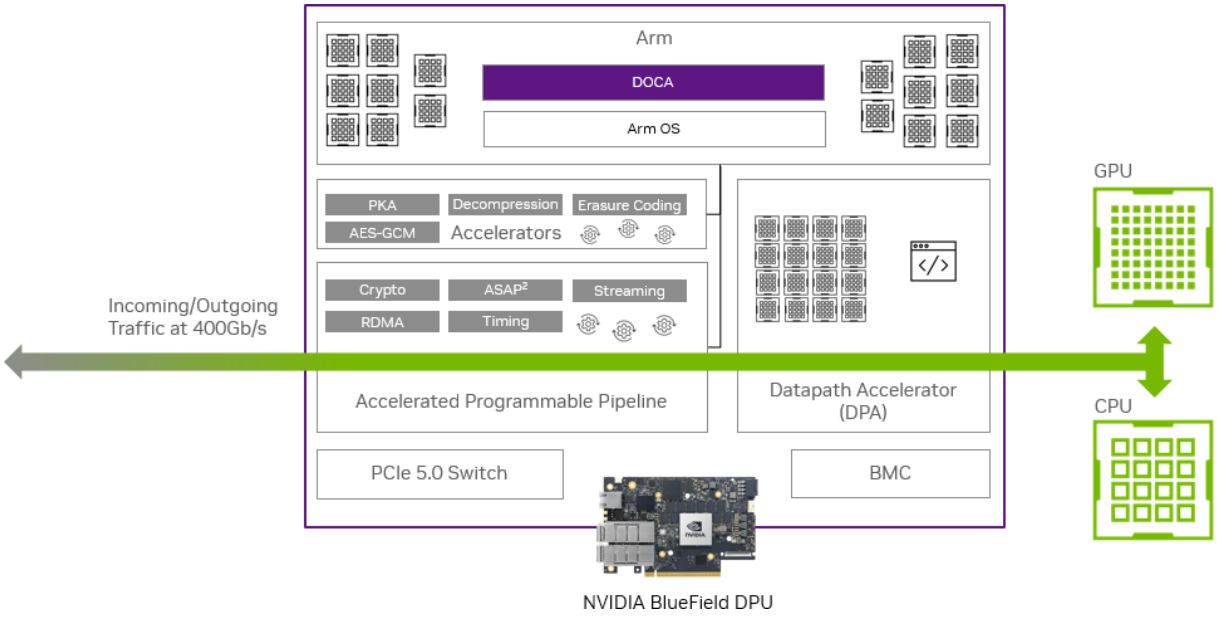
5.1 DOCA Programming Overview

This section contains the following pages:

- Hardware Architecture
- DOCA SDK Architecture

5.1.1 Hardware Architecture

BlueField's main hardware entities, which are optimized for different workloads.



5.1.1.1 Arm Cores

Optimized for control-path applications, general-purpose computing, and single-flow performance.

- 16 Arm Cortex-A78 cores for general-purpose processing
- Coherent Mesh architecture for efficient interconnectivity
- Last-Level Cache (LLC) for optimized memory performance
- DDR5 memory subsystem for high-speed data access
- Base OS and microservices for managing system resources

5.1.1.2 Accelerated Programmable Pipeline

Optimized for high-performance packet processing and advanced packet handling.

- Programmable 64-128 packet processor for flexible networking
- Multi-stage, highly parallelized architecture for throughput optimization
- Flow-based classification and action engine for efficient packet processing
- Support for RDMA, cryptographic acceleration, and time-based scheduling

5.1.1.3 Data-Path Accelerator

Designed for I/O-intensive applications, high insertion rate tasks, network flow processing, device emulation, and DMA operations.

- 16 hyper-threaded I/O and packet processing cores for handling intensive workloads
- Real-time OS for deterministic and low-latency operations

5.1.2 DOCA SDK Architecture

The DOCA SDK provides libraries for networking and data processing programmability, leveraging the NVIDIA® BlueField® networking platform (DPU or SuperNIC) and NVIDIA® ConnectX® NIC hardware accelerators.

The DOCA software framework is built on DOCA Core, which provides a unified software foundation for DOCA libraries. These libraries can be combined into processing pipelines or workflows that execute various offloaded tasks.

5.1.2.1 Device Subsystem

The DOCA SDK enables applications to offload resource-intensive tasks (e.g., encryption, compression) and network-related operations (e.g., packet acquisition, RDMA send) to specialized hardware.

5.1.2.1.1 Device Abstraction

The DOCA device subsystem provides an abstraction layer for the hardware processing units within BlueField and ConnectX devices. It allows applications to:

- Discover available hardware acceleration units provided by networking platforms
- Query capabilities and properties of these acceleration units
- Open and configure devices for libraries to allocate and share resources required for hardware acceleration

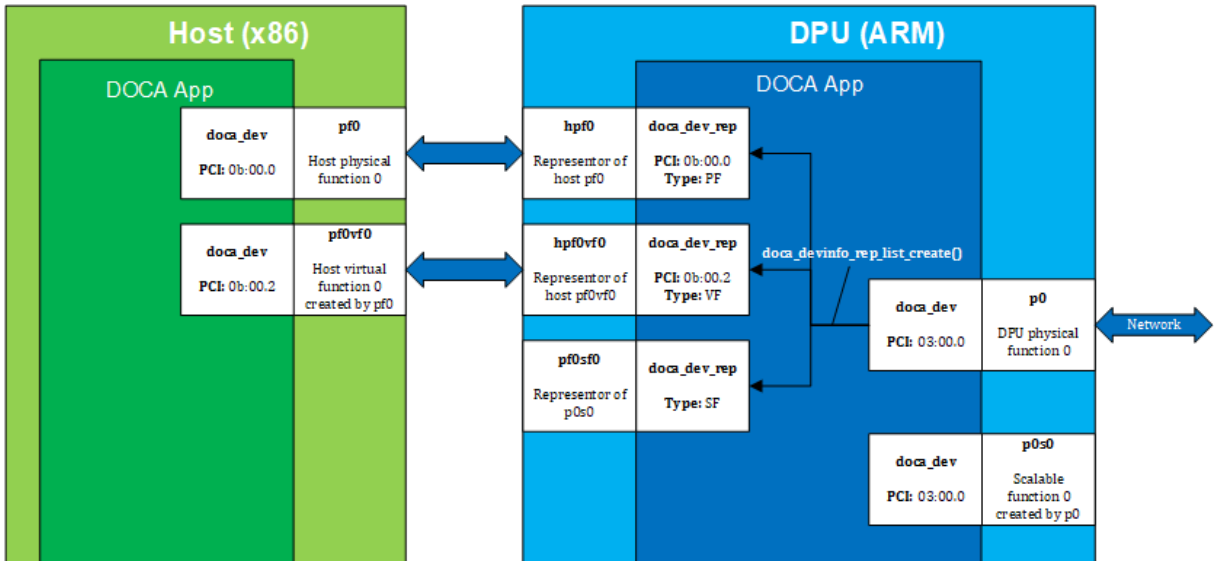
A system may have multiple available devices. Applications can select a device based on topology (e.g., PCIe address) or capabilities (e.g., encryption support).

5.1.2.1.2 DOCA Device Types

DOCA Core defines two types of DOCA Devices:

- Local device – A physical or virtual device exposed on the local system (BlueField or host). This includes:
  - Physical function (PF)
  - Virtual function (VF)
  - Scalable function (SF)
- Representor device – A proxy device on the BlueField side that represents a host-side device. The host-side function (e.g., PF, VF, or SF) corresponds to a 1:1 mapped representor on BlueField.

The following figure provides an example of host local devices with representors on BlueField:



DPU Mode

The diagram shows typical topology when using BlueField in DPU mode as described in BlueField Modes of Operation.

The diagram shows BlueField (on the right side of the figure) connected to a host (on the left). The host has physical function PF0 with a child virtual function VF0.

The BlueField side has a representor-device per host function in a 1-to-1 ratio (e.g., **hpf0** is the representor device for the host's PF0 device, etc.) as well as a representor for each SF function, such that both the SF and its representor reside in BlueField.

For more details on the DOCA Device subsystem, see section "DOCA Device".

5.1.2.2 Memory Management Subsystem

Hardware-accelerated processing tasks require data buffers as inputs and outputs. The DOCA SDK employs zero-copy technology to maximize performance by avoiding unnecessary data movement.

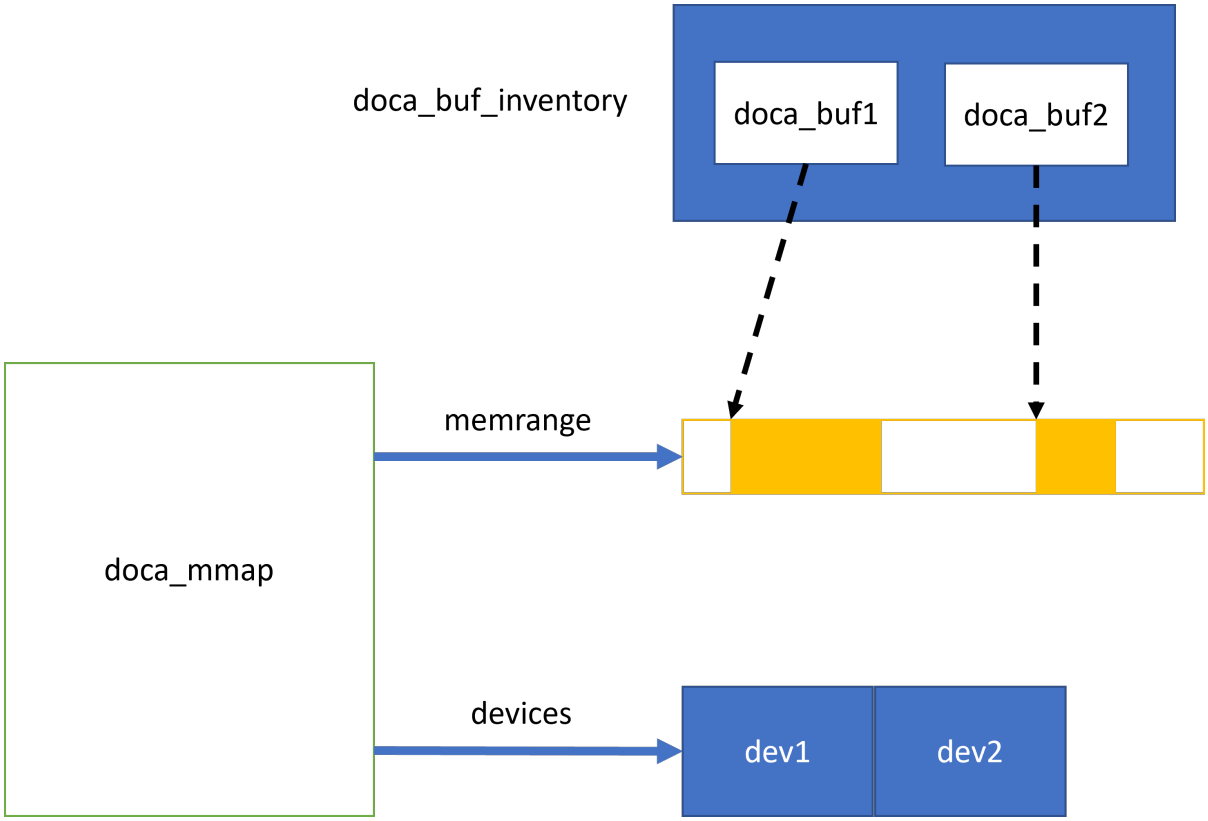
To enable zero-copy, applications must pre-register memory before using it for data buffers. The memory management subsystem facilitates:

- Memory registration:
  - Defines an application memory range for storing data buffers
  - Allows one or more devices to access the registered memory
  - Sets access permissions (e.g., read-only, read-write)
- Data buffer allocation management:
  - Allocates data buffers within the registered memory range
  - Supports memory pooling over registered memory

DOCA memory has the following main components:

- **doca_buf** – Represents a data buffer used as input/output for DOCA library operations
- **doca_mmap** – Describes registered memory accessible to devices with defined permissions. Each **doca_buf** resides within a **doca_mmap** memory range.
- **doca_buf_inventory** – A pool of **doca_buf** objects with shared characteristics (see more in sections "DOCA Core Buffers" and "DOCA Core Inventories")

The following diagram shows the various modules within the DOCA memory subsystem:



The diagram shows a **doca_buf_inventory** containing 2 **doca_buf**s. Each **doca_buf** points to a portion of the memory buffer which is part of a **doca_mmap**. The mmap is populated with one continuous memory range and is registered with 2 DOCA Devices, **dev1** and **dev2**.

> For more details about DOCA Memory management subsystem, see section "DOCA Memory Subsystem".

5.1.2.3 Execution Model

DOCA libraries abstract low-level hardware operations, allowing applications to focus on high-level processing tasks such as encryption, packet processing, and compression.

Each DOCA library defines dedicated APIs for performing these tasks. These libraries interact with hardware processing units via contexts.

5.1.2.3.1 Task and Event Model

DOCA APIs follow an asynchronous execution model, where applications interact with hardware through tasks and events:

- Task-based execution:
  a. The application prepares task arguments
  b. The application submits the task, requesting hardware processing
  c. The application receives a completion callback upon task completion
- Event-driven execution:
  a. The application registers for an event, instructing hardware to report when the event occurs
  b. The application receives a callback every time the event is triggered

5.1.2.3.2 Progress Engine – Managing Asynchronous Processing

Since hardware processing is asynchronous, DOCA provides a Progress Engine (PE) to manage task and event completion.

The PE supports:

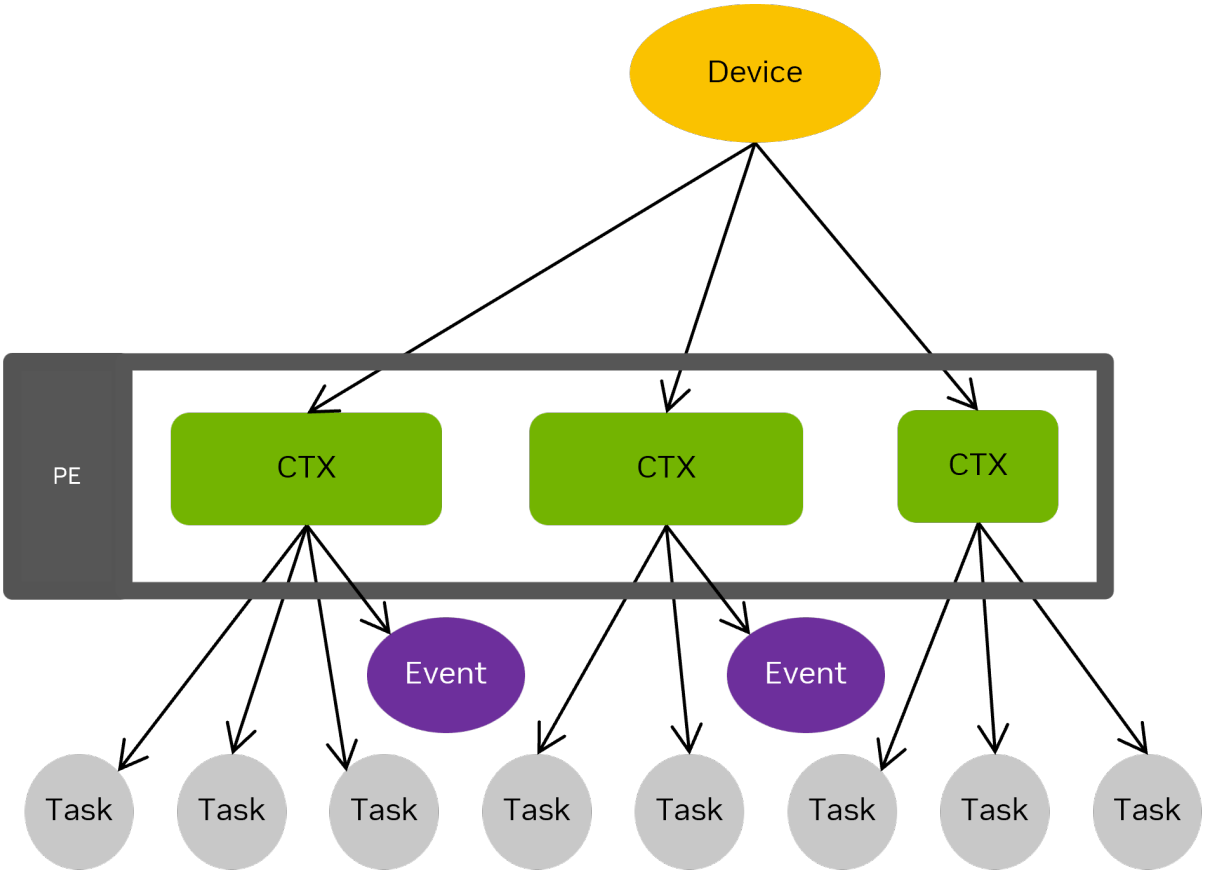
- Polling (or busy waiting) mode – The application repeatedly checks for completed tasks/events
- Notification mode – The application registers for OS-based notifications (e.g., Linux event FD) and is notified upon task/event completion

Once completion occurs, whether caused by a task or event, the relevant callback is invoked as part of the PE method.

A single PE instance allows waiting on multiple tasks/events from different contexts. As such, it is possible for an application to utilize a single PE per thread.

For more details about the DOCA Progress Engine, see section "DOCA Progress Engine".

The following diagram illustrates how a combination of various DOCA modules combine DOCA cross-library processing runtime.



The diagram shows 3 contexts utilizing the same device, each context has some tasks/events that have been submitted/registered by the application. All 3 contexts are connected to the same PE, where the application can use the same PE to wait on all completions at once.

For more details about DOCA Execution model see section "DOCA Execution Model".

5.2 DOCA Compatibility Policy

This page details the DOCA compatibility policy which governs how different components of the DOCA software stack—spanning the host, device, firmware, and hardware—and interact across different releases.

5.2.1 Introduction and Purpose

The DOCA compatibility policy provides customers with high velocity and flexibility during updates, enabling them to update specific layers of their system without requiring a full stack overhaul.

Key motivations:

- Decoupled updates – Supporting customers who have different teams managing separate parts of the software stack (e.g., firmware vs. user-space applications).
- Targeted upgrades – Allowing upgrades relevant to only a subset of devices on a server.
- Reduced maintenance – Minimizing downtime and complexity by only updating necessary components.
- Investment protection – Guaranteeing that customer applications will continue to operate successfully after updating to a newer DOCA software release.

This policy applies to all NVIDIA network device products and their associated software and firmware stacks.

5.2.2 Defining Compatibility

Compatibility is defined by the relationship between two layers when one is updated while the other remains unchanged. This relationship is defined using three key parameters: Layers, type, and direction.

5.2.2.1 Layers

Compatibility involves two interacting layers of the software/firmware stack running on either the host server or the network device.

5.2.2.2 Type of Compatibility

- Source compatibility (API persistence) – The ability to re-compile older source code without errors against newer platform libraries. This ensures that the published interfaces (APIs) remain consistent.
- Binary compatibility (symbols persistence) – The ability to run an older compiled program (binary) without errors against newer platform runtime libraries.
- Behavioral compatibility – Preserving the logical output and function of the code at runtime. Given the same input for the same interface, the same logical behavior is expected.

5.2.2.3 Direction of Compatibility

- Backward compatibility – The functionality continues to work when the lower layer (e.g., device firmware) is updated while the upper layer (e.g., host application) remains the same.
- Forward compatibility – The functionality continues to work when the upper layer is updated while the lower layer remains the same.

5.2.3 Software Release and Versioning Policy

DOCA product releases follow a structured cadence and versioning scheme to clearly communicate the scope of changes.

- Release cadence – The software platform package is released quarterly (e.g., Jan, Apr, July, Oct) as "General Availability (GA) releases," which include new features, performance enhancements, and bug fixes.

- Long-term support (LTS) – The October release of each year is designated as an LTS version.
- LTS versions are supported for three years.
- They receive updates that contain only bug fixes, no new features.
- An LTS version receives seven total updates ("LTS train") – three in the first year and two in each of the subsequent two years.
- Semantic versioning – NVIDIA follows the Semantic Versioning scheme (X.Y.Z), where:
- X – Major version – Incremented when incompatible API changes may be introduced.
- Y – Minor version – Incremented when new functionality is added in a backward compatible manner.
- Z – Patch version – Incremented for backward compatible bug fixes.

5.2.4 DOCA Compatibility Policy Types

DOCA defines specific compatibility policies based on the user role:

- DOCA Dependency Compatibility Policy – Designed for DOCA Administrators
- DOCA SDK Compatibility Policy – Designed for DOCA Developers

The following subpages describe these policies in more detail:

- DOCA Dependency Compatibility Policy
- DOCA SDK Compatibility Policy

5.2.5 DOCA Dependency Compatibility Policy

The DOCA Dependency Compatibility Policy ensures flexibility and efficiency during updates, enabling customers to upgrade specific layers of their system without requiring a full stack overhaul.

5.2.5.1 Key Objectives

- Decoupled updates: Supports environments where distinct teams manage separate parts of the software stack (e.g., firmware vs. user-space applications).
- Targeted upgrades: Enables upgrades targeted at specific subsets of devices within a server.
- Reduced maintenance overhead: Minimizes downtime and complexity by limiting updates to necessary components only.
- Operational continuity: Guarantees that customer applications continue to operate successfully after updating to a newer software platform release.

This policy applies to all NVIDIA network device products and their associated software and firmware stacks.

5.2.5.2 DOCA Dependency Policy Levels

DOCA defines two levels of dependency compatibility: Perpetual compatibility and Limited compatibility.

5.2.5.2.1 Level 1: Perpetual Dependency Compatibility

- Type: Perpetual forward and backward binary compatibility (e.g., a newer host component must work with an older firmware component, and vice versa).
- Condition: This is the highest level of support. Either the upper or lower layer may be upgraded independently, and all prior functionality is guaranteed to work seamlessly.
- Application: This policy applies to LTS (Long Term Support) update releases within a three-year support window.

5.2.5.2.2 Level 2: Limited Dependency Compatibility

- Type: Limited forward and backward binary compatibility.
- Condition: Breaking changes may be introduced annually, specifically with the October GA release.
- Limitation: Compatibility is guaranteed for all releases within a single annual cycle (October through July).

  > The July release is fully compatible with the preceding April, January, and October releases of the same cycle. However, it may not be compatible with the next October release.

- Application: This policy applies to new GA feature releases.

  > This does not apply to LTS update releases, which always follow Level 1 (Perpetual Dependency Compatibility).

5.2.5.3 Application of Policies

The following table summarizes how the compatibility policies apply to specific layers of the software stack:

#	Upper Layer	Lower Layer	Compatibility Level
1	DOCA-Host: DOCA-OFED profile	FW / BF-FW-Bundle	Level 1 – Perpetual
2	DOCA-Host: OTHER profiles (Networking, All)	FW / BF-FW-Bundle	Level 2 – Limited
3	DOCA-Host: DOCA-OFED profile	BF-Bundle	Level 1 – Perpetual
4	DOCA-Host: OTHER profiles (Networking, All)	BF-Bundle	Level 2 – Limited
5	DOCA-DPU	BF-FW-Bundle	Level 2 – Limited
6	DOCA Services (NGC and non-NGC)	BF-Bundle	Level 2 – Limited
7	DOCA Services (NGC and non-NGC)	FW / BF-FW-Bundle	Level 2 – Limited

5.2.5.4 Update Methodology

Compatibility support is a necessary precondition for supporting various customer update flows. While the exact flows are use-case dependent, this policy enables three primary update methodologies:

- Offline update: The system is taken offline during a maintenance window (usually involving a full system reboot) to apply changes. This is the simplest update method.
- Deferred update: New software is deployed without immediate service interruption. The system runs on the old version until a later system reset (determined by administrators) is performed for the changes to take effect.
- Hitless update: Updates are applied seamlessly without downtime or restart, maintaining continuous service availability. This typically applies to more granular components (e.g., a single service update).

> ⚠ Since the updated component must interact successfully with the rest of the system's un-updated components, it is advised to utilize containers to achieve this isolation.

5.2.6 DOCA SDK Compatibility Policy

The NVIDIA DOCA™ SDK enables developers to efficiently build applications and services on NVIDIA® BlueField® networking platforms.

DOCA follows a quarterly release cadence, delivering new features, performance enhancements, and critical bug fixes. The backward compatibility policy ensures that users can update to the latest DOCA software package—including libraries, drivers, and tools—without requiring modifications to their applications.

5.2.6.1 DOCA SDK Versioning

DOCA versions follow the Semantic Versioning scheme (i.e., X.Y.Z) where each component is updated based on the following criteria:

- Major version (X) – Introduces incompatible API changes
- Minor version (Y) – Adds functionality in a backward-compatible manner
- Patch version (Z) – Includes backward-compatible bug fixes

5.2.6.2 DOCA SDK API Backward Compatibility

A key characteristic of an enterprise-grade SDK is backward compatibility, which allows application developers to upgrade SDK versions without breaking existing functionality. This preserves their investment by ensuring applications remain operational across updates.

DOCA SDK APIs progress through the following lifecycle stages:

- Experimental – APIs marked as `DOCA_EXPERIMENTAL` are not guaranteed to be available in future releases.

- Stable – APIs marked as **DOCA_STABLE** are supported throughout the lifecycle of the current major version.
- Deprecated – APIs marked as **DOCA_DEPRECATED** will be removed in a future release. If previously marked as **DOCA_STABLE**, removal will occur in a major release.
- Removed – APIs that were available in an older major version but are no longer supported. If previously **DOCA_STABLE**, the binary representation remains intact to maintain binary backward compatibility.

The following sections describe the types of backward compatibility and how they align with semantic versioning.

5.2.6.2.1 Source Compatibility

Source compatibility ensures that a program written and compiled using libraries of a given DOCA SDK version continues to compile successfully with a newer DOCA SDK version.

As outlined in section "DOCA SDK Versioning", DOCA SDK maintains source compatibility across minor and patch versions. However, major version updates may introduce API changes, deprecations, or removals (refer to "DOCA SDK API Backward Compatibility" for details on API lifecycle stages). As a result, an application that compiles successfully with an older major version may require modifications to compile against a newer major version.

The above definition is aimed at DOCA SDK libraries & the symbols they expose. Compilation tools, on the other hand, may include better error or warning detection, resulting in failure to compile. This is a desired result, and is not considered "a source compatibility breakage".

5.2.6.2.2 Binary Compatibility

Binary compatibility ensures that a program dynamically linked to a given DOCA SDK library (***.so**) can successfully link against a newer version of the same library.

Binary compatibility is only applicable for programs dynamically linked with DOCA SDK libraries.

DOCA SDK provides a versioned C-style application binary interface (ABI) that maintains binary compatibility across both minor and major versions. This guarantees that upgrading the installed DOCA SDK version does not break existing applications.

5.2.6.2.3 Behavioral Compatibility

Behavioral (or semantic) compatibility ensures that a function or component produces the same output given the same input. Applications developed, compiled, linked, and tested with a given DOCA SDK version will function correctly with newer DOCA SDK versions, as long as they rely on defined behaviors. Bug fixes are the only exceptions.

5.2.6.3 Backward Compatibility for Non-Opaque Structs Using CTSD

The compile-time struct dispatching (CTSD) mechanism enables backward compatibility for non-opaque structs in DOCA libraries.

CTSD allows developers to write applications using generic API and struct definitions, which the compiler replaces with version-specific implementations at compile time. This approach maintains type safety and runtime compatibility.

The mechanism employs a macro-based scheme to map version-agnostic names to version-specific implementations, ensuring type safety and compile-time verification.

For example, when a user writes code using **doca_flow_func()**, the compiler resolves it to the appropriate version-specific implementation (e.g., **internal_doca_flow_func_v2_4()**) based on the selected library version.

5.2.6.3.1 Library Version Management

Each DOCA library exposes non-opaque structs maintains its own version history, independent of the main DOCA SDK version. These versions are declared in the library's header file (e.g., **doca_flow.h**).

For instance, a library may define version constants such as:

```
#define LIB_FLOW_VERSION_2_4
#define LIB_FLOW_VERSION_2_5
```

5.2.6.3.2 User Responsibility

To use CTSD, users must specify the desired library version by setting a preprocessor definition. For example:

```
#define LIB_FLOW_VERSION LIB_FLOW_VERSION_2_4
```

By default, libraries use the latest available version.

For specific version changelogs and documentation, refer to the corresponding library's documentation.

5.2.6.4 DPA SDK Compatibility Considerations

DPA SDK present additional compatibility considerations that should be taken into consideration:

1. DPA applications consuming DPA SDK libraries are composed of two parts: Host process and Device process. The host process is dynamically linked with host SDK libraries, and the device side is statically linked with the Device SDK libraries. As a result, if DOCA is upgraded without recompiling the application, then in effect, new library code is loaded on the host, but old device library code is loaded on DPA. This presents a new compatibility limitation between device library and host library. This specific compatibility will follow the "Limited dependency compatibility" policy defined in DOCA Dependency Compatibility Policy, where the host library is newer than the device library (i.e. "forward compatibility" only).
2. DPA device libraries are statically linked with the application, thus the binary compatibility between the DPA device library and the DPA application is not applicable. New DPA device code can only be deployed by recompiling the DPA application.
3. DPACC compiler toolchain produces binary files using LLVM bitcode format. The underlying version of LLVM framework is updated in every October/Rel2B release. As a result, the

compatibility between a binary DPA library generated by an older DPACC version and the DPACC used to build the application is also following the “Limited dependency compatibility” policy defined in DOCA Dependency Compatibility Policy.

4. DPACC (as any other compilation tool) may include better error or warning detection, resulting in failing to compile an application with a newer DPACC. This is a desired result and is not considered "a source compatibility breakage".

5.2.6.5 DOCA SDK Protocol Compatibility

Some DOCA SDK components facilitate communication between remote entities, such as:

- Host-to-BlueField
- BlueField-to-BlueField
- Host-to-host

Applications utilizing DOCA may be deployed in large clusters and upgraded at different times. To support seamless upgrades, DOCA SDK maintains protocol compatibility between different SDK versions, allowing for rolling upgrades without disrupting operations. Nodes running different DOCA SDK versions can continue communicating throughout the transition.

5.2.6.6 DOCA SDK Dependencies Compatibility

DOCA is distributed as a meta-package, available in the following formats:

- **`*.bfb`** – For installation on the BlueField networking platform’s Arm processor
- **`*.rpm`** / **`*.deb`** – For installation on the host server

Each package includes libraries, tools, executables, firmware, and sample applications.

DOCA SDK is tested as a complete package, ensuring that all components work together. There is no guarantee that upgrading individual components separately will maintain functionality. Therefore, updating to a newer DOCA version requires updating the entire meta-package.

5.3 DOCA Development Best Practices

The following sub-sections describe some best practices DOCA SDK users/developers should consider when using DOCA SDK.

- Capability Checking
- Debuggability

5.3.1 Capability Checking

Applications using a DOCA Device rely on a specific subset of features to function correctly. Therefore, it is recommended to verify whether the required features are available on the selected DOCA Device.

To facilitate this, the DOCA SDK provides capability APIs, a set of standardized APIs designed for querying feature support. These APIs should be used before initializing any resources to detect

unsupported features early, allowing applications to implement fallback mechanisms instead of encountering unexpected errors during execution.

5.3.1.1 Device Capability

Applications using DOCA Core APIs may need to determine whether a specific DOCA Device supports required capabilities.

To achieve this, the **doca_devinfo** and **doca_devinfo_rep** APIs expose capabilities using the following prefixes:

- **doca_devinfo_cap_***
- **doca_devinfo_rep_cap_***

Examples:

```
doca_error_t doca_devinfo_cap_is_hotplug_manager_supported(const struct doca_devinfo *devinfo, uint8_t
*is_hotplug_manager);
doca_error_t doca_devinfo_rep_cap_is_filter_emulated_supported(const struct doca_devinfo *devinfo, uint8_t
*filter_emulated_supported);
```

5.3.1.2 Library Capability

Each DOCA library provides capability APIs for:

- Querying valid configuration property limits (e.g., maximum/minimum values)
- Verifying whether a specific library task is supported on a given DOCA Device

All library capability APIs follow the naming convention `doca_<library_name>_cap_*`.

Additionally:

- Configuration limitation capability APIs use the prefix **doca_<library_name>_cap_[task_<task_type>]_get_min/max_***
- Task support capability APIs use the naming template **doca_<library_name>_cap_task_<task_type>_is_supported**

Examples from DOCA DMA:

```
doca_error_t doca_dma_cap_task_memcpy_is_supported(const struct doca_devinfo *devinfo);
doca_error_t doca_dma_cap_get_max_num_tasks(struct doca_dma *dma, uint32_t *max_num_tasks);
doca_error_t doca_dma_cap_task_memcpy_get_max_buf_size(const struct doca_devinfo *devinfo, uint64_t *buf_size);
```

5.3.1.3 Core Capability

Like other DOCA libraries, DOCA Core modules expose capability APIs to check feature availability.

Examples:

- A hotplugging application can verify if a DOCA Device supports hotplugging emulated PCIe functions using:

  ```
  doca_error_t doca_devinfo_cap_is_hotplug_manager_supported(const struct doca_devinfo *devinfo, uint8_t
  *is_hotplug_manager);
  ```

- An application using DOCA mmap for memory sharing between a host and BlueField must export the **doca_mmap** from the host and import it on BlueField. Before proceeding, the application can verify if these operations are supported for a given a **doca_devinfo** using:

```
doca_error_t doca_mmap_cap_is_export_pci_supported(const struct doca_devinfo *devinfo, uint8_t
*mmap_export);
doca_error_t doca_mmap_cap_is_create_from_export_pci_supported(const struct doca_devinfo *devinfo, uint8_t
*from_export);
```

5.3.2 Debuggability

DOCA SDK provides structured error handling and logging mechanisms to help developers diagnose and resolve issues efficiently.

5.3.2.1 Return Values

All DOCA APIs return a status code of type doca_error_t.

It is essential to check the return value of every DOCA API call to ensure successful execution. In the event of an error, refer to the function's documentation for the meaning of the returned status code and appropriate troubleshooting steps.

5.3.2.2 SDK Logging

The DOCA SDK provides error messages and debug prints to aid in debugging and issue resolution.

To enable DOCA SDK log messages, an application must:

1. Create a logging backend.
2. Set the verbosity level (if needed).

For more details on configuring and using DOCA logging, refer to "Enabling DOCA SDK Libraries Logging".

5.4 DOCA Libraries

Each DOCA library provides APIs, documentation, and examples designed to accelerate the development of data-center and networking applications.

This section describes the available libraries, their support levels, and key usage guidelines.

The quality level of each library reflects the maturity and support status of its major functionality. Individual APIs within a library may have a different quality level (e.g., alpha or beta). For detailed API-level quality information, refer to the respective library programming guide.

Library	Quality Level
DOCA Common	Beta
DOCA Flow	GA
Flex IO	GA
DOCA DPA	Beta

Library	Quality Level
DOCA PCC	Beta
DOCA DMA	GA
DOCA Comch	GA
DOCA UROM	Alpha
DOCA RDMA	GA
DOCA Ethernet	Beta
DOCA GPUNetIO	Beta
DOCA App Shield	GA
DOCA Compress	GA
DOCA SHA	GA
DOCA STA	Alpha
DOCA Erasure Coding	GA
DOCA AES-GCM	GA
DOCA Rivermax	Beta
DOCA Telemetry Exporter	GA
DOCA Telemetry	GA
DOCA Device Emulation	GA
DOCA DevEmu PCI Generic	GA
DOCA DevEmu Virtio-FS	GA
DOCA Management	Beta

5.4.1 DOCA Common

DOCA Common consists of the following libraries:

- DOCA Core
- DOCA Log

5.4.1.1 DOCA Core

This document provides guidelines on using DOCA Core objects as part of DOCA SDK programming.

5.4.1.1.1 Introduction

The quality status of DOCA libraries is listed here.

DOCA Core objects provide a unified and holistic interface for application developers to interact with various DOCA libraries. The DOCA Core API and objects bring a standardized flow and building blocks for applications to build upon while hiding the internal details of dealing with hardware and other software components. DOCA Core is designed to give the right level of abstraction while maintaining performance.

DOCA Core has the same API (header files) for both NVIDIA® BlueField® and CPU installations, but specific API calls may return **DOCA_ERROR_NOT_SUPPORTED** if the API is not implemented for that processor. However, this is not the case for Windows and Linux as DOCA Core does have API differences between Windows and Linux installations.

DOCA Core exposes C-language API to application writers and users must include the right header file to use according to the DOCA Core facilities needed for their application.

DOCA Core can be divided into the following software modules:

DOCA Core Module	Description
General	• DOCA Core enumerations and basic structures • Header files – **doca_error.h** , **doca_types.h**

Device handling	• Queries device information (host-side and BlueField) and device capabilities (e.g., device's PCIe BDF address)
	  • On BlueField
	  • Gets local BlueField devices
	  • Gets representors list (representing host local devices)
	  • On the host
	  • Gets local devices
	  • Gets representors list (representing local devices in the case of CX or DPU in NIC mode)
	  • Queries device capabilities and library capabilities
	• Opens and uses the selected device representor
	• Map device identifiers
	• Relevant entities – **doca_devinfo** , **doca_devinfo_rep** , **doca_dev** , **doca_dev_rep** , **doca_dpn_id**
	• Header files – **doca_dev.h** , **doca_dpn.h**
	There is a symmetry between representor entities and their corresponding "represented" devices. The use of the **rep** prefix in the API or object names indicates that the functionality or object is specific to a representor. This naming convention helps differentiate representor-specific components from others.

Memory management	• Handles optimized memory pools to be used by applications and enables sharing resources between DOCA libraries (while hiding hardware-related technicalities) • Data buffer services (e.g., linked list of buffers to support scatter-gather list) • Maps host memory to BlueField for direct access • Relevant entities – **doca_buf** , **doca_mmap** , **doca_buf_inventory** , **doca_buf_array** , **doca_bufpool** • Header files – **doca_buf.h** , **doca_buf_inventory.h** , **doca_mmap.h** , **doca_buf_array.h** , **doca_bufpool**

DOCA Core Module	Description
Progress engine and task execution	• Enables submitting tasks to DOCA libraries and track task progress (supports both polling mode and event-driven mode) • Relevant entities – `doca_ctx`, `doca_task`, `doca_event`, `doca_event_handle_t`, `doca_pe` • Header files – `doca_ctx.h`
Sync events	• Sync events are used to synchronize different processors (e.g., synchronize BlueField and host) • header files – `doca_dpa_sync_event.h`, `doca_sync_event.h`

The following sections describe DOCA Core's architecture and subsystems along with some basic flows that help users get started using DOCA Core.

5.4.1.1.2 Prerequisites

DOCA Core objects are supported on NVIDIA® BlueField® networking platforms (DPU or SuperNIC) and the host machine. Both must meet the following prerequisites:

- DOCA version 2.0.2 or greater
- NVIDIA® BlueField® software 4.0.2 or greater
- NVIDIA® BlueField®-3 firmware version 32.37.1000 and higher
- NVIDIA® BlueField®-2 firmware version 24.37.1000 and higher
- Please refer to the DOCA Compatibility Policy

5.4.1.1.3 Changes From Previous Releases

5.4.1.1.3.1 Changes in 3.4.0

N/A

5.4.1.1.3.2 Changes in 3.5.0

Added device event notification API on `doca_dev` for asynchronous link up/down events, integrated with the Progress Engine.

5.4.1.1.4 Architecture

he following sections describe the architecture for the various DOCA Core software modules. Please refer to the NVIDIA DOCA Library APIs for DOCA header documentation.

5.4.1.1.4.1 General

All core objects adhere to same flow that later helps in doing no allocations in the fast path.

The flow is as follows:

1. Create the object instance (e.g., `doca_mmap_create`).
2. Configure the instance (e.g., `doca_mmap_set_memory_range`).

3. Start the instance (e.g., **doca_mmap_start**).

After the instance is started, it adheres to zero allocations and can be used safely in the data path. After the instance is complete, it must be stopped and destroyed (**doca_mmap_stop**, **doca_mmap_destroy**).

There are core objects that can be reconfigured and restarted again (i.e., create → configure → start → stop → configure → start). Please read the header file to see if specific objects support this option.

doca_error_t

All DOCA APIs return the status in the form of **doca_error_t**.

```
typedef enum doca_error {
    DOCA_SUCCESS,
    DOCA_ERROR_UNKNOWN,
    DOCA_ERROR_NOT_PERMITTED,          /**< Operation not permitted */
    DOCA_ERROR_IN_USE,                 /**< Resource already in use */
    DOCA_ERROR_NOT_SUPPORTED,          /**< Operation not supported */
    DOCA_ERROR_AGAIN,                  /**< Resource temporarily unavailable, try again */
    DOCA_ERROR_INVALID_VALUE,          /**< Invalid input */
    DOCA_ERROR_NO_MEMORY,              /**< Memory allocation failure */
    DOCA_ERROR_INITIALIZATION,         /**< Resource initialization failure */
    DOCA_ERROR_TIME_OUT,               /**< Timer expired waiting for resource */
    DOCA_ERROR_SHUTDOWN,               /**< Shut down in process or completed */
    DOCA_ERROR_CONNECTION_RESET,       /**< Connection reset by peer */
    DOCA_ERROR_CONNECTION_ABORTED,     /**< Connection aborted */
    DOCA_ERROR_CONNECTION_INPROGRESS,  /**< Connection in progress */
    DOCA_ERROR_NOT_CONNECTED,          /**< Not Connected */
    DOCA_ERROR_NO_LOCK,                /**< Unable to acquire required lock */
    DOCA_ERROR_NOT_FOUND,              /**< Resource Not Found */
    DOCA_ERROR_IO_FAILED,              /**< Input/Output Operation Failed */
    DOCA_ERROR_BAD_STATE,              /**< Bad State */
    DOCA_ERROR_UNSUPPORTED_VERSION,    /**< Unsupported version */
    DOCA_ERROR_OPERATING_SYSTEM,       /**< Operating system call failure */
    DOCA_ERROR_DRIVER,                 /**< DOCA Driver call failure */
    DOCA_ERROR_UNEXPECTED,             /**< An unexpected scenario was detected */
    DOCA_ERROR_ALREADY_EXIST,          /**< Resource already exist */
    DOCA_ERROR_FULL,                   /**< No more space in resource */
    DOCA_ERROR_EMPTY,                  /**< No entry is available in resource */
    DOCA_ERROR_IN_PROGRESS,            /**< Operation is in progress */
    DOCA_ERROR_TOO_BIG,                /**< Requested operation too big to be contained */
 } doca_error_t;
```

See **doca_error.h** for more.

Generic Structures/Enum

The following types are common across all DOCA APIs.

```
union doca_data {
        void *ptr;
        uint64_t u64;
};

enum doca_access_flags {
    DOCA_ACCESS_LOCAL_READ_ONLY     = 0,
    DOCA_ACCESS_LOCAL_READ_WRITE    = (1 << 0),
    DOCA_ACCESS_RDMA_READ           = (1 << 1),
    DOCA_ACCESS_RDMA_WRITE          = (1 << 2),
    DOCA_ACCESS_RDMA_ATOMIC         = (1 << 3),
    DOCA_ACCESS_DPU_READ_ONLY       = (1 << 4),
    DOCA_ACCESS_DPU_READ_WRITE      = (1 << 5),
};

enum doca_pci_func_type {
        DOCA_PCI_FUNC_PF = 0, /* physical function */
        DOCA_PCI_FUNC_VF,     /* virtual function */
        DOCA_PCI_FUNC_SF,     /* sub function */
};
```

For more see **doca_types.h**.

5.4.1.1.4.2 DOCA Device

Local Device and Representor

There are 2 topologies for the representors model:

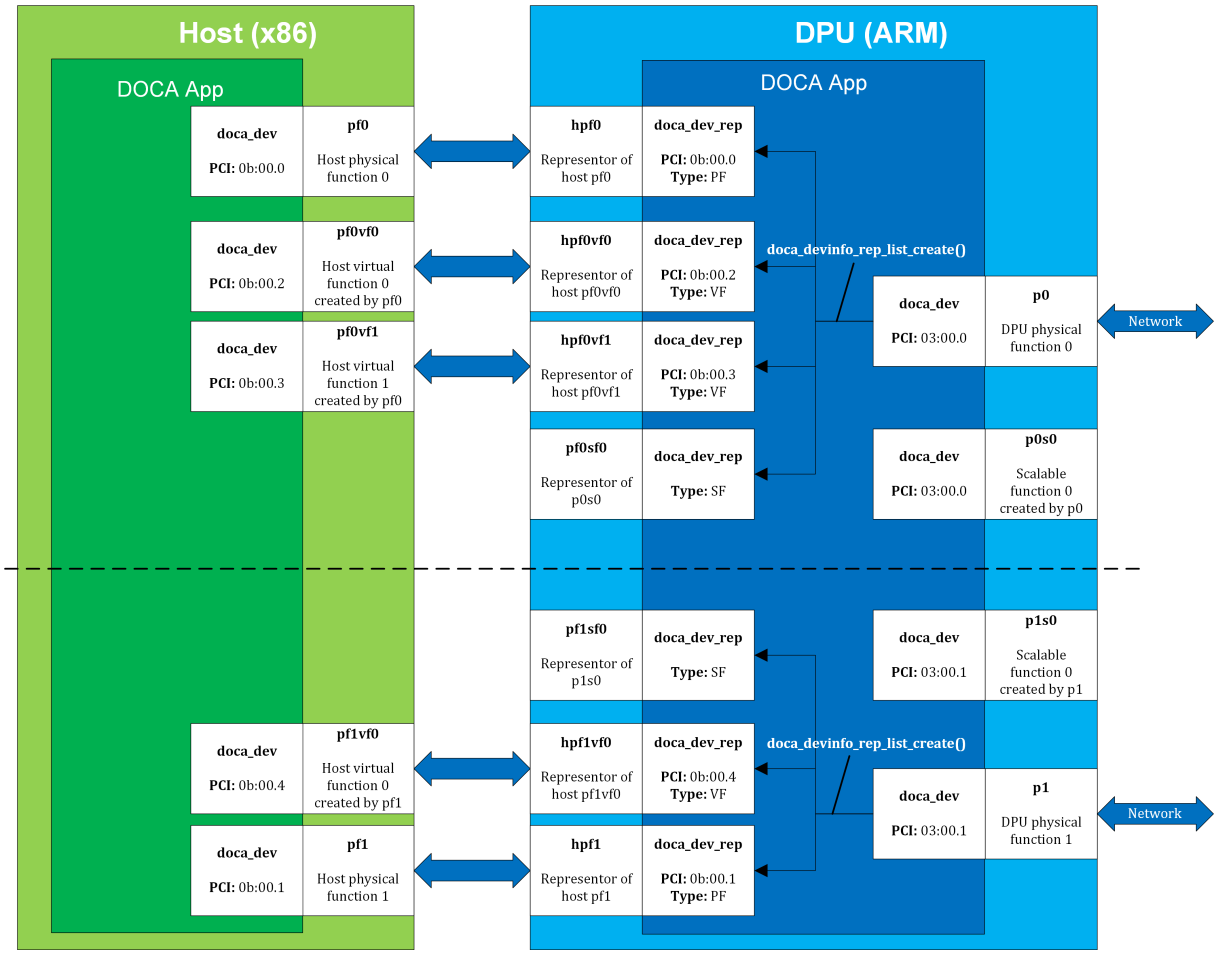
- DPU mode topology – BlueField must be operated in DPU mode (see explanation in section "DPU Topology").
- NIC mode topology – all devices and representors reside on the host. This is achieved using a BlueField operating in NIC mode or an NVIDIA® ConnectX® device. See explanation under DOCA Switching.

DPU Topology

A DOCA device represents an available processing unit, either hardware- or software-based. It exposes its properties to help applications select the most suitable device(s). DOCA Core supports two device types:

- Local device:
  - A physical device available on the local system (e.g., BlueField or host)
  - Capable of performing DOCA library processing tasks
- Representor device:
  - A proxy or representation of a local device
  - Typically used for host-side devices, with the representor located on the BlueField
  - Special Functions (SFs) reside entirely on the BlueField and also have their own representor devices

Topology Example



The following describes a typical topology, as shown in the diagram:

- Host topology:
  - The host system includes two physical functions (PFs): `PF0` and `PF1`.
  - `PF0` has two child virtual functions (VFs): `VF0` and `VF1`.
  - `PF1` has one associated VF: `VF0`.
  - Using the DOCA SDK API, users can query these five devices as local devices on the host.
- BlueField side:
  - BlueField maintains a 1-to-1 relation with host functions through representor devices. For example, `hpf0` is the representor for the host's `PF0` device.
  - BlueField also includes representors for SF devices. Both the SFs and their representors reside on the BlueField.

Querying Devices on BlueField

When querying local devices on BlueField (not representors), the result includes:

- Two BlueField DPU PFs (`p0` and `p1` in this example). These are the parent devices for all other devices.
- Associated devices:
  - 7 representor devices:

    - 5 representors for host functions (`hpf*` , shown as arrows connecting the host to BlueField in the diagram).
    - 2 representors for SF devices (`pf0sf0` and `pf1sf0`).
  - 2 local SF devices: These are not representors but physical devices on the BlueField (`p0s0` and `p1s0`).

The topology is divided into two parts (separated by a dotted line in the diagram), each represented by a BlueField physical device (`p0` and `p1`). Each BlueField device acts as the parent of:

- Local devices (e.g., PFs, VFs, and SFs).
- Representor devices (host PFs, host VFs, and SFs).

The parent device has access to the representor devices of all associated functions through the `doca_devinfo_rep_list_create` API.

Local Device and Representor Matching

Based on the topology diagram, the mmap export APIs can be used as follows:

Device to Select on Host When Using doca_mmap_export_dpu()	BlueField Matching Representor	Device to Select on BlueField When Using doca_mmap_create_from_export()
`pf0` – `0b:00.0`	`hpf0` – `0b:00.0`	`p0` – `03:00.0`
`pf0vf0` – `0b:00.2`	`hpf0vf0` – `0b:00.2`	
`pf0vf1` – `0b:00.3`	`hpf0vf1` – `0b:00.3`	
`pf1` – `0b:00.1`	`hpf1` – `0b:00.1`	`p1` – `03:00.1`
`pf1vf0` – `0b:00.4`	`hpf1vf0` – `0b:00.4`	

Expected Flow

Device Discovery

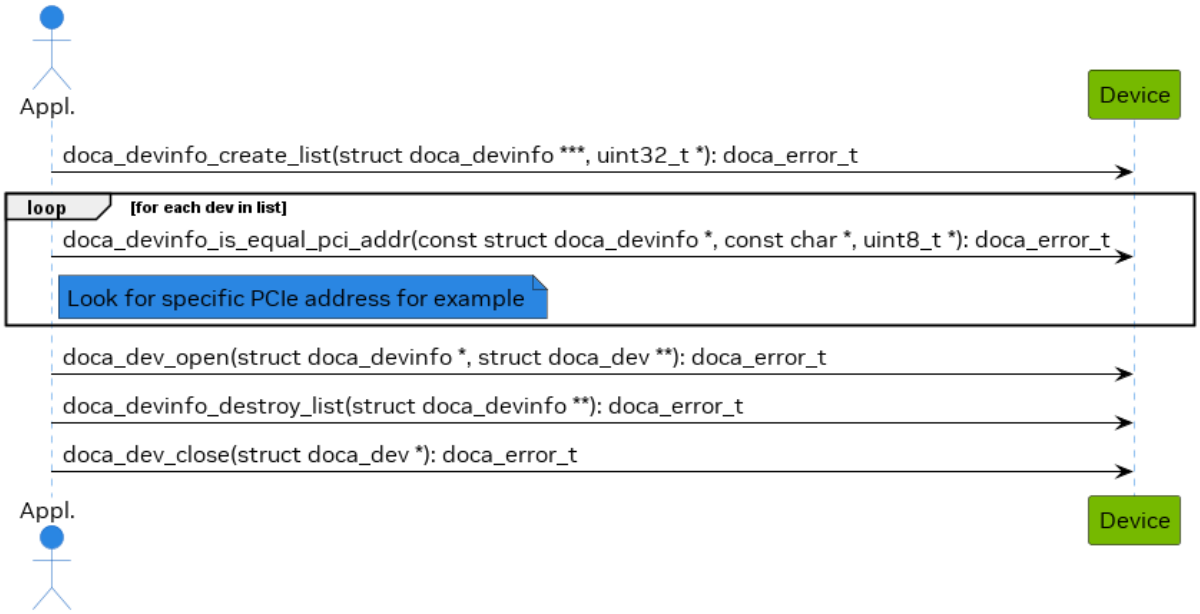
To work with DOCA libraries or DOCA Core objects, an application must open and use a device available on BlueField or the host.

Depending on the system setup, there are typically multiple devices available. For details on device topology and hierarchy, refer to the "DPU Topology" section.

An application can decide which device to select based on capabilities, the DOCA Core API, and every other library which provides a wide range of device capabilities. The flow is as follows:

1. The application gets a list of available devices.
2. Select a specific `doca_devinfo` to work with according to one of its properties and capabilities. This example looks for a specific PCIe address.
3. Once a suitable `doca_devinfo` is identified, open the device using the DOCA Core API to obtain a `doca_dev` handle.
4. After opening the desired device, release the `doca_devinfo` list to free up resources.
5. Use the `doca_dev` handle to interact with DOCA libraries or perform desired operations.

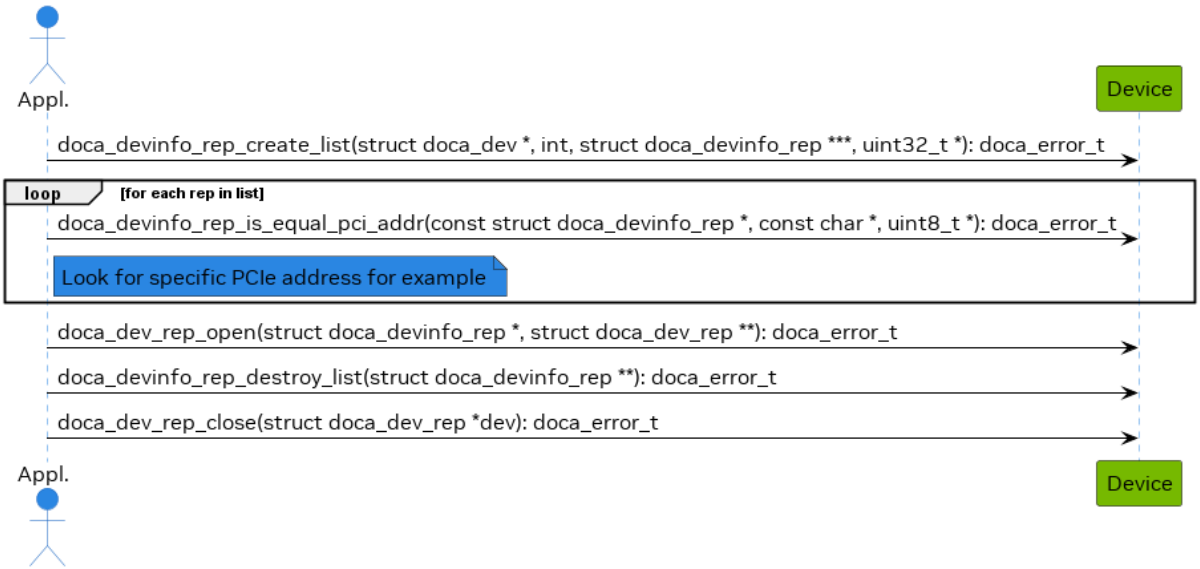
6. When the application finishes using the device, ensure that the **doca_dev** is properly closed to release resources.



Representor Device Discovery

To work with DOCA libraries or DOCA Core objects, some applications must open and use a representor device. Before they can open the representor device and use it, applications need tools to allow them to select the appropriate representor device with the necessary capabilities. The DOCA Core API provides a wide range of device capabilities to help the application select the right device pair (device and its representor). The flow is as follows:

1. The application "knows" which device it wants to use (e.g., by its PCIe BDF address). It can be done using DOCA Core API or OS services.
2. The application gets a list of device representors for a specific device.
3. Select a specific **doca_devinfo_rep** to work with according to one of its properties. This example looks for a specific PCIe address.
4. Once a suitable **doca_devinfo_rep** is identified, open **doca_dev_rep**.
5. After the user opens the right device representor, they can close the **doca_devinfo_rep** list and continue working with **doca_dev_rep**. The application eventually must close **doca_dev_rep** too.



As mentioned previously, the DOCA Core API can identify devices and their representors that have a unique property (e.g., the BDF address, the same BDF for the device, and its representor).

> ⚠ Regarding representor device property caching, the function **doca_devinfo_rep_create_list** provides a snapshot of the DOCA representor device properties when it is called. If any representor's properties are changed dynamically (e.g., BDF address changes after bus reset), the device properties that the function returns would not reflect this change. One should create the list again to get the updated properties of the representors.

Device PCIe-to-DPN Mapping

Certain hardware features utilize an identification scheme based on DPN (Depth, PCI Bus, Node). This addressing method is required when interacting with the PCIe bus in configurations where the BlueField or ConnectX device functions as a PCIe hub for nested or downstream PCIe devices.

The DOCA Core library provides a dedicated function to map a standard PCIe SBDF (Segment:Bus:Device.Function) address to its equivalent DPN identifier, representing either the device itself or one of its child components:

Function	**doca_dpn_map_pci_addr_to_dpn_id**
Behavior	Identifies and returns the appropriate DPN value for the specified address
Scope Constraint	DPN identifiers are unique strictly within the context of a specific parent device. Consequently, the parent device handle must be provided as an input parameter to the mapping function.

Verification via mlxlink

You can retrieve a comprehensive list of available PCIe addresses and their corresponding DPN values for a specific device using the **mlxlink** utility:

```
mlxlink -d <DEVICE_ID> --port_type PCIE --show_links
```

Make sure to replace **<DEVICE_ID>** with the target device's PCIe address, MST name, or MLX identifier.

DOCA Application Restart

Restarting a DOCA application can be done gracefully or non-gracefully. Each method has its own steps and considerations to ensure proper resource management and minimize downtime.

In a graceful restart, the application follows a structured process to ensure that all resources are properly managed and released before restarting. The application should first gracefully free all associated resources including those tied to the DOCA device (e.g., DOCA Flow switch ports and their components). After freeing the resources, the application calls **doca_dev_close** to close the DOCA device instance and then terminates. In a non-graceful restart, the application crashes unexpectedly without freeing the associated resources. The application is then started again.

The **doca_dev_accelerate_resource_reclaim** API can be useful in both scenarios to optimize the reclaim process for resources associated with the device. By retaining critical resources in the cache, this API ensures that they can be quickly reclaimed when needed, minimizing downtime and speeding up the recovery process, especially in non-graceful restart scenarios.

For a graceful restart, users should call the **doca_dev_accelerate_resource_reclaim** API before initiating the device resource cleanup. After calling the API, users can proceed with freeing all associated resources, closing the DOCA device instance with **doca_dev_close**, and terminating the application. This ensures that resources are properly managed and released before the application restarts.

For a non-graceful restart, where the application crashes unexpectedly, the **doca_dev_accelerate_resource_reclaim** API can be called periodically (e.g., every 5 seconds) to ensure readiness. This periodic invocation enables continuous retention of critical resources in the cache so that when the application restarts, these resources can be quickly reclaimed, minimizing downtime and speeding up the recovery process.

After restarting the application, whether gracefully or non-gracefully, the application should call **doca_dev_open** to create a new DOCA device instance.

> Immediately after the return of **doca_dev_open**, it is recommended to call the **doca_dev_accelerate_resource_reclaim** API to extend the retention period of the resources associated with the device in the cache.

Afterwards, the application should allocate the necessary resources associated with the DOCA device. This ensures that the application can resume its operations with the required resources in place, leveraging the retained resources in the cache for a faster and more efficient restart process.

> Using this API is not without cost, as it may lead to a lack or shortage of system resources, potentially causing overall system performance degradation. Only use the API for the specified use-cases above. If you are unsure about a specific application use-case, please contact NVIDIA Enterprise Support for questions.

Device Event Notification

Applications can receive asynchronous notifications when a device's link state changes. This allows reacting to link up/down events without polling.

Device event notification is delivered through the DOCA Progress Engine (PE). The registered callback is invoked from `doca_pe_progress()` when a subscribed event occurs. The callback must be non-blocking.

Supported event types:

Event	Description
`DOCA_DEV_EVENT_TYPE_LINK_DOWN`	Device link went down
`DOCA_DEV_EVENT_TYPE_LINK_UP`	Device link came up

To subscribe to all events supported by the device, use `DOCA_DEV_EVENT_MASK_ALL`. Alternatively, pass a mask returned by `doca_devinfo_cap_is_event_notification_supported()`.

To read the current link state synchronously, without waiting for an event, use `doca_dev_get_link_state()`.

Requirements and Considerations

- Supported on Linux only.
- The device must be open before registering for event notification.
- Registration alone does not deliver events. The internal notification context must be connected to a PE and started with `doca_ctx_start()`.
- Callback configuration, event mask, and user data can be updated only while the notification context is idle.
- The device cannot be closed while the notification context is running. Stop the context and unregister before calling `doca_dev_close()`.
- For the PE event loop pattern, refer to the Blocking Mode - Notification Driven section.

Key API Functions

Function	Description
`doca_devinfo_cap_is_event_notification_supported()`	Returns a bitmask of supported events on the device
`doca_dev_register_event_notification()`	Registers callback and event mask on an open `doca_dev`
`doca_dev_unregister_event_notification()`	Unregisters and destroys the internal notification context
`doca_dev_as_ctx()`	Returns the `doca_ctx` associated with device event notification
`doca_dev_get_link_state()`	Reads the current link state synchronously

5.4.1.1.4.3 DOCA Memory Subsystem

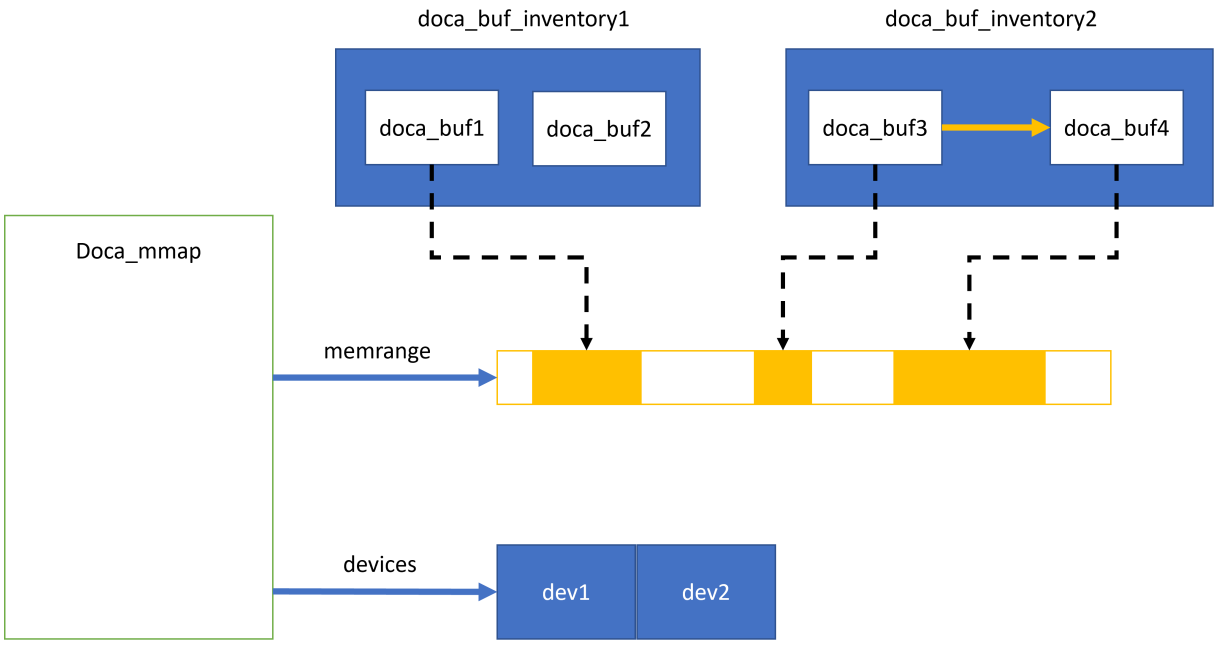
DOCA memory subsystem is designed to optimize performance while keeping a minimal memory footprint (to facilitate scalability) as main design goal.

DOCA memory has the following main components:

- **doca_buf** – this is the data buffer descriptor. This is not the actual data buffer, rather, it is a descriptor that holds metadata on the "pointed" data buffer.
- **doca_mmap** – this is the data buffers pool which **doca_buf** points at. The application provides the memory as a single memory region, as well as permissions for certain devices to access it.

As the **doca_mmap** serves as the memory pool for data buffers, there is also an entity called **doca_buf_inventory** which serves as a pool of **doca_buf** with same characteristics (see more in sections "DOCA Core Buffers" and "DOCA Core Inventories"). Like all DOCA entities, memory subsystem objects are opaque and can be instantiated by DOCA SDK only.

The following diagram shows the various modules within the DOCA memory subsystem.



In the diagram, you may see two **doca_buf_inventory** s. Each **doca_buf** points to a portion of the memory buffer which is part of a **doca_mmap** . The mmap is populated with one continuous memory buffer **memrange** and is mapped to two devices, **dev1** and **dev2** .

Requirements and Considerations

- The DOCA memory subsystem mandates the usage of pools as opposed to dynamic allocation
  - Pool for **doca_buf** → **doca_buf_inventory**
  - Pool for data memory → **doca_mmap**

- The memory buffer in the mmap can be mapped to one device or more
- Devices in the mmap are restricted by access permissions defining how they can access the memory buffer
- **doca_buf** points to a specific memory buffer (or part of it) and holds the metadata for that buffer
- The internals of mapping and working with the device (e.g., memory registrations) is hidden from the application
- As best practice, the application should start the **doca_mmap** in the initialization phase as the start operation is time consuming. **doca_mmap** should not be started as part of the data path unless necessary.
- The host-mapped memory buffer can be accessed by BlueField

doca_mmap

doca_mmap is more than just a data buffer as it hides a lot of details (e.g., RDMA technicalities, device handling, etc.) from the application developer while giving the right level of abstraction to the software using it. **doca_mmap** is the best way to share memory between the host and BlueField so BlueField can have direct access to the host-side memory or vice versa.

DOCA SDK supports several types of mmap that help with different use cases: local mmap and mmap from export.

Local mmap

This is the basic type of mmap which maps local buffers to the local device(s).

1. The application creates the **doca_mmap**.
2. The application sets the memory range of the mmap using **doca_mmap_set_memrange**. The memory range is memory that the application allocates and manages (usually holding the pool of data sent to the device's processing units).
3. The application adds devices, granting the devices access to the memory region.
4. The application can set a custom access token for a device in the mmap, using **doca_mmap_set_export_access_token** so that mmap export operations use that token instead of an automatically generated random one.
5. The application can specify the access permission for the devices to that memory range using **doca_mmap_set_permissions**.
    - If the mmap is used only locally, then **DOCA_ACCESS_LOCAL_*** must be specified
    - If the mmap is created on the host but shared with BlueField (see step 6), then **DOCA_ACCESS_PCI_*** must be specified
    - If the mmap is created on BlueField but shared with the host (see step 6), then **DOCA_ACCESS_PCI_*** must be specified
    - If the mmap is shared with a remote RDMA target, then **DOCA_ACCESS_RDMA_*** must be specified
6. The application starts the mmap.

    ⚠ From this point no more changes can be made to the mmap.

7. To share the mmap with BlueField/host or the RDMA remote target, call **doca_mmap_export_pci** or **doca_mmap_export_rdma** respectively. If appropriate access has not been provided, the export fails.

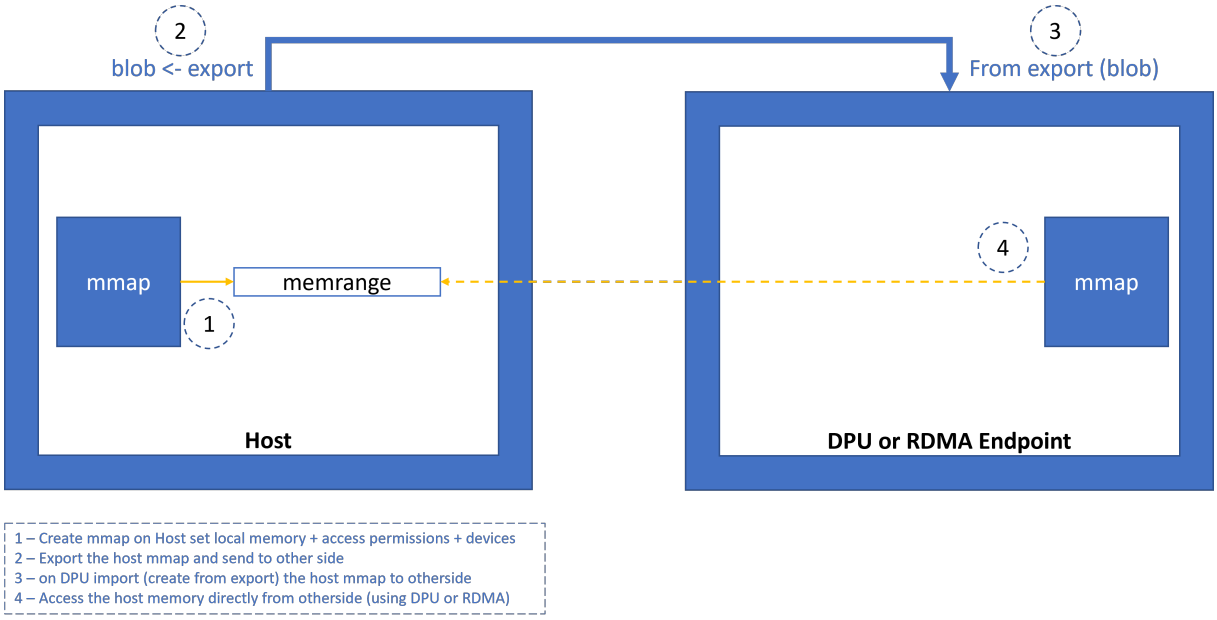
   > The exported data contains sensitive information. Make sure to pass this data through a secure channel!

8. The generated blob from the previous step can be shared out of band using a socket. If shared with a BlueField, it is recommended to use the DOCA Comm Channel instead. See the DMA Copy Application Guide for the exact flow.

mmap from Export

This mmap is used to access the host memory (from BlueField) or the remote RDMA target's memory.

1. The application receives a blob from the other side. The blob contains data returned from step 6 in the former bullet.
2. The application calls **doca_mmap_create_from_export** and receives a new mmap that represents memory defined by the other side.



Now the application can create **doca_buf** to point to this imported mmap and have direct access to the other machine's memory.

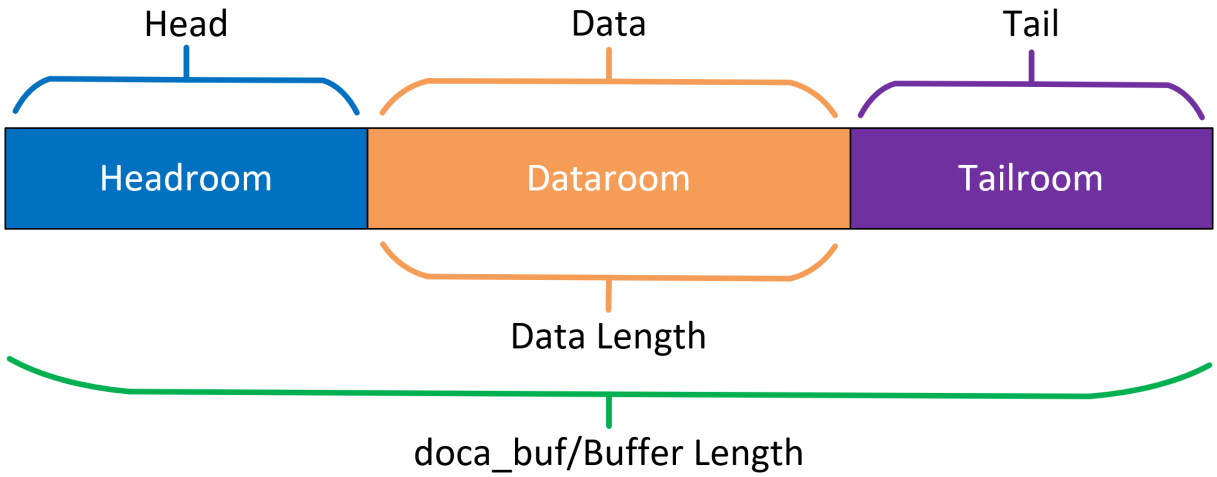
> BlueField can access memory exported to BlueField if the exporter is a host on the same machine. Or it can access memory exported through RDMA which can be on the same machine, a remote host, or on a remote BlueField.

The host can only access memory exported through RDMA. This can be memory on a remote host, remote BlueField, or BlueField on same machine.

Buffers

The DOCA buffer object is used to reference memory that is accessible by BlueField hardware. The buffer can be utilized across different BlueField accelerators. The buffer may reference CPU, GPU, host, or even RDMA memory. However, this is abstracted so once a buffer is created, it can be handled in a similar way regardless of how it got created. This section covers usage of the DOCA buffer after it is allocated.

The DOCA buffer has an address and length describing a memory region. Each buffer can also point to data within the region using the data address and data length. This distinguishes three sections of the buffer: The headroom, the dataroom, and the tailroom.



- Headroom – memory region starting from the buffer's address up to the buffer's data address
- Dataroom – memory region starting from the buffer's data address with a length indicated by the buffer's data length
- Tailroom – memory region starting from the end of the dataroom to the end of the buffer
- Buffer length – the total length of the headroom, the dataroom, and the tailroom

Buffer Considerations

- There are multiple ways to create the buffer but, once created, it behaves in the same way (see section "DOCA Core#Inventories").
- The buffer may reference memory that is not accessible by the CPU (e.g., RDMA memory)
- The buffer is a thread-unsafe object
- The buffer can be used to represent non-continuous memory regions (scatter/gather list)
- The buffer does not own nor manage the data it references. Freeing a buffer does not affect the underlying memory.

Headroom

The headroom is considered user space. For example, this can be used by the user to hold relevant information regarding the buffer or data coupled with the data in the buffer's dataroom.

This section is ignored and remains untouched by DOCA libraries in all operations.

Dataroom

The dataroom is the content of the buffer, holding either data on which the user may want to perform different operations using DOCA libraries or the result of such operations.

Tailroom

The tailroom is considered as free writing space in the buffer by DOCA libraries (i.e., a memory region that may be written over in different operations where the buffer is used as output).

Buffer as Source

When using `doca_buf` as a source buffer, the source data is considered as the data section only (the dataroom).

Buffer as Destination

When using `doca_buf` as a destination buffer, data is written to the tailroom (i.e., appended after existing data, if any).

When DOCA libraries append data to the buffer, the data length is increased accordingly.

Scatter/Gather List

To execute operations on non-continuous memory regions, it is possible to create a buffer list. The list would be represented by a single `doca_buf` which represents the head of the list.

To create a list of buffers, the user must first allocate each buffer individually and then chain them. Once they are chained, they can be unchained as well:

- The chaining operation, `doca_buf_chain_list()`, receives two lists (heads) and appends the second list to the end of the first list
- The unchaining operation, `doca_buf_unchain_list()`, receives the list (head) and an element in the list, and separates them
- Once the list is created, it can be traversed using `doca_buf_get_next_in_list()`. `NULL` is returned once the last element is reached.
- The chaining operation, `doca_buf_chain_list_tail()`, appends a list head to a list tail. The application is responsible to maintain the list tail.

Passing the list to another library is same as passing a single buffer; the application sends the head of the list. DOCA libraries that support this feature can then treat the memory regions that comprise the list as one contiguous.

When using the buffer list as a source, the data of each buffer (in the dataroom) is gathered and used as continuous data for the given operation.

When using the buffer list as destination, data is scattered in the tailroom of the buffers in the list until it is all written (some buffers may not be written to).

Buffer Use Cases

The DOCA buffer is widely used by the DOCA acceleration libraries (e.g., DMA, compress, SHA). In these instances, the buffer can be provided as a source or as a destination.

Buffer use-case considerations:

- If the application wishes to use a linked list buffer and concatenate several **doca_buf**s to a scatter/gather list, the application is expected to ensure the library indeed supports a linked list buffer. For example, to check linked-list support for DMA memcpy task, the application may call **doca_dma_cap_task_memcpy_get_max_buf_list_len()**.
- Operations made on the buffer's data are not atomic unless stated otherwise
- Once a buffer has been passed to the library as part of the task, ownership of the buffer moves to the library until that task is complete

  ⚠ When using **doca_buf** as an input to some processing library (e.g., **doca_dma**), **doca_buf** must remain valid and unmodified until processing is complete.

- Writing to an in-flight buffer may result in anomalous behavior. Similarly, there are no guarantees for data validity when reading from an in-flight buffer.

Inventories

The inventory is the object responsible for allocating DOCA buffers. The most basic inventory allows allocations to be done without having to allocate any system memory. Other inventories involve enforcing that buffer addresses do not overlap.

Inventory Considerations

- All inventories adhere to zero allocation after start.
- Allocation of a DOCA buffer requires a data source and an inventory.
  - The data source defines where the data resides, what can access it, and with what permissions.
  - The data source must be created by the application. For creation of mmaps, see **doca_mmap**.
- The inventory describes the allocation pattern of the buffers, such as, random access or pool, variable-size or fixed-size buffers, and continuous or non-continuous memory.
- Some inventories require providing the data source, **doca_mmap**, when allocating the buffers, others require it on creation of the inventory.
- All inventory types are thread-unsafe.

Inventory Types

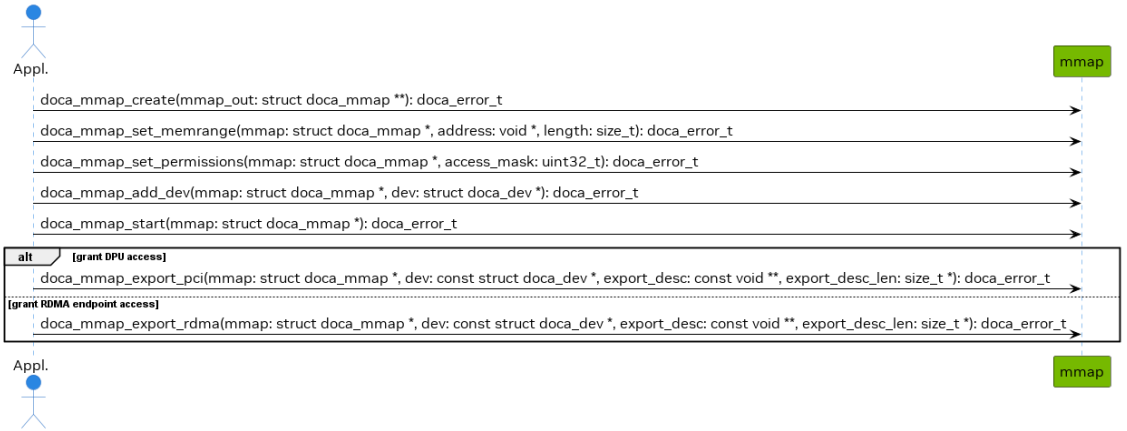
Inventory Type	Characteristics	When to Use	Notes
doca_buf_inventory	Multiple mmaps, flexible address, flexible buffer size.	When multiple sizes or mmaps are used.	Most common use case.

Inventory Type	Characteristics	When to Use	Notes
`doca_buf_array`	Single mmap, fixed buffer size. User receives an array of pointers to DOCA buffers. In case of DPA, mmap and buffer size can be unconfigured and later can be set from the DPA.	Use for creating DOCA buffers on GPU or DPA.	`doca_buf_arr` can be configured on the CPU and created on the GPU or DPA
`doca_bufpool`	Single mmap, fixed buffer size, address not controlled by the user.	Use as a pool of buffers of the same characteristics when buffer address is not important.	Slightly faster than `doca_buf_inventory`.

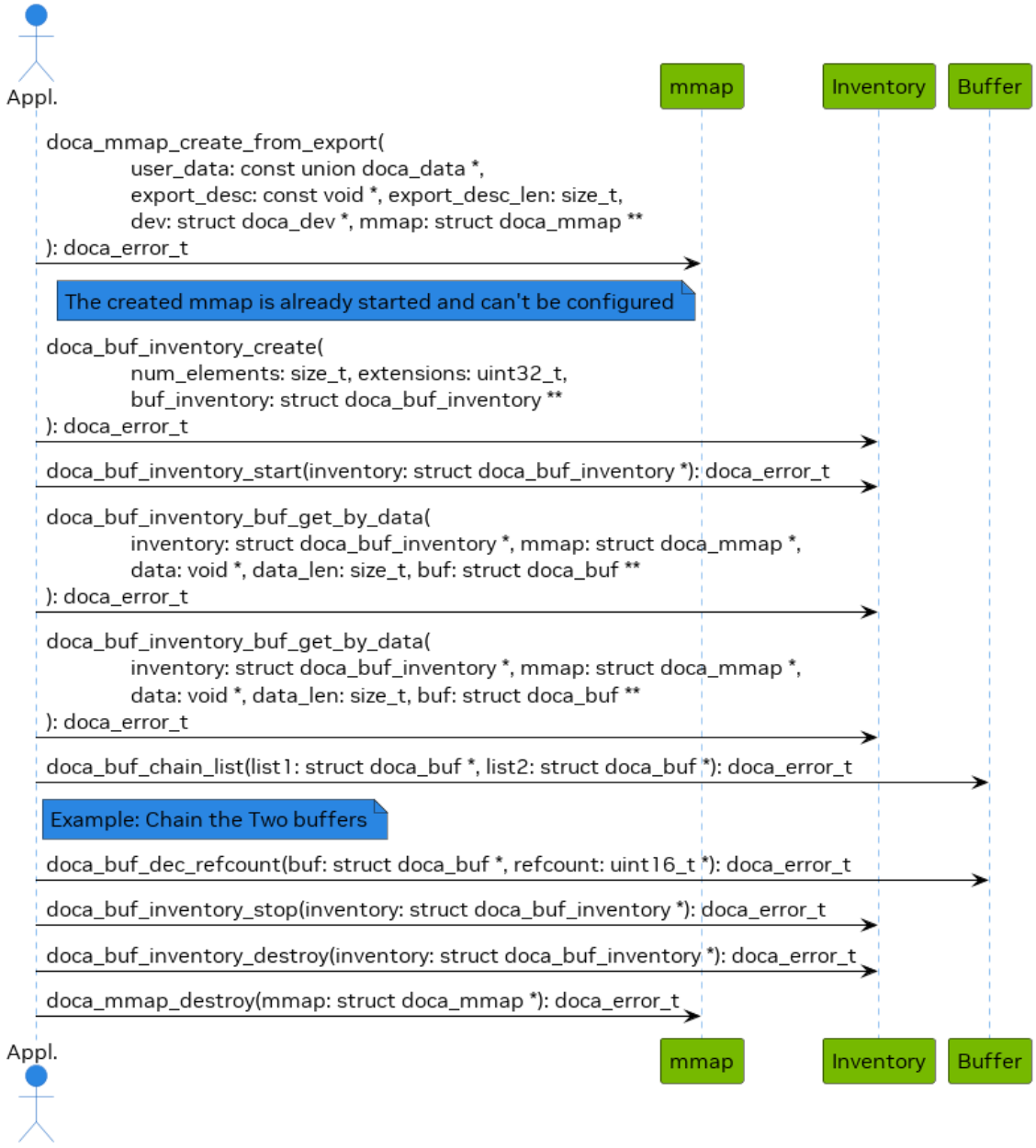
Example Flow

The following is a simplified example of the steps expected for exporting the host mmap to BlueField to be used by DOCA for direct access to the host memory (e.g., for DMA):

1. Create mmap on the host (see section "DOCA Core#Expected Flow" for information on how to choose the `doca_dev` to add to mmap if exporting to BlueField). This example adds a single `doca_dev` to the mmap and exports it so the BlueField/RDMA endpoint can use it.



2. Import to the BlueField/RDMA endpoint (e.g., use the mmap descriptor output parameter as input to **doca_mmap_create_from_export**).



5.4.1.1.4.4 DOCA Execution Model

The execution model is based on hardware processing on data and application threads. DOCA does not create an internal thread for processing data.

The workload is made up of tasks and events. Some tasks transform source data to destination data. The basic transformation is a DMA operation on the data which simply copies data from one memory location to another. Other operations allow users to receive packets from the network or involve calculating the SHA value of the source data and writing it to the destination.

For instance, a transform workload can be broken into three steps:

1. Read source data (**doca_buf** see memory subsystem).

2. Apply an operation on the read data (handled by a dedicated hardware accelerator).
3. Write the result of the operation to the destination (**doca_buf** see memory subsystem).

Each such operation is referred to as a task (**doca_task**).

Tasks describe operations that an application would like to submit to DOCA (hardware or BlueField). To do so, the application requires a means of communicating with the hardware/ BlueField. This is where the **doca_pe** comes into play. The progress engine (PE) is a per-thread object used to queue tasks to offload to DOCA and eventually receive their completion status.

doca_pe introduces three main operations:

1. Submission of tasks.
2. Checking progress/status of submitted tasks.
3. Receiving a notification on task completion (in the form of a callback).

A workload can be split into many different tasks that can be executed on different threads; each thread represented by a different PE. Each task must be associated to some context, where the context defines the type of task to be done.

A context can be obtained from some libraries within the DOCA SDK. For example, to submit DMA tasks, a DMA context can be acquired from **doca_dma.h** , whereas SHA context can be obtained using **doca_sha.h** . Each such context may allow submission of several task types.

A task is considered asynchronous in that once an application submits a task, the DOCA execution engine (hardware or BlueField) would start processing it, and the application can continue to do some other processing until the hardware finishes. To keep track of which task has finished, there are two modes of operation: polling mode and event-driven mode.

Requirements and Considerations

- The task submission/execution flow/API is optimized for performance (latency)
- DOCA does not manage internal (operating system) threads. Rather, progress is managed by application resources (calling DOCA API in polling mode or waiting on DOCA notification in event-driven mode).
- The basic object for executing the task is a **doca_task** . Each task is allocated from a specific DOCA library context.
- **doca_pe** represents a logical thread of execution for the application and tasks submitted to the progress engine (PE)

  > ⚠ PE is not thread safe and it is expected that each PE is managed by a single application thread (to submit a task and manage the PE).

- Execution-related elements (e.g., **doca_pe** , **doca_ctx** , **doca_task**) are opaque and the application performs minimal initialization/configuration before using these elements
- A task submitted to PE can fail (even after the submission succeeds). In some cases, it is possible to recover from the error. In other cases, the only option is to reinitialize the relevant objects.
- PE does not guarantee order (i.e., tasks submitted in certain order might finish out-of-order). If the application requires order, it must impose it (e.g., submit a dependent task once the previous task is done).

- A PE can either work in polling mode or event-driven mode, but not in both at same time
- All DOCA contexts support polling mode (i.e., can be added to a PE that supports polling mode)

DOCA Context

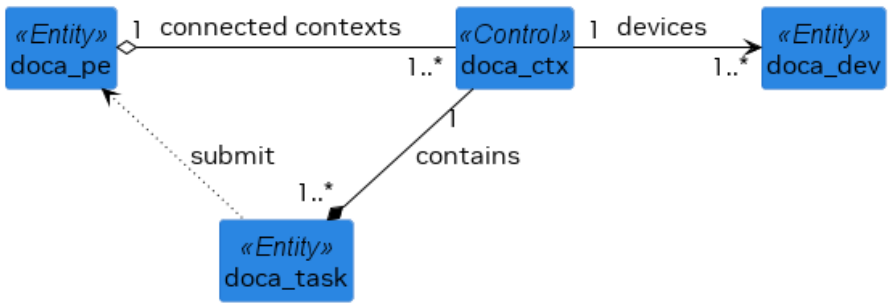
DOCA Context (**struct doca_ctx**) defines and provides (implements) task/event handling. A context is an instance of a specific DOCA library (i.e., when the library provides a DOCA Context, its functionality is defined by the list of tasks/events it can handle). When more than one type of task is supported by the context, it means that the supported task types have a certain degree of similarity to implement and utilize common functionality.

The following list defines the relationship between task contexts:

- Each context utilizes at least one DOCA Device functionality/accelerated processing capabilities
- For each task type there is one and only context type supporting it
- A context virtually contains an inventory per supported task type
- A context virtually defines all parameters of processing/execution per task type (e.g., size of inventory, device to accelerate processing)

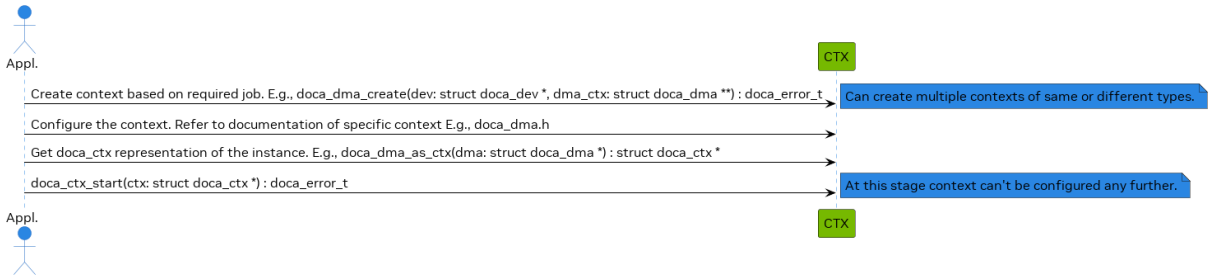
Each context needs an instance of progress engine (PE) as a runtime for its tasks (i.e., a context must be associated with a PE to execute tasks).

The following diagram shows the high-level (domain model) relations between various DOCA Core entities.



1. **doca_task** is associated to a relevant **doca_ctx** that executes the task (with the help of the relevant **doca_dev**).
2. **doca_task** , after it is initialized, is submitted to **doca_pe** for execution.
3. **doca_ctx** s are connected to the **doca_pe** . Once a **doca_task** is queued to **doca_pe** , it is executed by the **doca_ctx** that is associated with that task in this PE.

The following diagram describes the initialization sequence of a context:



After the context is started, it can be used to enable the submission of tasks to a PE based on the types of tasks that the context supports. See section "DOCA Core#DOCA Progress Engine" for more information.

> ⚠ Context is a thread-unsafe object which can be connected to a single PE only.

Configuration Phase

A DOCA context must be configured before attempting to start it using **`doca_ctx_start()`**. Some configurations are mandatory (e.g., providing **`doca_dev`**) while others are not.

- Configurations can be useful to allow certain tasks/events, to enable features which are disabled by default, and to optimize performance depending on a specific workload.
- Configurations are provided using setter functions. Refer to context documentation for a list of mandatory and optional configurations and their corresponding APIs.
- Configurations are provided after creating the context and before starting it. Once the context is started, it can no longer be configured unless it is stopped again.

Examples of common configurations:

- Providing a device – usually done as part of the create API
- Enabling tasks or registering to events – all tasks are disabled by default

Execution Phase

Once context configuration is complete, the context can be used to execute tasks. The context executes the tasks by offloading the workload to hardware, while software polls the tasks (i.e., waits) until they are complete.

In this phase, an application uses the context to allocate and submit asynchronous tasks, and then polls tasks (waits) until completion.

The application must build an event loop to poll the tasks (wait), utilizing one of the following modes:

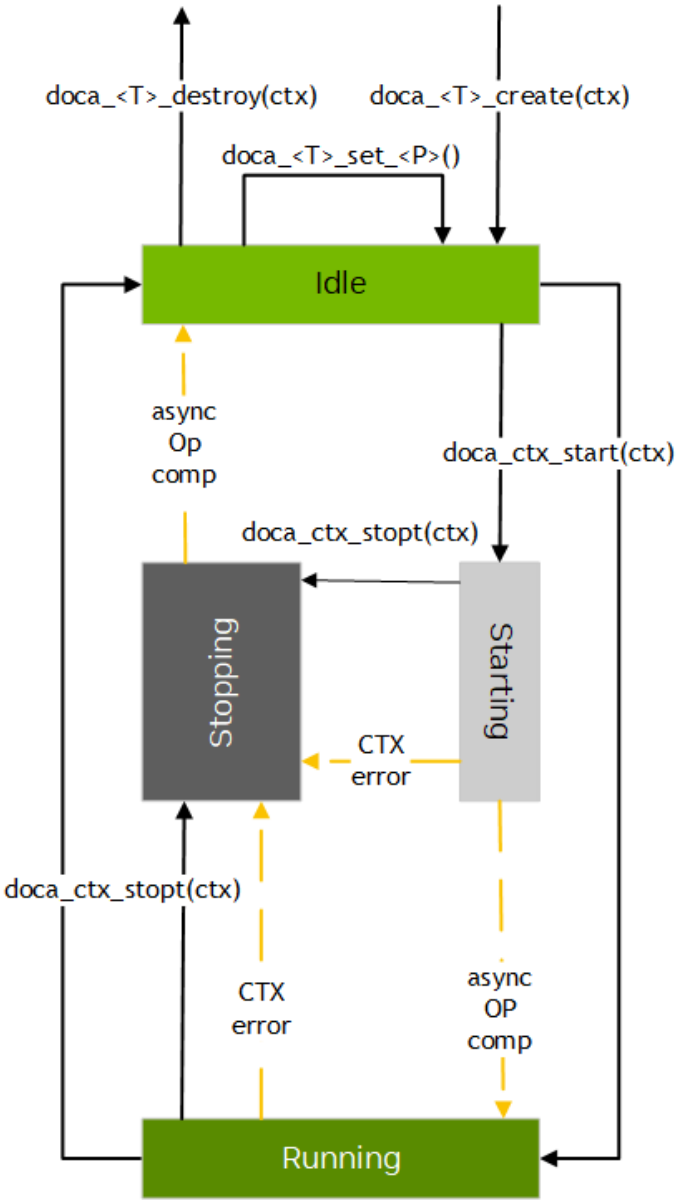
- DOCA Core#Polling Mode
- DOCA Core#Notification-driven Mode

In this phase, the context and all core objects perform zero allocations by utilizing memory pools. It is recommended that the application utilizes same approach for its own logic.

State Machine

State	Description
Idle	• 0 in-flight tasks • On init (right after `doca_<T>_create(ctx)`): All configuration APIs enabled • On reconf (on transition from stopping state): Some configuration APIs enabled
Starting	This state is mandatory for CTXs where transition to running state is conditioned by one or more async op completions/external events. For example, when a client connects to comm channel, it enters running state. Waiting for state change can be terminated by a voluntary (user) `doca_ctx_stop()` call or involuntary context state change due to internal error.
Running	• Task allocation/submission enabled (disabled in all other states) • All configuration APIs are disabled
Stopping	• Preparation before stopped state • Clean all in-flight tasks that may not complete in near future • Procedures relying on external entity actions should be terminated by CTX logic

The following diagram describes DOCA Context state transitions:



Internal Error

DOCA Context states can encounter internal errors at any time. If the state is starting or running, an internal error can cause an involuntary transition to stopping state.

For instance, an involuntary transition from running to stopping can happen when a task execution fails. This results in a completion with error for the failed task and all subsequent task completions.

After stopping, the state may become idle. However, **doca_ctx_start()** may fail if there is a configuration issue or if an error event prevented proper transition to starting or running state.

DOCA Task

A task is a unit of (functional/processing) workload offload-able to hardware. Most tasks utilize NVIDIA® BlueField® and NVIDIA® ConnectX® hardware to provide accelerated processing of the

workload defined by the task. Tasks are asynchronous operations (e.g., tasks submitted for processing via non-blocking **doca_task_submit()** API).

Upon task completion, the preset completion callback is executed in context of **doca_pe_progress()** call. The completion callback is a basic/generic property of the task, similar to user data. Most tasks are IO operations executed/accelerated by NVIDIA device hardware.

Task Properties

Task properties share generic properties which are common to all task types and type-specific properties. Since task structure is opaque (i.e., its content not exposed to the user), the access to task properties provided by set/get APIs.

The following are generic task properties:

- Setting completion callback – it has separate callbacks for successful completion and completion with failure.
- Getting/setting user data – used in completion callback as some structure associated with specific task object.
- Getting task status – intended to retrieve error code on completion with failure.

For each task there is only one owner: a context object. The **doca_task_get_ctx()** API can be used to get the generic context object.

The following are generic task APIs:

- Allocating and freeing from CTX (internal/virtual) inventory
- Configuring via setters (or init API)
- Submit-able (i.e., implements **doca_task_submit(task)**)

Upon completion, there is a set of getters to access the results of the task execution.

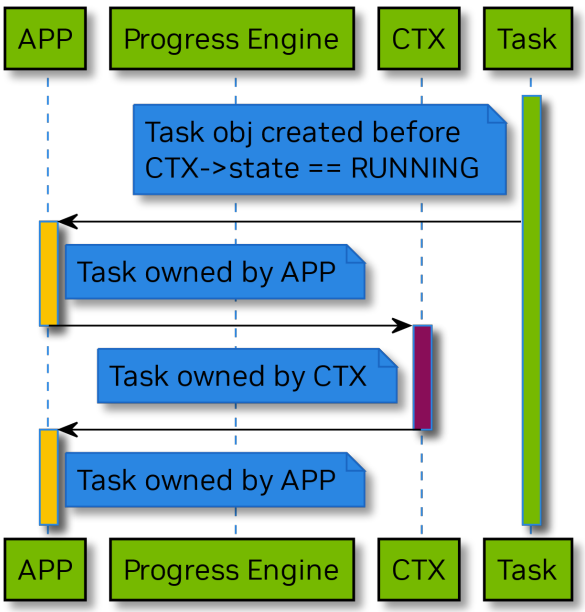
Task Lifecycle

This section describes the lifecycle of a DOCA Task, which is defined by a clear ownership model. From an application's perspective, a DOCA Context (CTX) provides a "virtual task inventory":

The entire task lifecycle is bound by the state of the DOCA Context:

- Begins: An application can only allocate a task when the owning CTX enters the **RUNNING** state.
- Ends: The application can no longer allocate tasks once the CTX leaves the **RUNNING** state (e.g., enters the **STOPPED** state).

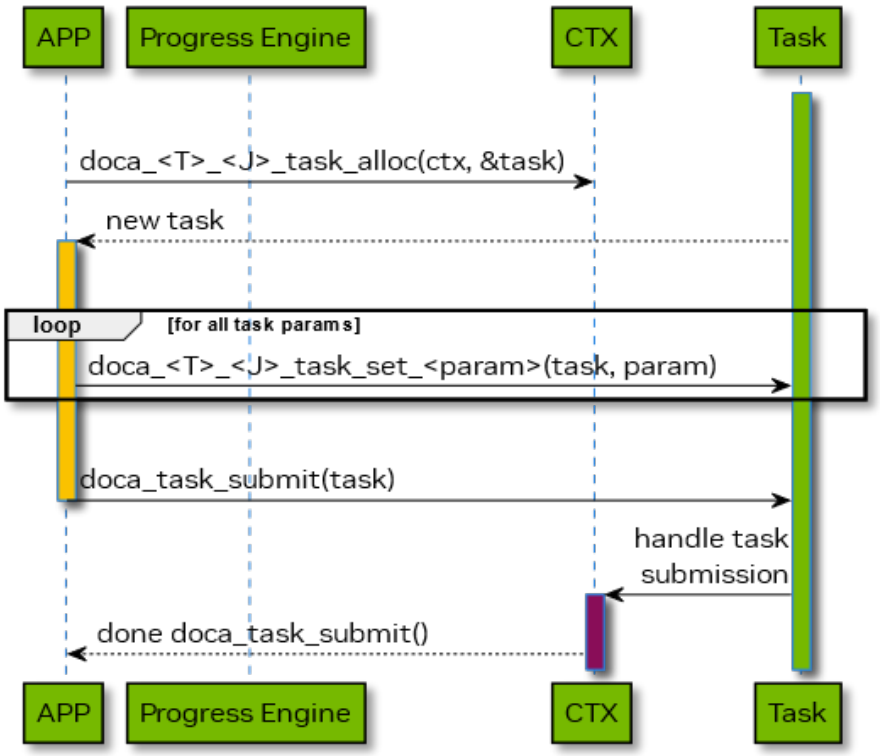
The core concept is that ownership of a task object is passed between the Application (APP) and the Context (CTX).



Phase 1: Task Allocation and Submission

This phase covers acquiring a task, setting its parameters, and submitting it for processing.

1. Allocation (APP Gains Ownership): The application requests a task from the context's inventory by calling a function like **`doca_<T>_<J>_task_alloc(ctx, &task)`**. Upon successful allocation, the task is owned by the application.
2. Configuration (APP Maintains Ownership): The application sets the task's parameters using functions templated as **`doca_<T>_<J>_task_set_<param>(task, param)`**. After each "set" call, the application retains ownership of the task object.
3. Submission (CTX Gains Ownership): Once all parameters are set, the application calls **`doca_task_submit(task)`**. This action passes ownership of the task object to the CTX for processing.

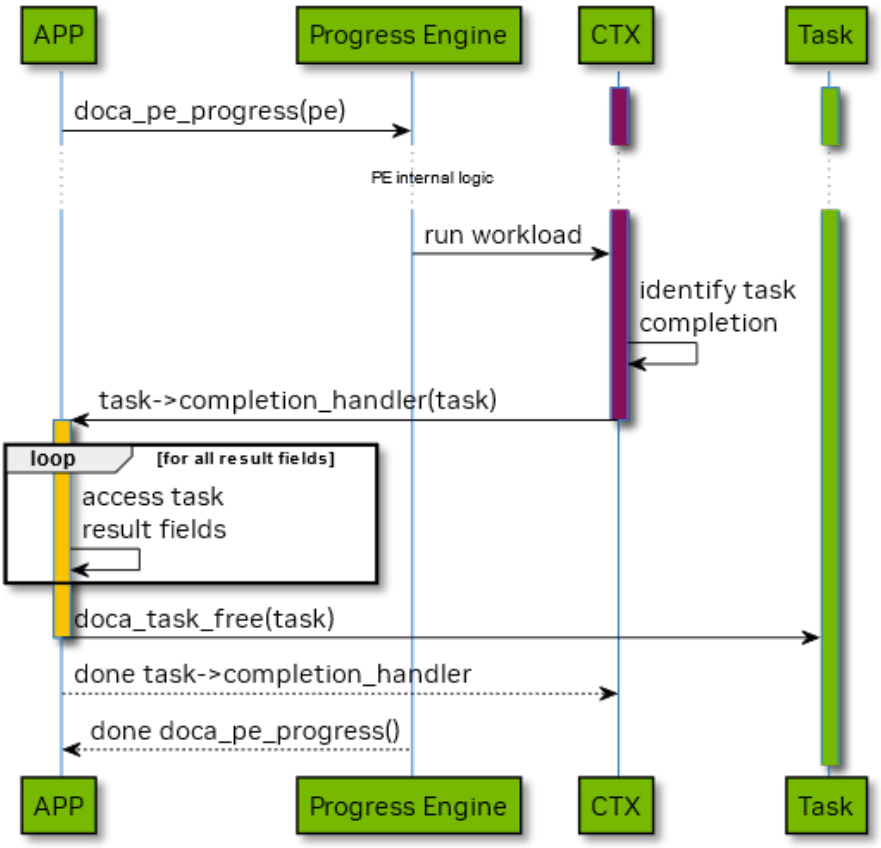


Phase 2: Task Processing and Completion

Once a task is submitted, the Progress Engine (PE) is responsible for its execution and completion.

1. Processing (Driven by APP): The task is processed by the hardware or software backend. The application must periodically call **`doca_pe_progress(pe)`** to drive this process. This function runs the PE's internal logic , executes the workload , and identifies completed tasks.
2. Completion (APP Regains Ownership): When the PE identifies a completed task, it invokes the application-provided completion handler (e.g., **`task->completion_handler(task)`**). As soon as this handler is called, ownership of the task is passed back to the application.

Inside the completion handler, the application can safely access the task's result fields.



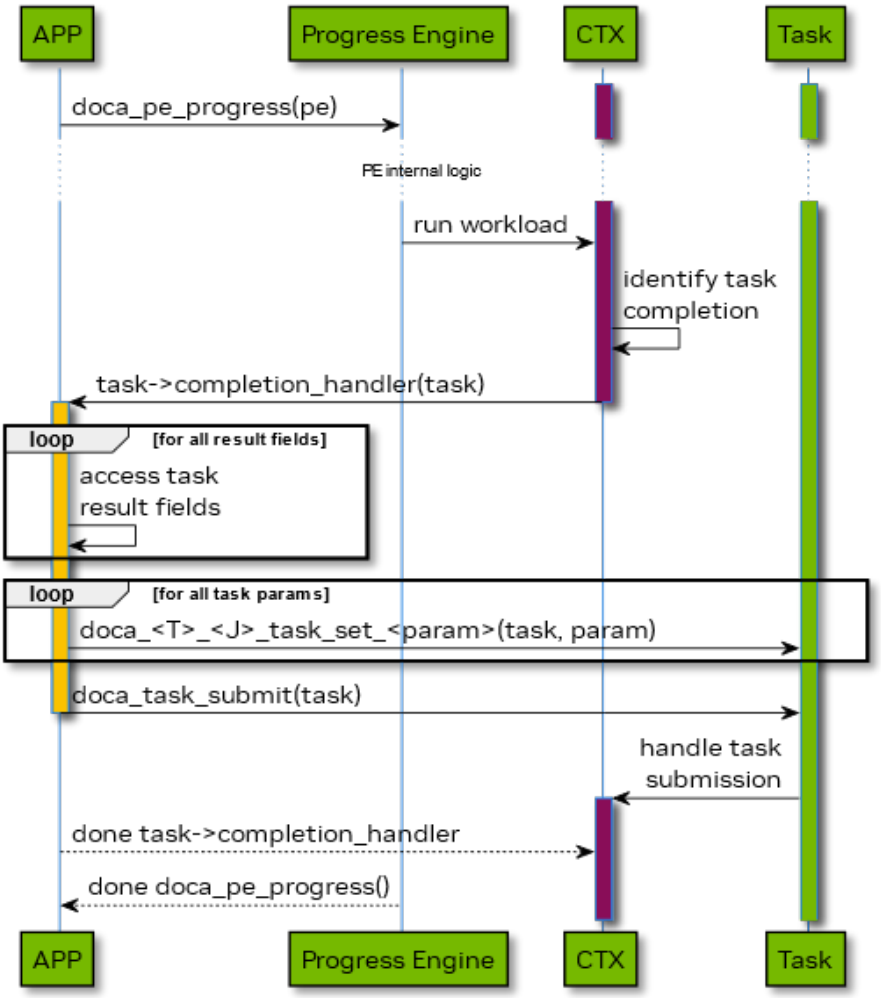
Phase 3: Post-Completion (Free or Re-submit)

After a task is completed and the application regains ownership, it has two choices inside the completion handler: free the task or re-submit it.

Option A: Freeing the Task

If the task is no longer needed, the application calls **`doca_task_free(task)`**. This action:

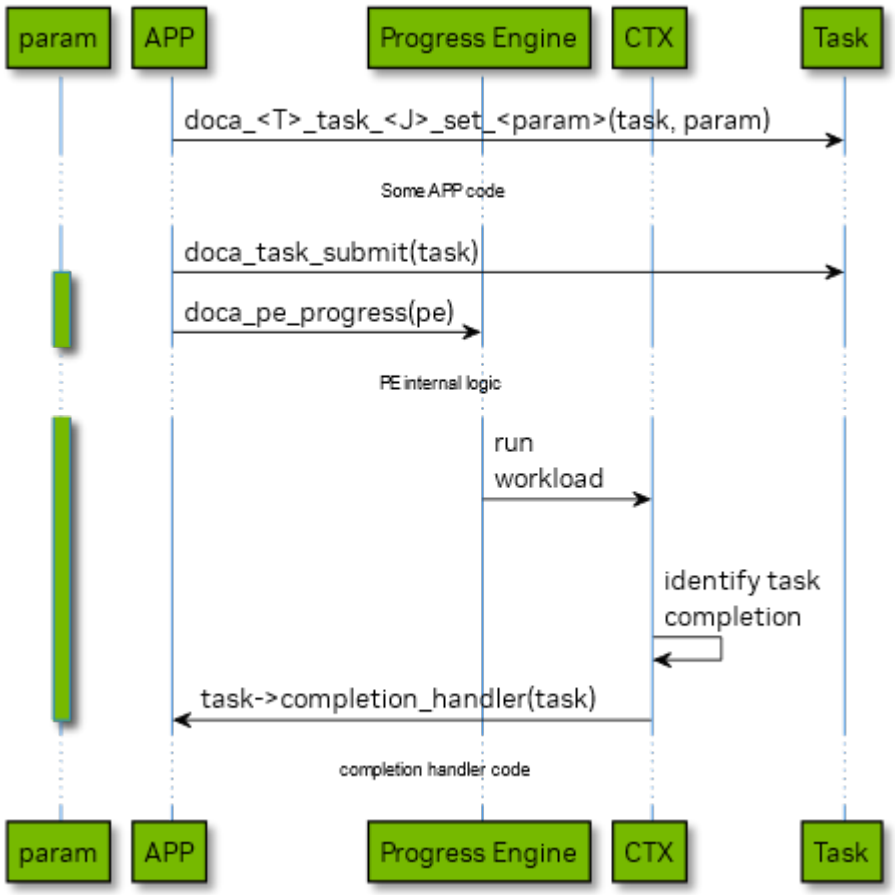
- Passes ownership of the task back to the CTX.
- Returns the task to the context's virtual inventory as an uninitialized, pre-allocated object, ready to be allocated again.



Option B: Re-submitting the Task (Task Chaining)

Instead of freeing the task, the application can immediately re-use it. This is a common pattern for high-performance "task chaining." From within the completion handler, the application can:

1. Access the task's results.
2. Set new parameters for the *next* operation (e.g., **doca_<T>_<J>_task_set_<param>(task, param)**).
3. Call **doca_task_submit(task)** again.



Note on Task Parameter Lifecycle

The lifecycle of a task's *parameters* (e.g., data buffers) is as critical as the task object itself.

- When the application calls **doca_task_submit(task)**, ownership of all task arguments (parameters) is passed to the DOCA Context.
- Ownership of these arguments is passed back to the application only when the task's **completion_handler** is invoked.

The application must not modify, destroy, or free any task-related arguments or objects while they are owned by the DOCA Context (i.e., after submission and before completion). Furthermore, task results are only valid within the scope of the completion callback. They should be considered invalidated after exiting the handler or after calling **doca_task_free()** or any **..._task_set_<param>()** function.

DOCA Progress Engine

The progress engine (PE) enables asynchronous processing and handling of multiple tasks and events of different types in a single-threaded execution environment. It is an event loop for all context-based DOCA libraries, with I/O completion being the most common event type.

PE is designed to be thread unsafe (i.e., it can only be used in one thread at a time) but a single OS thread can use multiple PEs. The user can assign different priorities to different contexts by

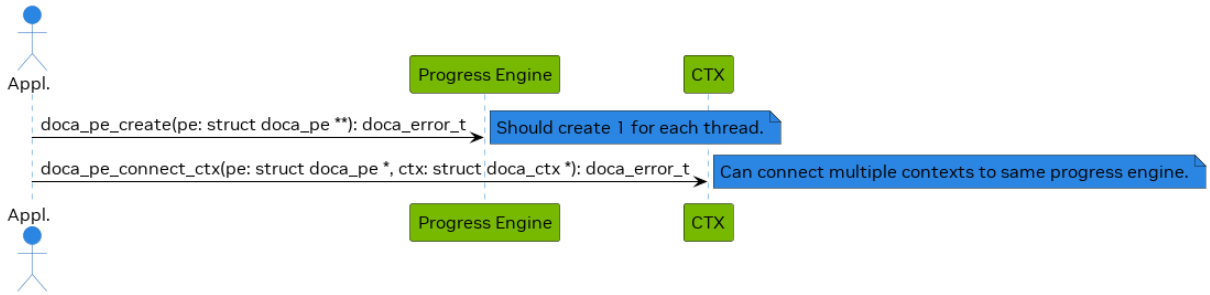
adding them to different PEs and adjusting the polling frequency for each PE accordingly. Another way to view the PE is as a queue of workload units that are scheduled for execution.

There are no explicit APIs to add and/or schedule a workload to/on a PE but a workload can be added by:

- Adding a DOCA context to PE
- Registering a DOCA event to probe (by the PE) and executing the associated handler if the probe is positive

PE is responsible for scheduling workloads (i.e., picking the next workload to execute). The order of workload execution is independent of task submission order, event registration order, or order of context associations with a given PE object. Multiple task completion callbacks may be executed in an order different from the order of related task submissions.

The following diagram describes the initialization flow of the PE:



After a PE is created and connected to contexts, it can start progressing tasks which are submitted to the contexts. Refer to context documentation to find details such as what tasks can be submitted using the context.

Note that the PE can be connected to multiple contexts. Such contexts can be of the same type or of different types. This allows submitting different task types to the same PE and waiting for any of them to finish from the same place/thread.

After initializing the PE, an application can define an event loop using one of these modes:

- Polling mode
- Blocking (notification-driven) mode

PE as Event Loop Mode of Operation

All completion handlers for both tasks and events are strictly executed within the context of the **doca_pe_progress()** function.

When invoked, **doca_pe_progress()** iterates through and executes every workload unit scheduled for execution. During this loop, the function evaluates the workload and resolves based on the following conditions:

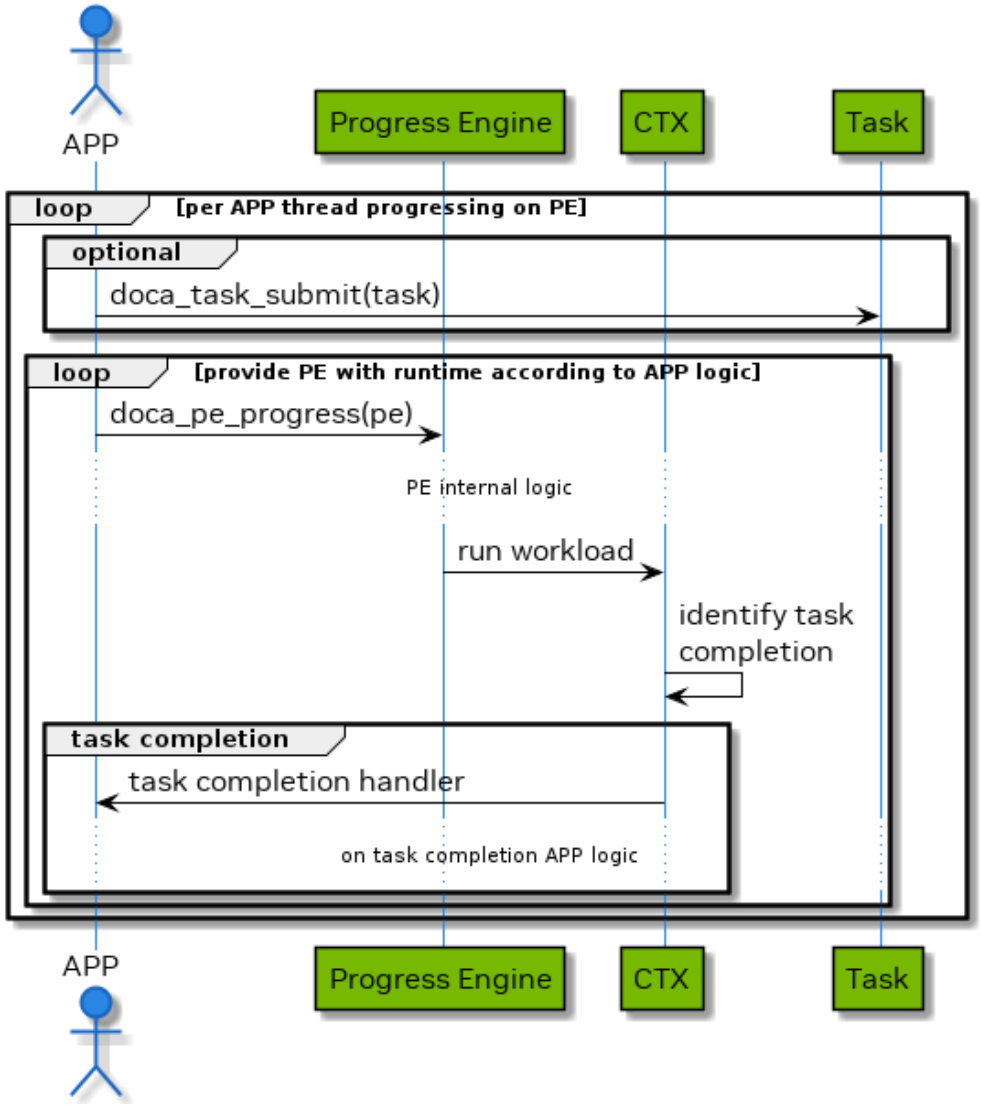
Condition	Action	Return Status
Task completion	Executes the associated handler and breaks the loop.	**made some progress**

Condition	Action	Return Status
Positive event probe	Executes the associated handler and breaks the loop.	made some progress
Considerable partial progress	Breaks the loop if significant internal progress is achieved (contributing to a future task completion or positive event probe).	made some progress
Default fallback	Reaches the end of the loop without meeting any of the above conditions and terminates.	no progress

Polling Mode

In this mode, the application submits a task and then does busy-wait to find out when the task has completed.

The following diagram demonstrates this sequence:



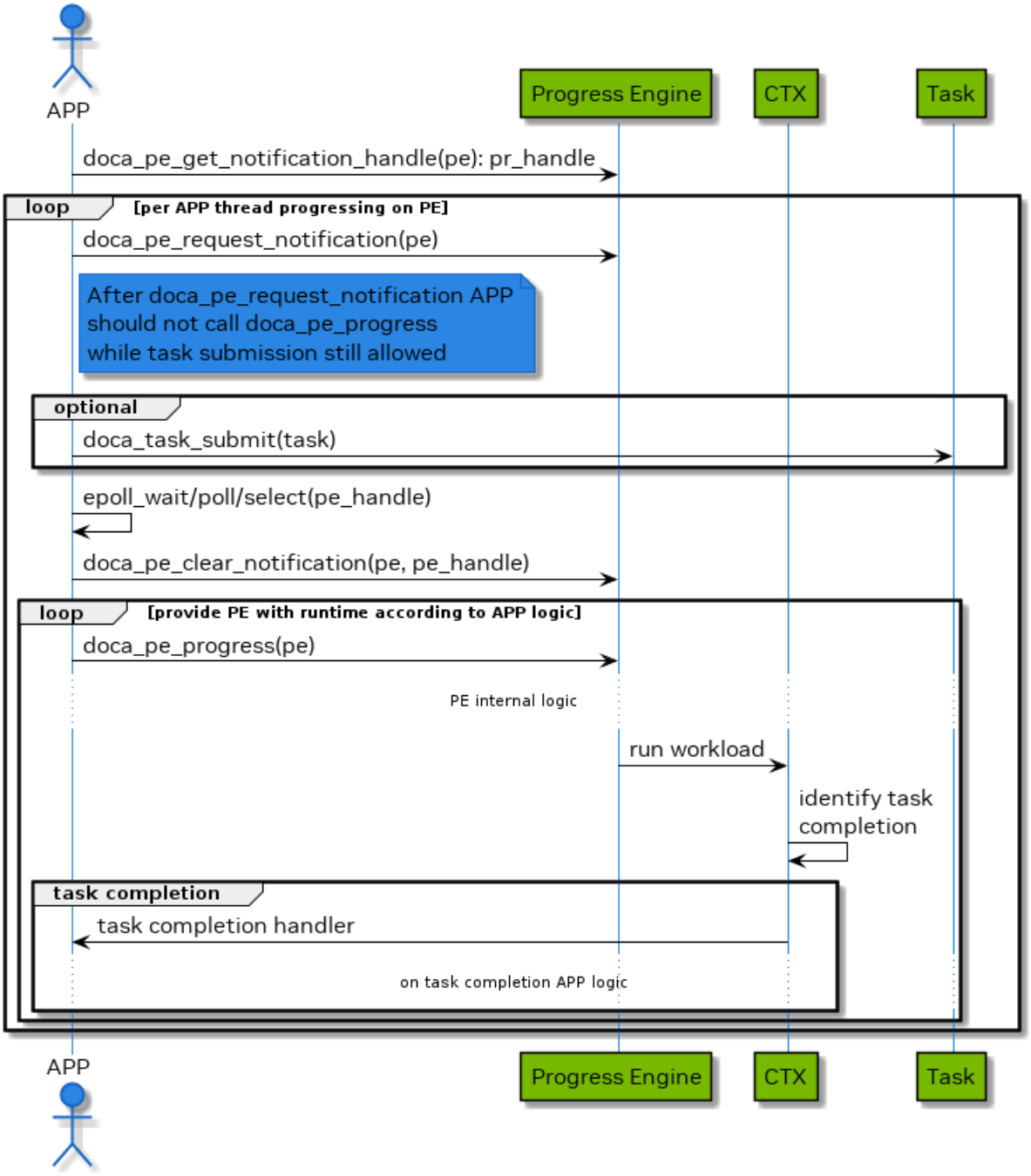
1. The application submits all tasks (one or more) and tracks the number of task completions to know if all tasks are done.
2. The application waits for a task to complete by consecutive polls on **doca_pe_progress()**.
    a. If **doca_pe_progress()** returns 1, it means progress is being made (i.e., some task completed or some event handled).
    b. Each time a task is completed or an event is handled, its preset completion or event handling callback is executed accordingly.
    c. If a task is completed with an error, preset task completion with error callback is executed (see section "Error Handling").
3. The application may add code to completion callbacks or event handlers for tracking the amount of completed and pending workloads.

In this mode, the application is always using the CPU even when it is doing nothing (busy-wait).

Blocking Mode - Notification Driven

In this mode, the application submits a task and then waits for a notification to be received before querying the status.

The following diagram demonstrates this sequence:



1. The application gets a notification handle from the **doca_pe** representing a Linux file descriptor which is used to signal the application that some work has finished.
2. The application then arms the PE with **doca_pe_request_notification()**.

> ⚠ This must be done every time an application is interested in receiving a notification from the PE.

> ⚠ After `doca_pe_request_notification()`, no calls to `doca_pe_progress()` are allowed. In other words, `doca_pe_request_notification()` should be followed by `doca_pe_clear_notification` before any calls to `doca_pe_progress()`.

3. The application submits a task.
4. The application waits (e.g., Linux epoll/select) for a signal to be received on the `pe-fd`.
5. The application clears the notifications received, notifying the PE that a signal has been received and allowing it to perform notification handling.
6. The application attempts to handle received notifications via (multiple) calls to `doca_pe_progress()`.

   > ⚠ There is no guarantee that the call to `doca_pe_progress()` would execute any task completion/event handler, but the PE can continue the operation.

7. The application handles its internal state changes caused by task completions and event handlers called in the previous step.
8. Repeat steps 2-7 until all tasks are completed and all expected events are handled.

Progress Engine versus Epoll

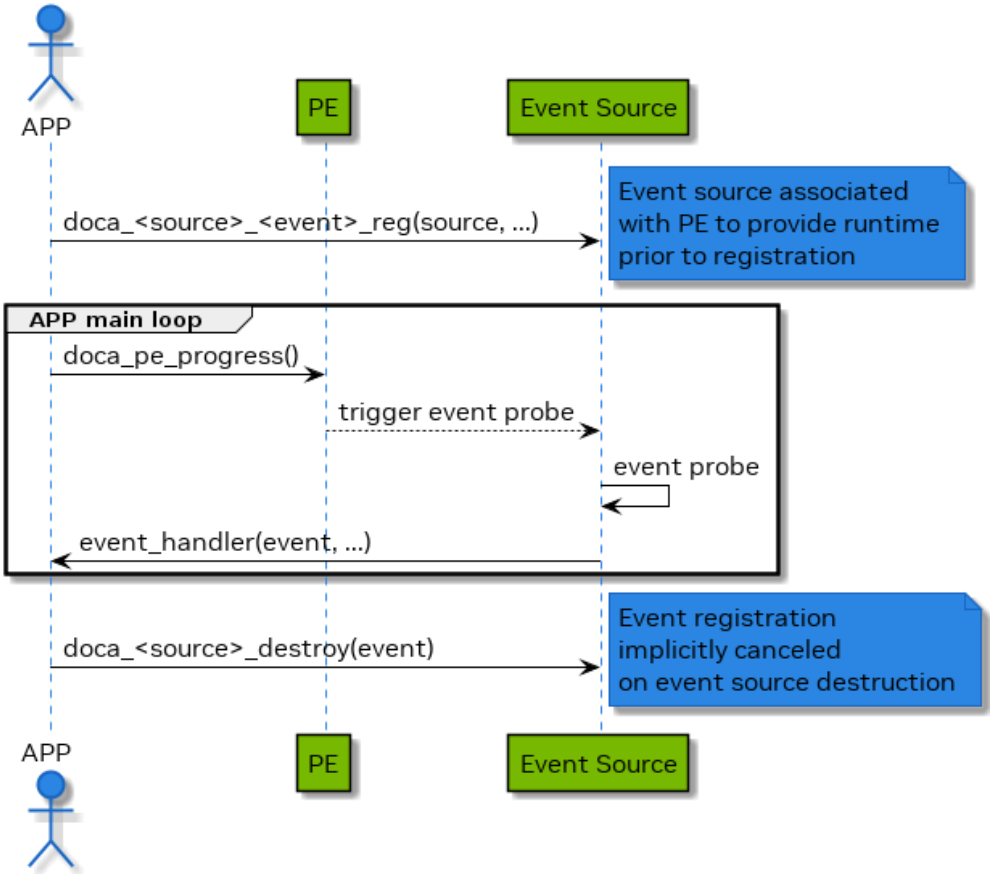
The epoll mechanism in Linux and the DOCA PE handles high concurrency in event-driven architectures. Epoll, like a post office, tracks "mailboxes" (file descriptors) and notifies the "postman" (the `epoll_wait` function) when a "letter" (event) arrives. DOCA PE, like a restaurant, uses a single "waiter" to handle "orders" (workload units) from "customers" (DOCA contexts). When an order is ready, it is placed on a "tray" (task completion handler/event handler execution) and delivered in the order received. Both systems efficiently manage resources while waiting for events or tasks to complete.

DOCA Event

An event is a type of occurrence that can be detected or verified by the DOCA software, which can then trigger a handler (a callback function) to perform an action. Events are associated with a specific source object, which is the entity whose state or attribute change defines the event's occurrence. For example, a context state change event is caused by the change of state of a context object.

To register an event, the user must call the `doca_<event_type>_reg(pe, ...)` function, passing a pointer to the user handler function and an opaque argument for the handler. The user must also associate the event handler with a PE, which is responsible for running the workloads that involve event detection and handler execution.

Once an event is registered, it is periodically checked by the `doca_pe_progress()` function, which runs in the same execution context as the PE to which the event is bound. If the event condition is met, the handler function is invoked. Events are not thread-safe objects and should only be accessed by the PE to which they are bound.

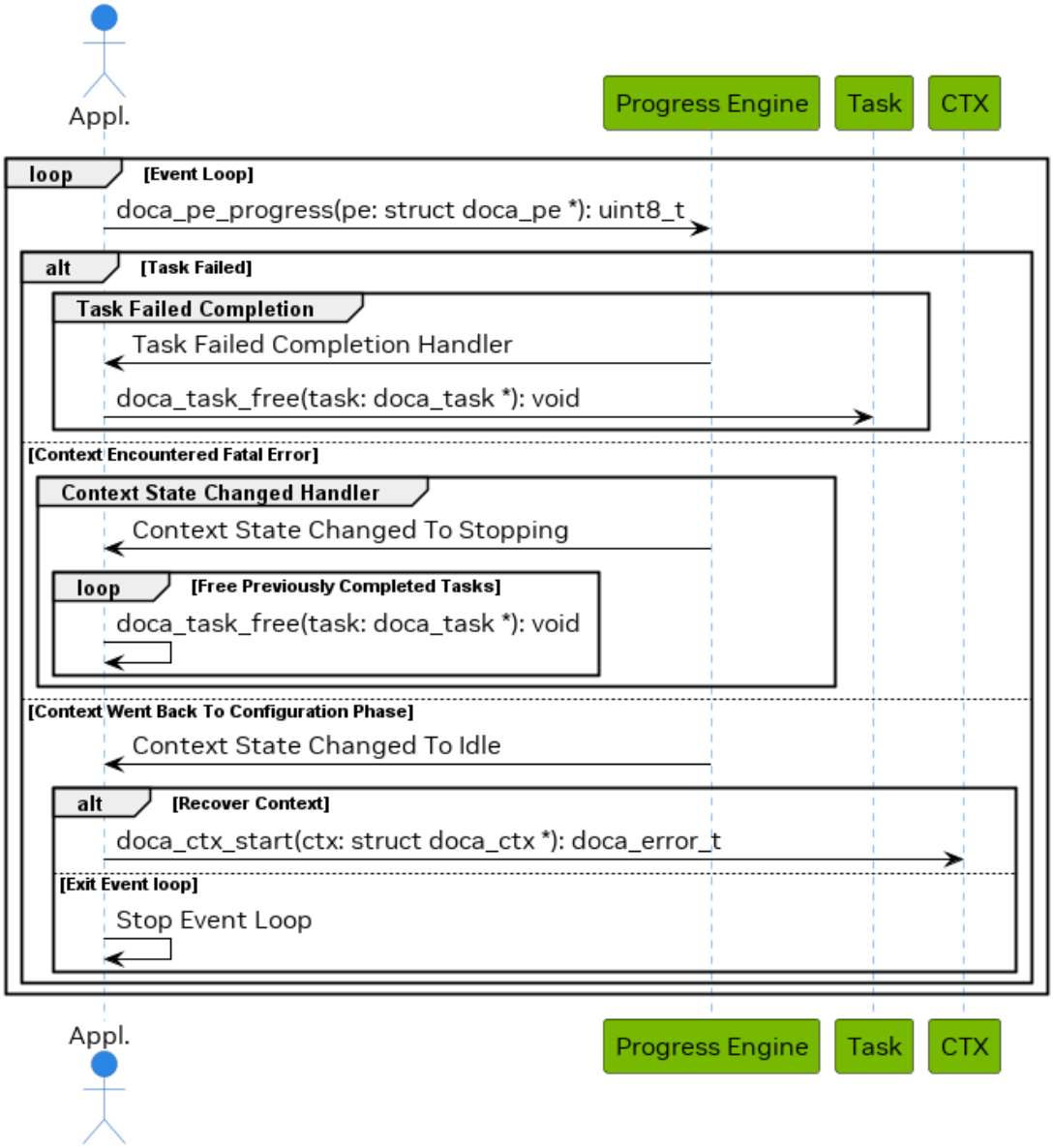


Error Handling

After a task is submitted successfully, consequent calls to **doca_pe_progress()** may fail (i.e., task failure completion callback is called).

Once a task fails, the context may transition to stopping state, in this state, the application has to progress all in-flight tasks until completion before destroying or restarting the context.

The following diagram shows how an application may handle an error from **doca_pe_progress()**:



1. Application runs event loop.
2. Any of the following may happen:
    - [Optional] Task fails, and the task failed completion handler is called
        - This may be caused by bad task parameters or another fatal error
        - Handler releases the task and all associated resources
    - [Optional] Context transitions to stopping state, and the context state changed handler is called
        - This may be caused by failure of a task or another fatal error
        - In this state, all in-flight tasks are guaranteed to fail
        - Handler releases tasks that are not in-flight if such tasks exist
    - [Optional] Context transitions to idle state, and the context state changed handler is called
        - This may happen due to encountering an error and the context does not have any resources that must be freed by the application

- In this case, the application may decide to recover the context by calling start again or it may decide to destroy the context and possibly exit the application

Task and Event Batching

DOCA Batching is an approach for grouping multiple tasks or events of the same type and handling them as a single unit. DOCA offers two options of achieving this as described in the following subsections.

Batch Task/Event

In this batching option, a library (e.g., **doca_eth_txq**) offers a task that represents a batched operation (e.g., sending multiple packets), the task is considered a batch task and has a task type that is separate from the non-batched operation (e.g., sending a single packet).

To submit the batch task, the user is required to build the batch and then submit it at once, similar to submitting a regular task.

The completion of the batch is based on the completion of all items in the batch and is handled as the completion of a single unit. This allows for multiple DOCA Task initialization/submission and multiple DOCA Task/Event completion handling in a single API call (see DOCA Ethernet for example).

Iterative Batch

In this batching option, it is possible to utilize existing task types to build a batch operation, where each task within the batch is submitted individually and each task receives its own completion.

Furthermore, the batch is built iteratively, where the user is not required to have information for the entire batch ahead of time.

To utilize this option, the user can submit each task in the batch using an extended submit API **doca_task_submit_ex** while providing additional submit flags.

The extended submit API is similar to a regular submit API (**doca_task_submit**) but with the ability to receive submit flags. These flags are used as hints to the library that executes the tasks. They can have implications on the current task but may also have implications on previously submitted flags, as described in the following table:

Submit Flag	Effect on Current Task		Effect on Previous Tasks		Default Behavior of doca_task_submit	Comments
	Flag Provided	Flag not Provided	Flag Provided	Flag not Provided		
DOCA_TASK_SUBMIT_FLAG_FLUSH	Task is submitted for hardware execution immediately, and is considered "flushed".	Task may not be submitted for hardware execution, and is considered "unflushed".	All previous tasks which are considered unflushed become flushed.	None	Flag is provided	As long as the task is unflushed, it never completes. The flag allows batching such that multiple tasks are flushed at once, instead of individually.

Submit Flag	Effect on Current Task		Effect on Previous Tasks		Default Behavior of doca_task_submit	Comments
	Flag Provided	Flag not Provided	Flag Provided	Flag not Provided		
DOCA_TASK_SUBMIT_FLAG_OPTIMIZE_REPORTS	The user does not receive task completion after hardware has completed execution of the task, and the completion is considered "unreported".	The user receives task completion after hardware has completed execution of the task, and the completion is considered "reported".	None	Once the hardware completes execution of this task, all previous unreported completions become reported.	Flag is not provided	As long as the task is unreported, the user would never know that it has been completed. The completion of a task is reported through a completion callback using the progress engine. The library does not guarantee any order of execution / completion of tasks.

Submit Flag	Effect on Current Task		Effect on Previous Tasks		Default Behavior of doca_task_submit	Comments
	Flag Provided	Flag not Provided	Flag Provided	Flag not Provided		
						The flag allows batching, such that multiple task completions are reported using a single hardware completion, instead of receiving a completion for every task.

We don't have a way to export this macro.

DOCA Graph Execution

DOCA Graph facilitates running a set of actions (tasks, user callbacks, graphs) in a specific order and dependencies. DOCA Graph runs on a DOCA progress engine.

DOCA Graph creates graph instances that are submitted to the progress engine (`doca_graph_instance_submit`).

Nodes

DOCA Graph is comprised of context, user, and sub-graph nodes. Each of these types can be in any of the following positions in the network:

- Root nodes – a root node does not have a parent. The graph can have one or more root nodes. All roots begin running when the graph instance is submitted.
- Edge nodes – an edge node is a node that does not have child nodes connected to it. The graph instance is completed when all edge nodes are completed.
- Intermediate node – a node connected to parent and child nodes

Context Node

A context node runs a specific DOCA task and uses a specific DOCA context (`doca_ctx`). The context must be connected to the progress engine before the graph is started.

The task lifespan must be longer or equal to the life span of the graph instance.

User Node

A user node runs a user callback to facilitate performing actions during the run time of the graph instance (e.g., adjust next node task data, compare results).

Sub-graph Node

A sub-graph node runs an instance of another graph.

Using DOCA Graph

1. Create the graph using **doca_graph_create**.
2. Create the graph nodes (e.g., **doca_graph_node_create_from_ctx**).
3. Define dependencies using **doca_graph_add_dependency**.

   ⚠ DOCA graph does not support circle dependencies (e.g., A => B => A).
4. Start the graph using **doca_graph_start**.
5. Create the graph instance using **doca_graph_instance_create**.
6. Set the nodes data (e.g., **doca_graph_instance_set_ctx_node_data**).
7. Submit the graph instance to the PE using **doca_graph_instance_submit**.
8. Call **doca_pe_progress** until the graph callback is invoked.
   - Progress engine can run graph instances and standalone tasks simultaneously.

DOCA Graph Limitations

- DOCA Graph does not support circle dependencies
- DOCA Graph must contain at least one context node. A graph containing a sub-graph with at least one context node is a valid configuration.

DOCA Graph Sample

The graph sample is based on the DOCA DMA library. The sample copies 2 buffers using DMA.

The graph ends with a user callback node that compares source and destinations.

Running DOCA Graph Sample

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample:

   ```
   cd /opt/mellanox/doca/samples/doca_common/graph/
   meson build
   ninja -C build
   ```
3. Sample (e.g., **doca_graph**) usage:

```
./build/doca_graph
```

No parameters required.

Alternative Data Path

DOCA Progress Engine utilizes the CPU to offload data path operations to hardware. However, some libraries support utilization of DPA and/or GPU.

Considerations:

- Not all contexts support alternative datapath
- Configuration phase is always done on CPU
- Datapath operations are always offloaded to hardware. The unit that offloads the operation itself can be either CPU/DPA/GPU.
- The default mode of operation is CPU
- Each mode of operation introduces a different set of APIs to be used in execution path. The used APIs are mutually exclusive for specific context instance.

DPA

Users must first refer to the programming guide of the relevant context (e.g., DOCA RDMA) to check if datapath on DPA is supported. Additionally, the guide provides what operations can be used.

To set the datapath mode to DPA, acquire a DOCA DPA instance, then use the **doca_ctx_set_datapath_on_dpa()** API.

After the context has been started with this mode, it becomes possible to get a DPA handle, using an API defined by the relevant context (e.g., **doca_rdma_get_dpa_handle()**). This handle can then be used to access DPA data path APIs within DPA code.

GPU

Users must first refer to the programming guide of the relevant context (E.g., DOCA Ethernet) to check if datapath on GPU is supported. Additionally, the guide provides what operations can be used.

To set the data path mode to GPU, acquire a DOCA GPU instance, then use the **doca_ctx_set_datapath_on_gpu()** API.

After the context has been started with this mode, it becomes possible to get a GPU handle, using an API defined by the relevant context (e.g., **doca_eth_rxq_get_gpu_handle()**). This handle can then be used to access GPU data path APIs within GPU code.

5.4.1.1.4.5 Object Life Cycle

Most DOCA Core objects share the same handling model in which:

1. The object is allocated by DOCA so it is opaque for the application (e.g., **doca_buf_inventory_create**, **doca_mmap_create**).
2. The application initializes the object and sets the desired properties (e.g., **doca_mmap_set_memrange**).

3. The object is started, and no configuration or attribute change is allowed (e.g., `doca_buf_inventory_start`, `doca_mmap_start`).
4. The object is used.
5. The object is stopped and deleted (e.g., `doca_buf_inventory_stop` → `doca_buf_inventory_destroy`, `doca_mmap_stop` → `doca_mmap_destroy`).

The following procedure describes the mmap export mechanism between two machines (remote machines or host-BlueField):

1. Memory is allocated on Machine1.
2. Mmap is created and is provided memory from step 1.
3. Mmap is exported to the Machine2 pinning the memory.
4. On the Machine2, an imported mmap is created and holds a reference to actual memory residing on Machine1.
5. Imported mmap can be used by Machine2 to allocate buffers.
6. Imported mmap is destroyed.
7. Exported mmap is destroyed.
8. Original memory is destroyed.

5.4.1.1.4.6 RDMA Bridge

The DOCA Core library provides building blocks for applications to use while abstracting many details relying on the RDMA driver. While this takes away complexity, it adds flexibility especially for applications already based on rdma-core. The RDMA bridge allows interoperability between DOCA SDK and rdma-core such that existing applications can convert DOCA-based objects to rdma-core-based objects.

Requirements and Considerations

- This library enables applications already using rdma-core to port their existing application or extend it using DOCA SDK.
- Bridge allows converting DOCA objects to equivalent rdma-core objects.

DOCA Core Objects to RDMA Core Objects Mapping

The RDMA bridge allows translating a DOCA Core object to a matching RDMA Core object. The following table shows how the one object maps to the other.

RDMA Core Object	DOCA Equivalent	RDMA Object to DOCA Object	DOCA Object to RDMA Object
`ibv_pd`	`doca_dev`	`doca_rdma_bridge_open_dev_from_pd`	`doca_rdma_bridge_get_dev_pd`
`ibv_mr`	`doca_buf`	`-`	`doca_rdma_bridge_get_buf_mkey`

5.4.1.1.4.7 DOCA Clock

The DOCA Clock library provides access to timestamps or cycle counters from clocks on both the calling host platform and an attached NVIDIA® BlueField® DPU or NVIDIA® ConnectX® adapter.

The API allows you to retrieve a value from a single clock or to perform a "cross-timestamp" between two different clocks (e.g., host and NIC) to get a correlated set of values.

Key API Functions

- **doca_clock_get_timestamp()** – Retrieves the current value from a single specified clock.
- **doca_clock_get_crosstimestamp()** – Simultaneously captures timestamps from two different clocks, providing a correlation between them.

Supported Clock Types

The library supports the clock types listed in the following table.

Clock Type	Description	Cap Function
NIC Free Running	Internal device timer, typically running from boot time. (seconds/nanoseconds)	**doca_clock_cap_nic_free_running_is_supported()**
NIC Real Time	Device's real-time clock. (seconds/nanoseconds)	**doca_clock_cap_nic_real_time_is_supported()**
NIC DPA Timer	Device DPA clock. (seconds/nanoseconds, microsecond accuracy)	**doca_clock_cap_nic_dpa_timer_is_supported()**
Host Counter Cycles	Host CPU's internal counter. (cycles)	N/A
Host Real Time	Host CPU's real-time clock. (seconds/nanoseconds)	N/A
Host Monotonic	Host CPU's monotonic clock, typically running from boot time. (seconds/nanoseconds)	N/A
Host Monotonic Raw	Host CPU's monotonic clock without frequency adjustments. (seconds/nanoseconds)	N/A

We don't have a way to export this macro.

Considerations

Please note the following requirements and behaviors when using the DOCA Clock library:

- Real-Time Clock Configuration: To access or cross-timestamp the **NIC Real Time** clock, it must be enabled on the device.

```
sudo mlxconfig -d <device_id> set REAL_TIME_CLOCK_ENABLE=1
```

- Firmware Requirement: Firmware version 47.1000 or newer is required to access the **NIC DPA Timer**.
- Permissions: While most operations (querying single clocks, NIC-to-NIC cross-timestamps) can run without root permissions, some Host-to-NIC cross-timestamp operations may require root privileges.

- Precision and Support: The availability of specific cross-timestamp combinations depends on the host's Linux packages, kernel drivers, and NIC capabilities. The library automatically attempts to use the most precise method available for any given clock combination and returns an **accuracy** value (in nanoseconds) along with the timestamps.

5.4.1.1.5 DOCA Core Samples

> All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

5.4.1.1.5.1 Progress Engine Samples

All progress engine (PE) samples use DOCA DMA because of its simplicity. PE samples should be used to understand the PE not DOCA DMA.

pe_common

pe_common.c and **pe_common.h** contain code that is used in most or all PE samples.

Users can find core code (e.g., create MMAP) and common code that uses PE (e.g., **poll_for_completion**).

Struct **pe_sample_state_base** (defined in **pe_common.h**) is the base state for all PE samples, containing common members that are used by most or all PE samples.

pe_polling

The polling sample is the most basic sample for using PE. Start with this sample to learn how to use DOCA PE.

> You can diff between **pe_polling_sample.c** and any other **pe_x_sample.c** to see the unique features that the other sample demonstrates.

The sample demonstrates the following functions:

- How to create a PE
- How to connect a context to the PE
- How to allocate tasks
- How to submit tasks
- How to run the PE
- How to cleanup (e.g., destroy context, destroy PE)

  > Pay attention to the order of destruction (e.g., all contexts must be destroyed before the PE).

The sample performs the following:

1. Uses one DMA context.
2. Allocates and submits 16 DMA tasks.

> Task completion callback checks that the copied content is valid.

3. Polls until all tasks are completed.

pe_async_stop

A context can be stopped while it still processes tasks. This stop is asynchronous because the context must complete/abort all tasks.

The sample demonstrates the following functions:

- How to asynchronously stop a context
- How to implement a context state changed callback (with regards to context moving from stopping to idle)
- How to implement task error callback (check if this is a real error or if the task is flushed)

The sample performs the following:

1. Submits 16 tasks and stops the context after half of the tasks are completed.
2. Polls until all tasks are complete (half are completed successfully, half are flushed).

The difference between **pe_polling_sample.c** and **pe_async_stop_sample.c** is to learn how to use PE APIs for event-driven mode.

pe_event

Event-driven mode reduces CPU utilization (wait for event until a task is complete) but may increase latency or reduce performance.

The sample demonstrates the following functions:

- How to run the PE in event-driven mode

The sample performs the following:

1. Runs 16 DMA tasks.
2. Waits for event.

The difference between **pe_polling_sample.c** and **pe_event_sample.c** is to learn how to use PE APIs for event-driven mode.

pe_multi_context

A PE can host more than one instance of a specific context. This facilitates running a single PE with multiple BlueField devices.

The sample demonstrates the following functions:

- How to run a single PE with multiple instances of a specific context

The sample performs the following:

1. Connects 4 instances of DOCA DMA context to the PE.
2. Allocates and submits 4 tasks to every context instance.
3. Polls until all tasks are complete.

The difference between **pe_polling_sample.c** and **pe_multi_context_sample.c** is to learn how to use PE with multiple instances of a context.

pe_reactive

PE and contexts can be maintained in callbacks (task completion and state changed).

The sample demonstrates the following functions:

- How to maintain the context and PE in the callbacks instead of the program's main function

The user must make sure to:

- Review the task completion callback and the state changed callbacks
- Review the difference between **poll_to_completion** and the polling loop in main

The sample performs the following:

1. Runs 16 DMA tasks.
2. Stops the DMA context in the completion callback after all tasks are complete.

The difference between **pe_polling_sample.c** and **pe_reactive_sample.c** is to learn how to use PE in reactive model.

pe_single_task_cb

A DOCA task can invoke a success or error callback. Both callbacks share the same structure (same input parameters).

DOCA recommends using 2 callbacks:

- Success callback – does not need to check the task status, thereby improving performance
- Error callback – may need to run a different flow than success callback

The sample demonstrates the following functions:

- How to use a single callback instead of two callbacks

The sample performs the following:

1. Runs 16 DMA tasks.
2. Handles completion with a single callback.

The difference between **pe_polling_sample.c** and **pe_single_task_comp_cb_sample.c** is to learn how to use PE with a single completion callback.

pe_task_error

Task execution may fail causing the associated context (e.g., DMA) to move to stopping state due to this fatal error.

The sample demonstrates the following functions:

- How to mitigate a task error during runtime

The user must make sure to:

- Review the state changed callback and the error callback to see how the sample mitigates context error

The sample performs the following:

1. Submits 255 tasks.
2. Allocates the second task with invalid parameters that cause hardware to fail.
3. Mitigates the failure and polls until all submitted tasks are flushed.

The difference between **pe_polling_sample.c** and **pe_task_error_sample.c** is to learn how to mitigate context error.

pe_task_resubmit

A task can be freed or reused after it is completed:

- Task resubmit can improve performance because the program does not free and allocate the task.
- Task resubmit can reduce memory usage (using a smaller task pool).
- Task members (e.g., source or destination buffer) can be set, so resubmission can be used if the source or destination are changed every iteration.

The sample demonstrates the following functions:

- How to re-submit a task in the completion callback
- How to replace buffers in a DMA task (similar to other task types)

The sample performs the following:

1. Allocates a set of 4 tasks and 16 buffer pairs.
2. Uses the tasks to copy all sources to destinations by resubmitting the tasks.

The difference between **pe_polling_sample.c** and **pe_task_resubmit_sample.c** is to learn how to use task resubmission.

pe_task_try_submit

doca_task_submit does not validate task inputs (to increase performance). Developers can use **doca_task_try_submit** to validate the tasks during development.

Task validation impacts performance and should not be used in production.

The sample demonstrates the following functions:

- How to use **doca_task_try_submit** instead of **doca_task_submit**

The sample performs the following:

1. Allocates and tries to submit tasks using **doca_task_try_submit**.

The difference between **pe_polling_sample.c** and **pe_task_try_submit_sample.c** is to learn how to use **doca_task_try_submit**.

5.4.1.1.5.2 Graph Sample

The graph sample demonstrates how to use DOCA graph with PE. The sample can be used to learn how to build and use DOCA graph.

The sample uses two nodes of DOCA DMA and one user node.

The graph runs both DMA nodes (copying a source buffer to two destinations). Once both nodes are complete, the graph runs the user node that compares the buffers.

The sample runs 10 instances of the graph in parallel.

5.4.1.1.5.3 Clock Sample

The **doca_clock_cross_timestamp** sample demonstrates how to create and use a **doca_clock** context associated with a specific DOCA device.

The sample's behavior depends on the arguments provided:

- Single clock – If one clock is specified (using the **-c** flag), the program returns the current timestamp value for that single clock.
- Two clocks – If two clocks are specified (using both **-c** and **-n**), the program performs a cross-timestamp, returning the correlated values for both clocks and the achieved accuracy.

Sample usage:

```
Usage: doca_clock_cross_timestamp [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -p, --pci-addr                    DOCA device PCI device address
  -c, --primary-clock               'nic_fr', 'nic_rt', 'nic_dpa', 'host_cyles', 'host_rt', 'host_mon', or
'host_mon_raw'
  -n, --nic-clock                   'nic_fr', 'nic_rt', or 'nic_dpa'
```

5.4.1.1.5.4 Device Event Notification Sample

The **dev_event_notification** sample demonstrates how to monitor device link state changes using the device event notification API.

The sample opens a DOCA device, registers for link up/down events, connects the notification context to a PE, and waits for events using an epoll-driven PE loop.

5.4.1.1.6 Backward Compatibility of DOCA Core doca_buf

This section lists changes to the DOCA SDK which impacts backward compatibility.

5.4.1.1.6.1 DOCA Core doca_buf

Up to DOCA 2.0.2, the data length of the buffer is ignored when using the buffer as an output parameter, and the new data was written over the data that was there beforehand. From now on, new data is appended after existing data (if any) while updating the data length accordingly.

Because of this change, a destination buffer should be allocated without a data section (data length 0).

> ⚠ Allocating a destination buffer with data length = size will yield a buffer that can't be used as a destination buffer.

In cases where the data length is 0 in a destination buffer, this change would go unnoticed (as appending the data and writing to the data section has the same result).

Reusing buffers requires resetting the data length when wishing to write to the same data address (instead of appending the data), overwriting the existing data. A new function, **`doca_buf_reset_data_len()`**, has been added specifically for this need.

5.4.1.1.7 Sync Event

> ⚠ DOCA Sync Event API is considered thread-unsafe

> ⚠ DOCA Sync Event does not currently support GPU related features.

5.4.1.1.7.1 Introduction

DOCA Sync Event (SE) is a software synchronization mechanism for parallel execution across the CPU, DPU, DPA and remote nodes. The SE holds a 64-bit counter which can be updated, read, and waited upon from any of these units to achieve synchronization between executions on them.

To achieve the best performance, DOCA SE defines a subscriber and publisher locality, where:

- Publisher – the entity which updates (sets or increments) the event value
- Subscriber – the entity which gets and waits upon the SE

  > ℹ Both publisher and subscriber can read (get) the actual counter's value.

Based on hints, DOCA selects memory locality of the SE counter, closer to the subscriber side. Each DOCA SE is configured with a single publisher location and a single subscriber location which can be the CPU or DPU.

The SE control path happens on the CPU (either host CPU or DPU CPU) through the DOCA SE CPU handle. It is possible to retrieve different execution-unit-specific handles (DPU/DPA/GPU/remote handles) by exporting the SE instance through the CPU handle. Each SE handle refers to the DOCA SE instance from which it is retrieved. By using the execution-unit-specific handle, the associated SE instance can be operated from that execution unit.

In a basic scenario, synchronization is achieved by updating the SE from one execution and waiting upon the SE from another execution unit.

5.4.1.1.7.2 Prerequisites

DOCA SE can be used as a context which follows the architecture of a DOCA Core Context, it is recommended to read the following sections of the DOCA Core page before proceeding:

- DOCA Execution Model
- DOCA Device
- DOCA Memory Subsystem

5.4.1.1.7.3 Environment

DOCA SE based applications can run either on the host machine or on the NVIDIA® BlueField® DPU target and can involve DPA, GPU and other remote nodes.

Using DOCA SE with DPU requires BlueField to be configured to work in DPU mode as described in BlueField Modes of Operation.

> Asynchronous wait on a DOCA SE requires NVIDIA® BlueField-3® or newer.

5.4.1.1.7.4 Architecture

DOCA SE can be converted to a DOCA Context as defined by DOCA Core. See DOCA Context for more information.

As a context, DOCA SE leverages DOCA Core architecture to expose asynchronous tasks/events offloaded to hardware.

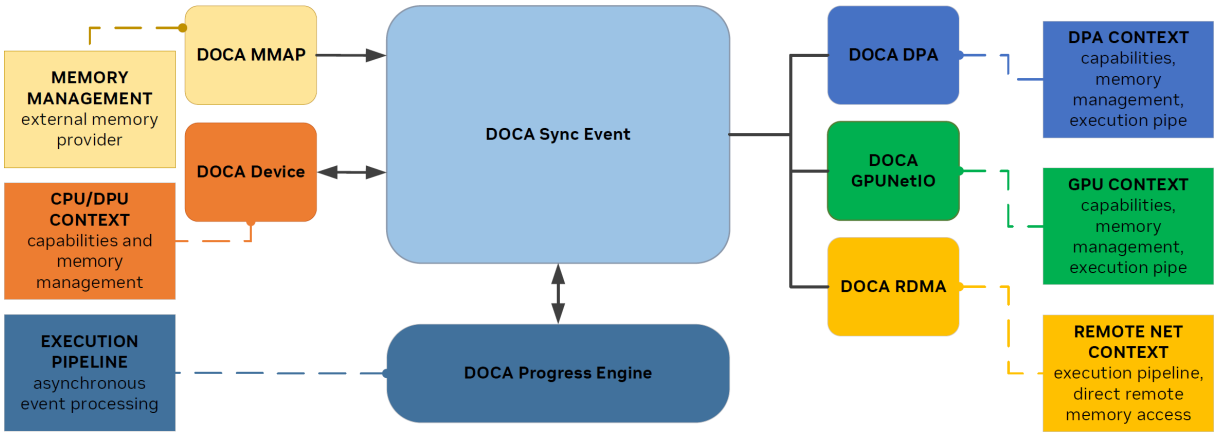
The figure that follows demonstrates components used by DOCA SE. DOCA Device provides information on the capabilities of the configured HW used by SE to control system resources.

DOCA DPA, GPUNetIO, and RDMA modules are required for cross-device synchronization (could be DPA, GPU, or remote peer respectively).

DOCA SE allows flexible memory management by its ability to specify an external buffer, where a DOCA mmap module handles memory registration for advanced synchronization scenarios.

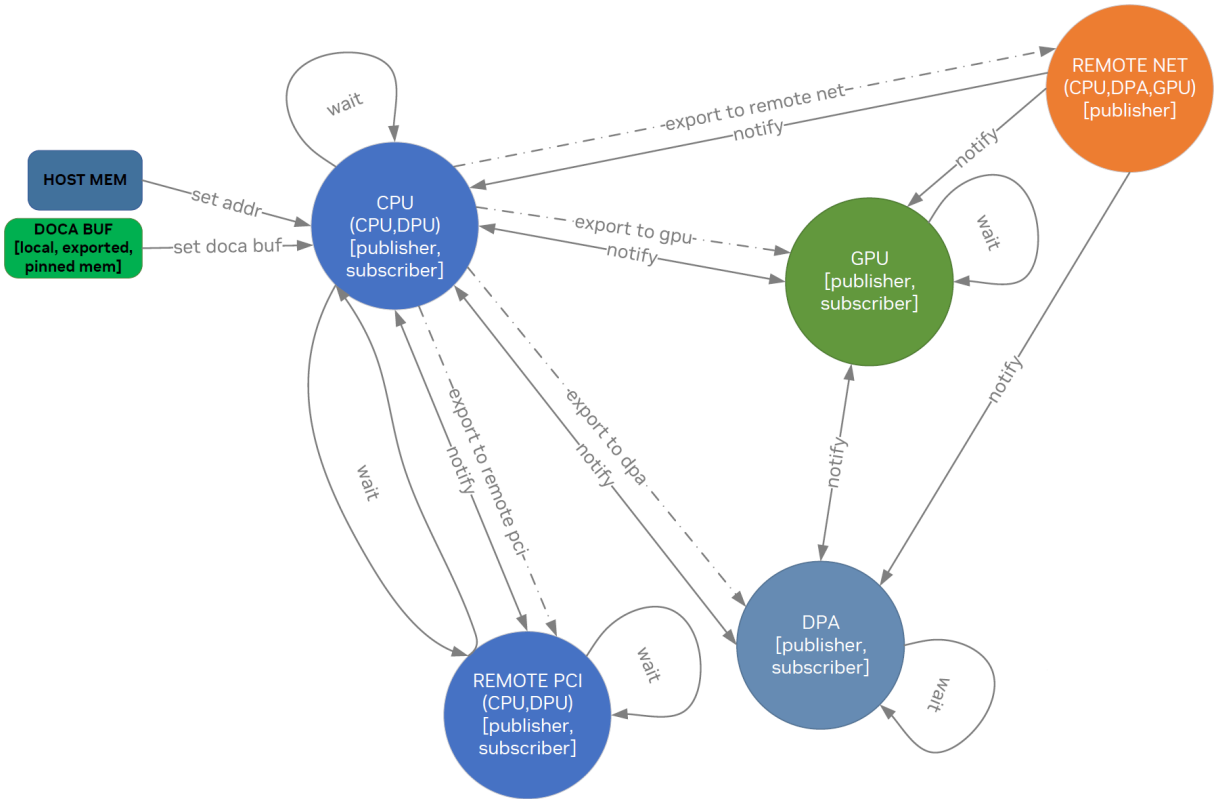
For asynchronous operation scheduling, SE uses the DOCA Progress Engine (PE) module.

DOCA Sync Event Components Diagram



The following diagram represents DOCA SE synchronization abilities on various devices.

DOCA Sync Event Interaction Diagram



DOCA Sync Event Objects

DOCA SE exposes different types of handles per execution unit as detailed in the following table.

Execution Unit	Type	Description
CPU (host/DPU)	`struct doca_sync_event`	Handle for interacting with the SE from the CPU

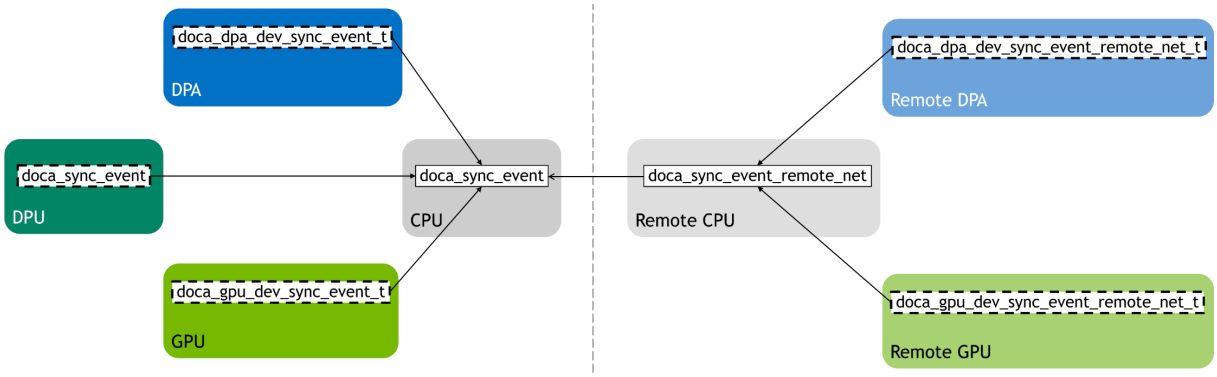
Execution Unit	Type	Description
DPU	`struct doca_sync_event`	Handle for interacting with the SE from the DPU
DPA	`doca_dpa_dev_sync_event_t`	Handle for interacting with the SE from the DPA
GPU	`doca_gpu_dev_sync_event_t`	Handle for interacting with the SE from the GPU
Remote net CPU	`doca_sync_event_remote_net`	Handle for interacting with the SE from a remote CPU
Remote net DPA	`doca_dpa_dev_sync_event_remote_net_t`	Handle for interacting with the SE from a remote DPA
Remote net GPU	`doca_gpu_dev_sync_event_remote_net_t`	Handle for interacting with the SE from a remote GPU

Each one of these handle types has its own dedicated API for creating the handle and interacting with it.

5.4.1.1.7.5 Configuration Phase

Any DOCA SE creation starts with creating CPU handle by calling `doca_sync_event_create` API.

After creation, the SE entity could be shared with local PCIe, remote CPU, DPA, and GPU by a dedicated handle creation via the DOCA SE export flow, as illustrated in the following diagram:



Operation Modes

DOCA SE exposes two different APIs for starting it depending on the desired operation mode, synchronous or asynchronous.

> ⚠ Once started, SE operation mode cannot be changed.

Synchronous Mode

Start the SE to operate in synchronous mode by calling `doca_sync_event_start`.

In synchronous operation mode, each data path operation (get, update, wait) blocks the calling thread from continuing until the operation is done.

> **⚠** An operation is considered done if the requested change fails and the exact error can be reported or if the requested change has taken effect.

Asynchronous Mode

To start the SE to operate in asynchronous mode, convert the SE instance to `doca_ctx` by calling `doca_sync_event_as_ctx`. Then use DOCA CTX API to start the SE and DOCA PE API to submit tasks on the SE (see section "DOCA Progress Engine" for more).

Configurations

Mandatory Configurations

These configurations must be set by the application before attempting to start the SE:

- DOCA SE CPU handle must be configured by providing the runtime hints on the publisher and subscriber locations. Both the subscriber and publisher locations must be configured using the following APIs:
  - `doca_sync_event_add_publisher_location_<cpu|dpa|gpu|remote_pci|remote_net>`
  - `doca_sync_event_add_subscriber_location_<cpu|dpa|gpu|remote_pci>`
- For the asynchronous use case, at least one task/event type must be configured. See configuration of tasks.

Optional Configurations

> **ℹ** If these configurations are not set, a default value is used.

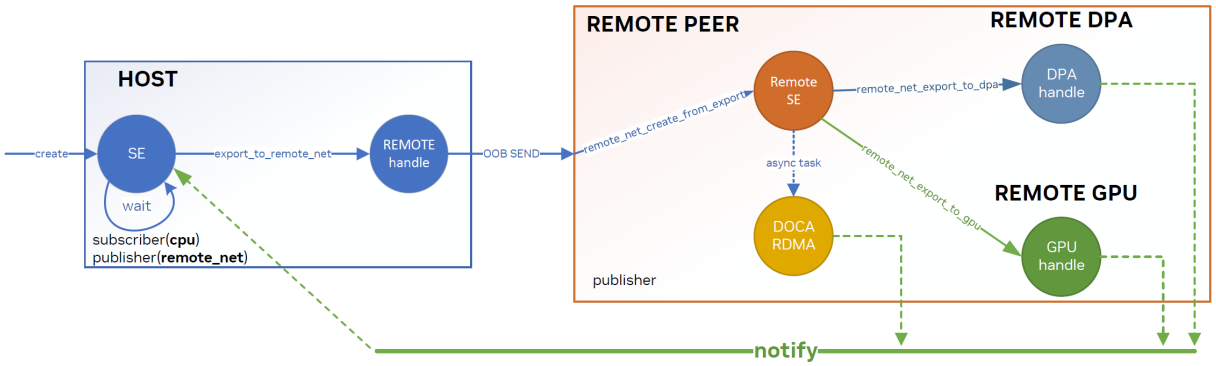
- These configurations provide an 8-byte buffer to be used as the backing memory of the SE. If set, it is user responsibility to handle the memory (i.e., preserve the memory allocated during DOCA SE lifecycle and free it after DOCA SE destruction). If not provided, the SE backing memory is allocated by the SE.
  - `doca_sync_event_set_addr`
  - `doca_sync_event_set_doca_buf`

Export DOCA Sync Event to Another Execution Unit

To use an SE from an execution unit other than the CPU, it must be exported to get a handle for the specific execution unit:

- DPA – `doca_sync_event_get_dpa_handle` returns a DOCA SE DPA handle (`doca_dpa_dev_sync_event_t`) which can be passed to the DPA SE data path APIs from the DPA kernel
- GPU – `doca_sync_event_get_gpu_handle` returns a DOCA SE GPU handle (`doca_gpu_dev_sync_event_t`) which can be passed to the GPU SE data path APIs for the CUDA kernel
- DPU – `doca_sync_event_export_to_remote_pci` returns a blob which can be used from the DPU CPU to instantiate a DOCA SE DPU handle (`struct doca_sync_event`) using the `doca_sync_event_create_from_export` function

DOCA SE allows notifications from remote peers (remote net) utilizing capabilities of the DOCA RDMA library. The following figure illustrates the remote net export flow:



- Remote net CPU – **doca_sync_event_export_to_remote_net** returns a blob which can be used from the remote net CPU to instantiate a DOCA SE remote net CPU handle (**struct doca_sync_event_remote_net**) using the **doca_sync_event_remote_net_create_from_export** function. The handle can be used directly for submitting asynchronous tasks through the **doca_rdma** library or exported to the remote DPA/GPU.
- Remote net DPA – **doca_sync_event_remote_net_get_dpa_handle** returns a DOCA SE remote net DPA handle (**doca_dpa_dev_sync_event_remote_net_t**) which can be passed to the DPA RDMA data path APIs from a DPA kernel
- Remote net GPU – **doca_sync_event_remote_net_get_gpu_handle** returns a DOCA SE remote net GPU handle (**doca_gpu_dev_sync_event_remote_net_t**) which can be passed to the GPU RDMA data path APIs from a CUDA kernel

The CPU handle (**struct doca_sync_event**) can be exported only to the location where the SE is configured.

Prior to calling any export function, users must first verify it is supported by calling the corresponding export capability getter: **doca_sync_event_cap_is_export_to_dpa_supported** , **doca_sync_event_cap_is_export_to_gpu_supported** , **doca_sync_event_cap_is_export_to_remote_pci_supported** , or **doca_sync_event_cap_is_export_to_remote_net_supported** .

Prior to calling any ***_create_from_export** function, users must first verify it is supported by calling the corresponding create from the export capability getter: **doca_sync_event_cap_is_create_from_export_supported** or **doca_sync_event_cap_remote_net_is_create_from_export_supported** .

Once created from an export, both the SE DPU handle **struct doca_sync_event** and the SE remote net CPU handle **struct doca_sync_event_remote_net** cannot be configured, but only the SE DPU handle must be started before it is used.

Data exported in **doca_sync_event_export_to_*** functions contains sensitive information. Make sure to pass this data through a secure channel!

Device Support

DOCA SE needs a device to operate. For instructions on selecting a device, see DOCA Core device discovery.

Both NVIDIA® BlueField®-2 and BlueField®-3 devices are supported as well as any **doca_dev** is supported.

Asynchronous wait (blocking/polling) is supported on NVIDIA® BlueField®-3 and NVIDIA® ConnectX®-7 and later.

As device capabilities may change in the future (see DOCA Capability Check), it is recommended to choose your device using any relevant capability method (starting with the prefix **doca_sync_event_cap_***).

Capability APIs to query whether sync event can be constructed from export blob:

- **doca_sync_event_cap_is_create_from_export_supported**
- **doca_sync_event_cap_remote_net_is_create_from_export_supported**

Capability APIs to query whether sync event can be exported to other execution units:

- **doca_sync_event_cap_is_export_to_remote_pci_supported**
- **doca_sync_event_cap_is_export_to_dpa_supported**
- **doca_sync_event_cap_is_export_to_gpu_supported**
- **doca_sync_event_cap_is_export_to_remote_net_supported**
- **doca_sync_event_cap_remote_net_is_export_to_dpa_supported**
- **doca_sync_event_cap_remote_net_is_export_to_gpu_supported**

Capability APIs to query whether an asynchronous task is supported:

- **doca_sync_event_cap_task_get_is_supported**
- **doca_sync_event_cap_task_notify_set_is_supported**
- **doca_sync_event_cap_task_notify_add_is_supported**
- **doca_sync_event_cap_task_wait_eq_is_supported**
- **doca_sync_event_cap_task_wait_neq_is_supported**

5.4.1.1.7.6 Execution Phase

This section describes execution on CPU. For additional execution environments refer to section "Alternative Datapath Options".

DOCA Sync Event Data Path Operations

The DOCA SE synchronization mechanism is achieved using exposed datapath operations. The API exposes a function for "writing" to the SE and for "reading" the SE.

The synchronous API is a set of functions which can be called directly by the user, while the asynchronous API is exposed by defining a corresponding **doca_task** for each synchronous function to be submitted on a DOCA PE (see DOCA Progress Engine and DOCA Context for additional information).

> Remote net CPU handle (**struct doca_sync_event_remote_net**) can be used for submitting asynchronous tasks using the DOCA RDMA library.

> Prior to asynchronous task submission, users must check if the job is supported using **doca_error_t doca_sync_event_cap_task_<task_type>_is_supported** .

The following subsections describe the DOCA SE datapath operation with respect to synchronous and asynchronous operation modes.

Publishing on DOCA Sync Event

Setting DOCA Sync Event Value

Users can set DOCA SE to a 64-bit value:

- Synchronously by calling **doca_sync_event_update_set**
- Asynchronously by submitting a **doca_sync_event_task_notify_set** task

Adding to DOCA Sync Event Value

Users can atomically increment the value of a DOCA SE:

- Synchronously by calling **doca_sync_event_update_add**
- Asynchronously by submitting a **doca_sync_event_task_notify_add** task

Subscribe on DOCA Sync Event

Getting DOCA Sync Event Value

Users can get the value of a DOCA SE:

- Synchronously by calling **doca_sync_event_get**
- Asynchronously by submitting a **doca_sync_event_task_get** task

Waiting on DOCA Sync Event

Waiting for an event is the main operation for achieving synchronization between different execution units.

Users can wait until an SE reaches a specific value in a variety of ways.

Synchronously

doca_sync_event_wait_gt waits for the value of a DOCA SE to be greater than a specified value in a "polling busy wait" manner (100% processor utilization). This API enables users to wait for an SE in real time.

doca_sync_event_wait_gt_yield waits for the value of a DOCA SE to be greater than a specified value in a "periodically busy wait" manner. After each polling iteration, the calling thread relinquishes the CPU, so a new thread gets to run. This API allows a tradeoff between real-time polling to CPU starvation.

doca_sync_event_wait_eq waits for the value of a DOCA SE to be equal to a specified value in a "polling busy wait" manner (100% processor utilization). This API enables users to wait for an SE in real time.

doca_sync_event_wait_eq_yield waits for the value of a DOCA SE to be equal to a specified value in a "periodically busy wait" manner. After each polling iteration, the calling thread relinquishes the CPU so a new thread gets to run. This API allows a tradeoff between real-time polling to CPU starvation.

doca_sync_event_wait_neq waits for the value of a DOCA SE to not be equal to a specified value in a "polling busy wait" manner (100% processor utilization). This API enables users to wait for an SE in real time.

doca_sync_event_wait_neq_yield waits for the value of a DOCA SE to not be equal to a specified value in a "periodically busy wait" manner. After each polling iteration, the calling thread relinquishes the CPU so a new thread gets to run. This API allows a tradeoff between real-time polling to CPU starvation.

This wait method is supported only from the CPU.

Asynchronously

DOCA SE exposes an asynchronous wait method by defining a **doca_sync_event_task_wait_eq** and **doca_sync_event_task_wait_neq** tasks.

Users can wait for wait-job completion in the following methods:

- Blocking – get a **doca_event_handle_t** from the **doca_pe** to blocking-wait on
- Polling – poll the wait task by calling **doca_pe_progress**

Asynchronous wait (blocking/polling) is supported on BlueField-3 and ConnectX-7 and later.

⚠ Users may leverage the `doca_sync_event_task_get` job to implement asynchronous wait by asynchronously submitting the task on a DOCA PE and comparing the result to some threshold.

Tasks

DOCA SE context exposes asynchronous tasks that leverage the DPU hardware according to the DOCA Core architecture. See DOCA Core Task.

Get Task

The get task retrieves the value of a DOCA SE.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_sync_event_task_get_set_conf`	`doca_sync_event_cap_task_get_is_supported`
Number of tasks	`doca_sync_event_task_get_set_conf`	-

Task Input

Common input described in DOCA Core Task.

Name	Description
Return value	8-bytes memory pointer to hold the DOCA SE value

Task Output

Common output described in DOCA Core Task.

Task Completion Success

After the task is completed successfully, the return value memory holds the DOCA SE value.

Task Completion Failure

If the task fails midway:

- The context may enter a stopping state if a fatal error occurs
- The return value memory may be modified

Task Limitations

All limitations are described in DOCA Core Task.

Notify Set Task

The notify set task allows setting the value of a DOCA SE.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_sync_event_task_notify_set_set_conf`	`doca_sync_event_cap_task_notify_set_is_supported`
Number of tasks	`doca_sync_event_task_notify_set_set_conf`	-

Task Input

Common input described in DOCA Core Task.

Name	Description
Set value	64-bit value to set the DOCA SE value to

Task Output

Common output described in DOCA Core Task.

Task Completion Success

After the task is completed successfully, the DOCA SE value is set to the given set value.

Task Completion Failure

If the task fails midway, the context may enter a stopping state if a fatal error occurs.

Task Limitations

This operation is not atomic. Other limitations are described in DOCA Core Task.

Notify Add Task

The notify add task allows atomically setting the value of a DOCA SE.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_sync_event_task_notify_add_set_conf`	`doca_sync_event_cap_task_notify_add_is_supported`
Number of tasks	`doca_sync_event_task_notify_add_set_conf`	-

Task Input

Common input described in DOCA Core Task.

Name	Description
Increment value	64-bit value to atomically increment the DOCA SE value by
Fetched value	8-bytes memory pointer to hold the DOCA SE value before the increment

Task Output

Common output described in DOCA Core Task.

Task Completion Success

After the task is completed successfully, the following occurs:

- The DOCA SE value is incremented according to the given increment value
- The fetched value memory holds the DOCA SE value before the increment

Task Completion Failure

If the task fails midway:

- The context may enter a stopping state if a fatal error occurs
- The fetched value memory may be modified.

Task Limitations

All limitations are described in DOCA Core Task.

Wait Equal-to Task

The wait-equal task allows atomically waiting for a DOCA SE value to be equal to some threshold.

Task Configuration

Description	API to set the configuration	API to query support
Enable the task	`doca_sync_event_task_wait_eq_set_conf`	`doca_sync_event_cap_task_wait_eq_is_supported`
Number of tasks	`doca_sync_event_task_wait_eq_set_conf`	-

Task Input

Common input described in DOCA Core Task.

Name	Description
Wait threshold	64-bit value to wait for the DOCA SE value to be equal to
Mask	64-bit mask to apply on the DOCA SE value before comparing with the wait threshold

Task Output

Common output described in DOCA Core Task.

Task Completion Success

After the task is completed successfully, the following occurs:

- The DOCA SE value is equal to the given wait threshold.

Task Completion Failure

If the task fails midway, the context may enter a stopping state if a fatal error occurs.

Task Limitations

Other limitations are described in DOCA Core Task.

Wait Not-equal-to Task

The wait-not-equal task allows atomically waiting for a DOCA SE value to not be equal to some threshold.

Task Configuration

Description	API to set the configuration	API to query support
Enable the task	`doca_sync_event_task_wait_neq_set_conf`	`doca_sync_event_cap_task_wait_neq_is_supported`
Number of tasks	`doca_sync_event_task_wait_neq_set_conf`	-

Task Input

Common input described in DOCA Core Task.

Name	Description
Wait threshold	64-bit value to wait for the DOCA SE value to be not equal to
Mask	64-bit mask to apply on the DOCA SE value before comparing with the wait threshold

Task Output

Common output described in DOCA Core Task.

Task Completion Success

After the task is completed successfully, the following occurs:

- The DOCA SE value is not equal to the given wait threshold.

Task Completion Failure

If the task fails midway, the context may enter a stopping state if a fatal error occurs.

Task Limitations

Limitations are described in DOCA Core Task.

Events

DOCA SE context exposes asynchronous events to notify about changes that happen unexpectedly, according to the DOCA Core architecture.

The only event DOCA SE context exposes is common events as described in DOCA Core Event.

5.4.1.1.7.7 State Machine

The DOCA SE context follows the Context state machine as described in DOCA Core Context State Machine.

The following subsection describe how to move to specific states and what is allowed in each state.

Idle

In this state, it is expected that the application will:

- Destroy the context; or
- Start the context

Allowed operations in this state:

- Configure the context according to section "Configurations"
- Start the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop after making sure all tasks have been freed
Stopping	Call progress until all tasks are completed and then freed

Starting

This state cannot be reached.

Running

In this state, it is expected that the application will:

- Allocate and submit tasks
- Call progress to complete tasks and/or receive events

Allowed operations in this state:

- Allocate previously configured task
- Submit an allocated task
- Call stop

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

Stopping

In this state, it is expected that the application will:

- Call progress to complete all inflight tasks (tasks will complete with failure)
- Free any completed tasks

Allowed operations in this state:

- Call progress

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call progress and fatal error occurs
Running	Call stop without freeing all tasks

5.4.1.1.7.8 DOCA Sync Event Tear Down

Multiple SE handles (for different execution units) associated with the same DOCA SE instance can live simultaneously, though the teardown flow is performed only from the CPU on the CPU handle.

> ⚠ Users must validate active handles associated with the CPU handle during the teardown flow because DOCA SE does not do that.

Stopping DOCA Sync Event

To stop a DOCA SE:

- Synchronous – call `doca_sync_event_stop` on the CPU handle
- Asynchronous – stop the DOCA context associated with the DOCA SE instance

> ⚠ Stopping a DOCA SE must be followed by destruction. Refer to section "Destroying DOCA Sync Event" for details.

Destroying DOCA Sync Event

Once stopped, a DOCA SE instance can be destroyed by calling `doca_sync_event_destroy` on the CPU handle.

Remote net CPU handle instance terminates and frees by calling `doca_sync_event_remote_net_destroy` on the remote net CPU handle.

Upon destruction, all the internal resources are released, allocated memory is freed, associated `doca_ctx` (if it exists) is destroyed, and any associated exported handles (other than CPU handles) and their resources are destroyed.

5.4.1.1.7.9 Alternative Datapath Options

DOCA SE supports datapath on CPU (see Execution Phase) and also on DPA and GPU.

GPU Datapath

DOCA SE does not currently support GPU related features.

DPA Datapath

> ℹ An SE with DPA-subscriber configuration currently supports synchronous APIs only.

Once a DOCA SE DPA handle (`doca_dpa_dev_sync_event_t`) has been retrieved it can be used within a DOCA DPA kernel as described in DOCA DPA Sync Event.

5.4.1.1.7.10 DOCA Sync Event Sample

This section provides DOCA SE sample implementation on top of the BlueField DPU.

The sample demonstrates how to share an SE between the host and the DPU while simultaneously interacting with the event from both the host and DPU sides using different handles.

Running DOCA Sync Event Sample

1. Refer to the following documents:
   - DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - DOCA Troubleshooting for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

```
cd /opt/mellanox/doca/samples/doca_common/sync_event_<local|remote>_pci
meson /tmp/build
ninja -C /tmp/build
```

> The binary **doca_sync_event_<local|remote>_pci** is created under **/tmp/build/**.

3. Sample usage:

```
Usage: doca_sync_event_remote_pci [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse all command flags from an input json file

Program Flags:
  -d, --pci-addr                    Device PCI address
  -r, --rep-pci-addr                DPU representor PCI address
  --async                           Start DOCA Sync Event in asynchronous mode (synchronous mode by
default)
  --async_num_tasks                 Async num tasks for asynchronous mode
  --atomic                          Update DOCA Sync Event using Add operation (Set operation by default)
```

> The flag **--rep-pci-addr** is relevant only for the DPU.

4. For additional information per sample, use the **-h** option:

```
/tmp/build/doca_sync_event_<local|remote>_pci -h
```

Samples

Sync Event Remote PCIe

> Whatever this sample or Sync Event Local PCIe should be run on the DPU before another sample runs on the host.

> These samples are also available on GitHub.

This sample demonstrates creating an SE from an export which is associated with an SE on a local PCIe (host or the DPU) and interacting with the SE to achieve synchronization between the host and DPU.

The sample logic includes:

1. Reading configuration files and saving their content into local buffers.
2. Locating and opening DOCA devices and DOCA representors (if running on the DPU) matching the given PCIe addresses.
3. Initializing DOCA Comm Channel.
4. Receiving SE blob through Comm Channel.
5. Creating SE from export.
6. Starting the above SE in the requested operation mode (synchronous or asynchronous).
7. Interacting with the SE:
    a. Waiting for signal from the host – synchronously or asynchronously (with busy wait polling) according to user input.
    b. Signaling the SE for the host – synchronously or asynchronously, using set or atomic add, according to user input.
8. Cleaning all resources.

Reference:

- **/opt/mellanox/doca/samples/doca_common/sync_event_remote_pci/sync_event_remote_pci_sample.c**
- **/opt/mellanox/doca/samples/doca_common/sync_event_remote_pci/sync_event_remote_pci_main.c**
- **/opt/mellanox/doca/samples/doca_common/sync_event_remote_pci/meson.build**

Sync Event Local PCIe

Whatever this sample or Sync Event Remote PCIe should be run on the DPU before another sample runs on the host.

These samples are also available on GitHub.

This sample demonstrates how to initialize a SE to be shared with a remote PCIe (host or the DPU) how to export it to a remote PCIe, and how to interact with the SE to achieve synchronization between the host and DPU.

The sample logic includes:

1. Reading configuration files and saving their content into local buffers.
2. Locating and opening DOCA devices and DOCA representors (if running on the DPU) matching the given PCIe addresses.
3. Creating and configuring the SE to be shared with a remote PCIe.
4. Starting the above SE in the requested operation mode (synchronous or asynchronous).
5. Initializing DOCA Comm Channel.

6. Exporting the SE and sending it through the Comm Channel.
7. Interacting with the SE :
    a. Signaling the SE for the remote PCIe – synchronously or asynchronously, using set or atomic add, according to user input.
    b. Waiting for a signal – synchronously or asynchronously, with busy wait polling, according to user input.
8. Cleaning all resources.

Reference:

- `/opt/mellanox/doca/samples/doca_common/sync_event_local_pci/sync_event_local_pci_sample.c`
- `/opt/mellanox/doca/samples/doca_common/sync_event_local_pci/sync_event_local_pci_main.c`
- `/opt/mellanox/doca/samples/doca_common/sync_event_local_pci/meson.build`

5.4.1.1.8 Mmap Advise

5.4.1.1.8.1 Introduction

DOCA Mmap Advise is used to give advanced memory-related instructions to NVIDIA® BlueField® DPUs in order to improve system or application performance.

> ⚠ To use DOCA Mmap Advise with BlueField, the device must be configured to work in DPU mode as described in BlueField Modes of Operation.

The operations in the instructions are meant to influence the performance of the application, but not its semantics. The operations allow an application to inform the NIC how it expects it to use some mapped memory areas, so the BlueField's hardware can choose appropriate optimization techniques.

5.4.1.1.8.2 Prerequisites

DOCA Mmap Advise is a context and follows the architecture of a DOCA Core Context, it is recommended to read the following sections of the DOCA Core page before proceeding:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

5.4.1.1.8.3 Architecture

DOCA Mmap Advise is a DOCA Context as defined by DOCA Core. See DOCA Core Context for more information.

DOCA Mmap Advise currently supports the following list of advised operations:

- Cache Invalidate Operation

Cache Invalidate Operation

When data is processed by BlueField's cores it may be temporarily stored in the cores' system-level cache (i.e., L3 cache). When a cache line is occupied and new data must be written to it, the cache management sub-system evicts the existing data, usually based on LRU policy, by performing a write-back operation to store this data in the main (DDR) memory. When this data is not required to be stored in the BlueField's memory (e.g., it is host data and is no longer needed after it is copied to the host's memory), the cache's write-back operation wastes memory bandwidth that reduces overall system performance, which is undesirable. The simplest way to avoid this write-back operation is to mark the appropriate cache lines as "invalid". This enables their immediate reuse, without additional operations.

The cache invalidate operation facilitates invalidating a set of cache lines.

5.4.1.1.8.4 Environment

Applications based on DOCA Mmap Advise can run on the BlueField target.

Objects

Device and Device Representor

The MMAP Advise context requires a DOCA Device to operate. The device is used to access memory and perform the copy operation. See DOCA Core Device Discovery.

> For the same DPU, it does not matter which device is used (i.e., PF, VF, SF) as all these devices utilize the same hardware components.

> The device must stay valid for as long as the MMAP Advise instance is not destroyed.

Memory Buffers

The cache invalidate task requires one DOCA Buffer containing the address space to invalidate depending on the allocation pattern of the buffers (refer to the table in section "Inventory Types"). To find what kind of memory is supported, refer to the table in section "Buffer Support".

Buffers must not be modified or read during the cache invalidate operation.

5.4.1.1.8.5 Configuration Phase

To start using the context, users must go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context, to allow execution of tasks and retrieval of events.

Configurations

The context can be configured to match the application's use case.

To find if a configuration is supported, or what the min/max value for it is, refer to section "Device Support".

Mandatory Configurations

These configurations are mandatory and must be set by the application before attempting to start the context:

- At least one task/event type must be configured. See configuration of tasks and/or events in sections "Tasks" and "Events" respectively for information.
- A device with appropriate support must be provided upon creation

Device Support

DOCA Mmap Advise requires a device to operate. To pick a device, refer to DOCA Core Device Discovery.

As device capabilities may change (see DOCA Core Device Support), it is recommended to select your device using the following method:

- `doca_mmap_advise_cap_task_cache_invalidate_is_supported`

Some devices expose different capabilities as follows:

- Maximum cache invalidate buffer size may differ.

Buffer Support

Tasks support buffers with the following features:

Buffer Type	Buffer
Local mmap buffer	Yes
MMAP from PCIe export buffer	No
MMAP from RDMA export buffer	No
Linked list buffer	No

5.4.1.1.8.6 Execution Phase

This section describes execution on the CPU using DOCA Core Progress Engine.

Tasks

DOCA Mmap Advise exposes asynchronous tasks that leverage DPU hardware according to the DOCA Core architecture. See DOCA Core Task for information.

Cache Invalidate Task

The cache invalidate task facilitates invalidating a set of cache lines, preventing them from being written back to the RAM (thus increasing performance).

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_mmap_advise_task_invalidate_cache_set_conf`	`doca_mmap_advise_cap_task_cache_invalidate_is_supported`

Description	API to Set the Configuration	API to Query Support
Number of tasks	`doca_mmap_advise_task_invalidate_cache_set_conf`	–
Maximal buffer size	–	`doca_mmap_advise_task_cache_invalidate_get_max_buf_size`
Maximal buffer list size	–	–

Task Input

Common input as described in DOCA Core Task.

Name	Description
buffer	Buffer that points to the memory to be invalidated

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task is completed successfully:

- The cache is invalidated

Task Completion Failure

If the task fails midway:

- The context may enter stopping state, if a fatal error occurs
- The cache is not invalidated

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the buffer should not be read/written to
- Other limitations are described in DOCA Core Task

Events

DOCA Mmap Advise exposes asynchronous events to notify on changes that happen unexpectedly, according to DOCA Core architecture.

The only events DOCA Mmap Advise exposes are common events as described in DOCA Core Event.

5.4.1.1.8.7 State Machine

DOCA Mmap Advise context follows the context state machine as described in DOCA Core Context State Machine.

The following section describes how to move states and what is allowed in each state.

Idle

In this state it is expected that the application:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to section "Configurations"
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop after making sure all tasks have been freed
Stopping	Call progress until all tasks are completed and freed

Starting

This state cannot be reached.

Running

In this state, it is expected that the application:

- Allocates and submits tasks
- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocating a previously configured task
- Submitting a task
- Calling stop

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

Stopping

In this state it is expected that the application:

- Calls progress to complete all in-flight tasks (tasks complete with failure)
- Frees any completed tasks

Allowed operations:

- Call progress

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call progress and fatal error occurs

Previous State	Transition Action
Running	Call stop without freeing all tasks

5.4.1.1.8.8 Alternative Datapath Options

DOCA Mmap Advise only supports datapath on the CPU. See section "Execution Phase".

5.4.1.1.8.9 Samples

Cache Invalidate Sample

The sample illustrates how to invalidate the cache for a memory range after copying it using DOCA DMA.

The sample logic includes:

1. Locating DOCA device.
2. Initializing needed DOCA core structures.
3. Populating DOCA memory map with two relevant buffers.
4. Allocating element in DOCA buffer inventory for each buffer.
5. Initializing DOCA DMA memory copy task object.
6. Initializing DOCA Mmap Advise cache invalidate task object
7. Submitting DMA task.
8. Polling for completion:
    a. Handling DMA task completion – submitting the cache invalidate task in the DMA task completion callback body.
    b. Handling cache invalidate task completion.
9. Polling for completion.
10. Destroying DMA, DOCA MMAP Advise, and DOCA Core objects.

Reference:

- **/opt/mellanox/doca/samples/doca_common/cache_invalidate/cache_invalidate_sample.c**
- **/opt/mellanox/doca/samples/doca_common/cache_invalidate/cache_invalidate_main.c**
- **/opt/mellanox/doca/samples/doca_common/cache_invalidate/meson.build**

5.4.1.2 DOCA Log

The DOCA logging infrastructure provides a mechanism for printing error messages, debug logs, and application-specific messages within the DOCA SDK.

To use DOCA logging, applications must include the **doca_log.h** header file in their source code.

5.4.1.2.1 Log Verbosity Level Enumerations

DOCA logging supports multiple verbosity levels, allowing applications to filter logs based on severity:

```
enum doca_log_level {
    DOCA_LOG_LEVEL_DISABLE = 10,    /**< Disable log messages */
```

```
    DOCA_LOG_LEVEL_CRIT = 20,       /**< Critical log level */
    DOCA_LOG_LEVEL_ERROR = 30,      /**< Error log level */
    DOCA_LOG_LEVEL_WARNING = 40,    /**< Warning log level */
    DOCA_LOG_LEVEL_INFO = 50,       /**< Info log level */
    DOCA_LOG_LEVEL_DEBUG = 60,      /**< Debug log level */
    DOCA_LOG_LEVEL_TRACE = 70,      /**< Trace log level */
};
```

The **DOCA_LOG_LEVEL_TRACE** verbosity level is available only if the macro **DOCA_LOGGING_ALLOW_TRACE** is defined before compilation.

Refer to **doca_log.h** for more information.

5.4.1.2.2 Logging Backends

A logging backend is the destination where log messages are directed. DOCA supports multiple backend types:

- FILE* – Any open file, including **stdout** or **stderr**
- File descriptor – Any valid file descriptor (e.g., raw files, sockets, pipes)
- Buffer (**buf**) – A memory buffer that stores log messages and triggers a callback for each message
- Syslog – Standard system logging mechanism

Each logging backend is initialized with the following default verbosity levels:

- Lower level – **DOCA_LOG_LEVEL_INFO**
- Upper level – **DOCA_LOG_LEVEL_CRIT**

DOCA SDK logs and application logs have different configurations and can be controlled independently through the DOCA logging API.

All messages are sent to all backends that meet the configured verbosity level.

5.4.1.2.3 Enabling DOCA SDK Libraries Logging

DOCA SDK libraries automatically print debug and error messages to the configured logging backends.

The maximum number of logging backends can be queried using **doca_log_get_max_num_backends_sdk()**.

To create a logging backend for SDK logs, use:

```
doca_log_backend_create_with_file_sdk()
doca_log_backend_create_with_fd_sdk()
doca_log_backend_create_with_buf_sdk()
doca_log_backend_create_with_syslog_sdk()
```

When creating a new SDK logging backend, the first message printed is the DOCA version, which is printed in verbosity level INFO.

The following functions control SDK logging levels:

- **doca_log_level_set_global_sdk_limit()** sets global verbosity level

  > This function applies to all SDK backends.

  > A newly created SDK backend's verbosity level is set to the SDK global verbosity level value by default.

- **doca_log_backend_set_sdk_level()** sets verbosity level for a specific SDK backend
- **doca_log_level_get_global_sdk_limit()** gets the global SDK verbosity level

> Log messages may change across DOCA versions. Do not rely on specific message formatting or content permanence.

5.4.1.2.4 Enabling DOCA Application Logging

Applications that use the DOCA SDK can utilize DOCA logging for structured logging

The maximum number of logging backends can be queried using **doca_log_get_max_num_backends()**.

The total number of application logging backends that can be created can be queried using the following function:

```
doca_log_get_max_num_backends()
```

To create an application logging backend, use:

```
doca_log_backend_create_with_file()
doca_log_backend_create_with_fd()
doca_log_backend_create_with_buf()
doca_log_backend_create_with_syslog()
```

The following functions configure logging limits:

- **doca_log_backend_set_level_{lower|upper}_limit()** sets lower and/or upper verbosity levels for a backend

  > The lower and upper verbosity levels of a newly created backend are set to the default values.

- **doca_log_backend_create_standard()** creates a standard, non-configurable logging setup. It creates two backends:
  - **stdout** – Logs messages up to **DOCA_LOG_LEVEL_INFO**
  - **stderr** – Logs messages from **DOCA_LOG_LEVEL_WARNING** to **DOCA_LOG_LEVEL_CRIT**
- **doca_log_backend_set_level_{lower|upper}_limit_strict()** sets strict limits for a backend (prevent future changes)

- **doca_log_level_set_global_{lower|upper}_limit()** sets global logging limits (affects all non-strict backends)

5.4.1.2.5 Logging DOCA Application Messages

To enable DOCA logging in an application, use the DOCA log registration macro at the beginning of each source file:

```
DOCA_LOG_REGISTER(source);
```

This registers a logger for that file and automatically manages teardown.

> The length of the source provided to **DOCA_LOG_REGISTER()** cannot be greater than **DOCA_LOGGER_NAME_SIZE**.

Log messages are printed using the following macros (similar to **printf()** syntax):

```
DOCA_LOG_CRIT(format, ...);   // Critical error messages
DOCA_LOG_ERR(format, ...);    // Standard error messages
DOCA_LOG_WARN(format, ...);   // Warnings
DOCA_LOG_INFO(format, ...);   // Informational messages
DOCA_LOG_DBG(format, ...);    // Debug messages
DOCA_LOG_TRC(format, ...);    // Trace-level messages
```

> Messages are printed to all application logging backends that fall within the configured verbosity range.

5.4.1.2.6 Controlling Verbosity Level of Loggers

Every logger, whether created by an SDK library or an application, operates independently. You can control the output of each logger by adjusting its individual verbosity level.

Use the following functions to create or destroy a list of active loggers:

- **doca_logger_create_list()** – Generates a list of all current loggers.
- **doca_logger_destroy_list()** – Frees the list memory.

> Configuration changes (such as verbosity levels) are persistent and remain in effect even after the list is destroyed.

Once you have the list, you can manage individual loggers using:

- **doca_logger_set_level()** – Sets the verbosity level for a specific logger.
- **doca_logger_get_level()** – Retrieves the current verbosity level.
- **doca_logger_get_name()** – Retrieves the name of a logger.

To locate a specific logger within the list:

- **doca_logger_find_by_name()** – Returns a pointer to the logger matching the specified name.

5.4.1.2.7 DOCA Log Aggregation

Log aggregation offers a way to optimize performance in multi-threaded environments by reducing contention on log backends (for example, multiple threads attempting to write concurrently to the same file). Log aggregation enables each worker thread to write log messages to an in-memory buffer. A dedicated background thread then periodically flushes these messages to the configured log backends.

This approach significantly improves worker thread efficiency, as application threads can continue processing without being delayed by logging operations or backend contention. All existing DOCA Log functionality is retained. No change to logging code is required beyond starting the aggregation thread and the shutdown considerations below.

Log Aggregation uses fixed size buffers to obtain maximum performance, therefore messages may be dropped when under heavy load. The log Aggregation API enables configuration of the backend to modify the impact of this based on the applications needs.

> ⚠ To guarantee all log messages are written, log aggregation should not be used; however, this will negatively impact performance.

- Only messages with a severity lower than **max_unaggregated_level** may be dropped
- Messages with a severity greater than or equal to **max_unaggregated_level** are guaranteed to be written.

To enable log aggregation, the background aggregation thread must be created before specific log backends. When enabled, all messages with severity greater than or equal max unaggreagated are guaranteed to be written, while those bellow may be dropped.

> ⚠ If the background thread is not started, DOCA Log behaves exactly as it did previously.

> ⚠ Log aggregation is only supported on Linux.

5.4.1.2.7.1 Log Aggregation API

doca_log_set_max_unaggreated_level(uint32_t level) - Sets the maximum aggregated level. Messages logged at or above this severity bypass aggregation and are immediately written to the backend, along with any buffered messages. The logging thread will block until this flush is complete. Default: **DOCA_LOG_LEVEL_WARNING**

doca_log_set_bg_sampling period(uint32 usec) - Sets the maximum sleep interval for the background aggregation thread. If buffered messages are present, the background thread continues processing without delay, only when no buffered messages are present will the thread sleep. A value of 0 may be used to process all messages without delay. Default: 100,000 microseconds

doca_log_set_max_aggregated_size(uint32_t size) - Sets the maximum size of a single batch of aggregated messages written to the backends. Default: 128KiB

doca_log_process_msg(void *unused) - Entry point for the background aggregation thread, this function should be passed to **pthread_create()**.

> ⚠ Only one aggregation thread may be running per application.

> ⚠ This function does not return under normal operation and runs until application shutdown.

5.4.1.2.7.2 Shutdown considerations

Because the aggregation thread runs until application termination, it may still be writing logs when the application exits, potentially leading to undefined behavior. Special considerations should be taken to shut down correctly:

1. Force a flush of the aggregation buffers by logging a message with a severity greater than or equal to **max_unaggregated_level**
2. This operation will block until all buffered messages are written.
3. After the flush is completed, no further messages should be logged.
4. It is now safe to clean up logging resources and terminate the application.

5.4.1.2.7.3 Example usage

```
void *log_fn(void* unused)
{
    DOCA_LOG_INFO("Log from secondary thread");
    return NULL;
}

int main()
{
    pthread_t aggregator_thread;
    pthread_create(&aggregator_thread, NULL, doca_log_process_msg, NULL);
    pthread_detatch(aggregator_thread);

    struct doca_log_backend *backend;
    FILE *log_file = fopen("/path/to/file.log", "w");
    doca_log_backend_create_with_file_sdk(log_file, &backend);

    pthread_t log_thread;
    ptherad_create(&log_thread, NULL, log_fn, NULL);

    DOCA_LOG_INFO("Log from primary thread");

    /* Force the aggregation buffers to be flushed */
    DOCA_LOG_WARN("Flush buffers");

    /* Clean up and exit */
    pthread_join(log_thread, NULL);
    fclose(log_file);

    return 0;
}
```

5.4.1.2.8 DOCA Log Samples

> ℹ All the DOCA LOG samples described in this section are governed under the BSD-3 software license agreement.

5.4.1.2.8.1 Log_limits_client Sample

The **doca_log_limits_client** sample demonstrates how to send a single UDP command (get-limits, set-lower-limits, or set-upper-limits) to a **doca_log_limits_server** to either read the current DOCA_LOG level limits or set the lower/upper DOCA_LOG level limit value.
This sample does not accept multiple commands, only the first valid command is sent to the log_limits_server.

Sample usage:

```
Usage: doca_log_limits_client [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -p, --port                        Server UDP port (default: 9999), should be a port number > 1024 to avoid "sudo"
 requirement
  -i, --ip                          Server IPv4 address (default: 127.0.0.1), only IPv4 is supported. A valid IPv4
address is expected to be 4 dot-separated numbers.
  -g, --get-limits                  Get both lower and upper log limits from the server
  -sl, --set-lower-limit            Set the lower log limit on the server, valid values must be >=10 and <=70 and a
multiple of 10
  -su, --set-upper-limit            Set the upper log limit on the server, valid values must be >=10 and <=70 and a
multiple of 10
```

```
# Read current global DOCA_LOG lower and upper level limits from the log_limits_server
./doca_log_limits_client [-i <server_ip>] [-p <port>] -g

# Set the log_limits_server's global DOCA_LOG lower level limit to DOCA_LOG_LEVEL_DEBUG
./doca_log_limits_client [-i <server_ip>] [-p <port>] -sl 60

# Set  the log_limits_server's global DOCA_LOG upper level limit to DOCA_LOG_LEVEL_CRIT
./doca_log_limits_client [-i <server_ip>] [-p <port>] -su 20
```

5.4.1.2.8.2 Log Limits Server Sample

The **doca_log_limits_server** sample demonstrates how to dynamically adjust global log level limits using the DOCA_LOG API. It operates as a UDP server that continuously generates DOCA LOG messages across all severity levels (**CRIT** , **ERROR** , **WARN** , **INFO** , **DEBUG** , and **TRACE**).

Client-Server Interaction

The server provisions a UDP socket to accept incoming connections from a **doca_log_limits_client** . It parses and executes the following commands:

Client Command	Executed DOCA APIs	Server Behavior & Response
get-limits	**doca_log_level_get_global_lower_limit** **doca_log_level_get_global_upper_limit**	Returns the current lower and upper limit values directly to the client.
set-lower-limit	**doca_log_level_set_global_lower_limit**	No response is sent to the client. The dynamic level change is immediately visible in the server's continuous log output.

Client Command	Executed DOCA APIs	Server Behavior & Response
`set-upper-limit`	`doca_log_level_set_global_upper_limit`	No response is sent to the client. The dynamic level change is immediately visible in the server's continuous log output.

> ⚠ The server continues running and outputting log messages to demonstrate the impact of the limit changes until it is terminated via `Ctrl+C`.

This sample runs as a UDP server that continuously prints DOCA LOG messages at various levels (CRIT, ERROR, WARN, INFO, DEBUG, TRACE):

Execution and Usage

Start the server:

```
$ ./doca_log_limits_server [-p <port>]
```

Command line flags:

```
Usage: doca_log_limits_server [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file
```

Program flags:

Short Flag	Long Flag	Default	Description
`-p`	`--port`	`9999`	The server UDP port. It is highly recommended to use a port number `> 1024` to avoid requiring `sudo` (root) privileges.

5.4.2 DOCA Flow

This guide describes how to deploy the DOCA Flow library, the philosophy of the DOCA Flow API, and how to use it. The guide is intended for developers writing network function applications that focus on packet processing (such as gateways). It assumes familiarity with the network stack and DPDK.

5.4.2.1 Introduction

> ⚠ The quality status of DOCA libraries is listed here.

DOCA Flow is the most fundamental API for building generic packet processing pipes in hardware. The DOCA Flow library provides an API for building a set of pipes, where each pipe consists of match criteria, monitoring, and a set of actions. Pipes can be chained so that after a pipe-defined action is executed, the packet may proceed to another pipe.

Using DOCA Flow API, it is easy to develop hardware-accelerated applications that have a match on up to two layers of packets (tunneled).

- MAC/VLAN/ETHERTYPE
- IPv4/IPv6
- TCP/UDP/ICMP
- GRE/VXLAN/GTP-U/ESP/PSP
- Metadata
- SRv6 (**segments_left** field)

The execution pipe can include packet modification actions such as the following:

- Modify MAC address
- Modify IP address
- Modify L4 (ports)
- Strip tunnel
- Add tunnel
- Set metadata
- SRv6 actions
    - Insert SRv6 Segment Routing Header (SRH)
    - Decrement SRv6 segments left
    - Copy active SID to IPv6 destination based on segments left field

The execution pipe can also have monitoring actions such as the following:

- Count
- Policers

The pipe also has a forwarding target which can be any of the following:

- Software (RSS to subset of queues)
- Port
- Another pipe
- Drop packets

5.4.2.2 Prerequisites

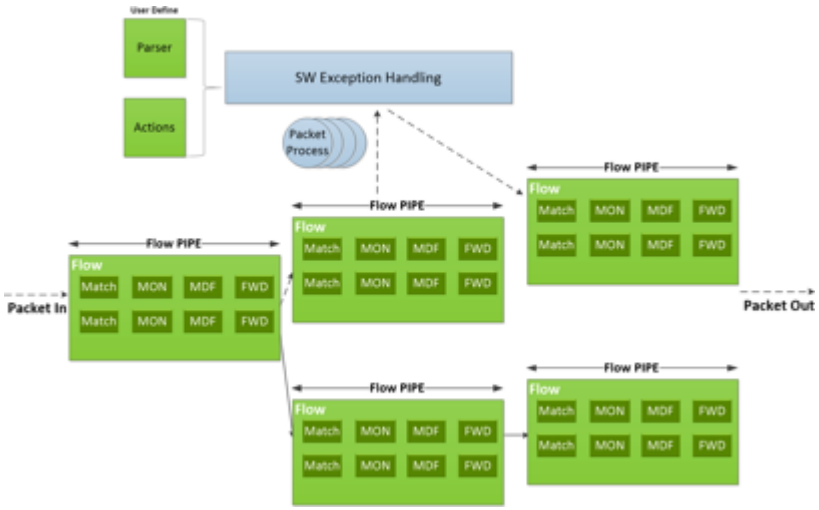
A DOCA Flow-based application can run on either the host machine or an NVIDIA® BlueField® DPU target.

These applications require the allocation of huge pages, which must be configured prior to execution. Use the following commands to allocate and mount 2MB huge pages:

```
$ echo '1024' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
$ sudo mkdir /mnt/huge
$ sudo mount -t hugetlbfs -o pagesize=2M nodev /mnt/huge
```

5.4.2.3 Architecture

The following diagram shows how the DOCA Flow library defines a pipe template, receives a packet for processing, creates the pipe entry, and offloads the flow rule in NIC hardware.



Features of DOCA Flow:

- User-defined set of matches parser and actions
- DOCA Flow pipes can be created or destroyed dynamically
- Packet processing is fully accelerated by hardware with a specific entry in a flow pipe
- Packets that do not match any of the pipe entries in hardware can be sent to Arm cores for exception handling and then reinjected back to hardware

The DOCA Flow pipe consists of the following components:

- Monitor (MON in the diagram) - counts, meters
- Modify (MDF in the diagram) - modifies a field
- Forward (FWD in the diagram) - forwards to the next stage in packet processing

5.4.2.4 Steering Domains

DOCA Flow organizes pipes into high-level containers named domains to address the specific needs of the underlying architecture.

A key element in defining a domain is the packet direction and a set of allowed actions.

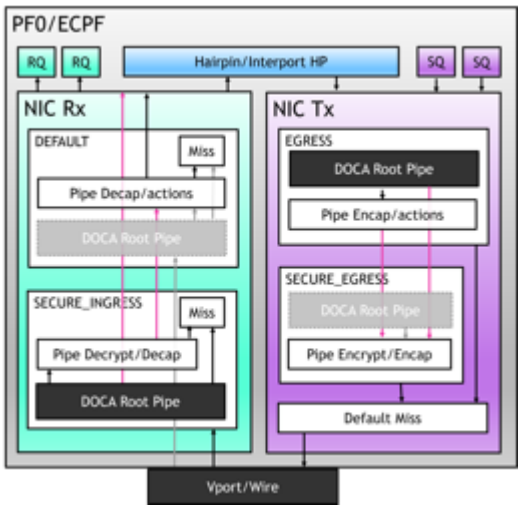
- A domain is a pipe attribute (also relates to shared objects)
- A domain restricts the set of allowed actions
- Transition between domains is well-defined (packets cannot cross domains arbitrarily)
- A domain may restrict the sharing of objects between packet directions
- Packet direction can restrict the move between domains

5.4.2.4.1 List of Steering Domains

DOCA Flow provides the following set of predefined steering domains:

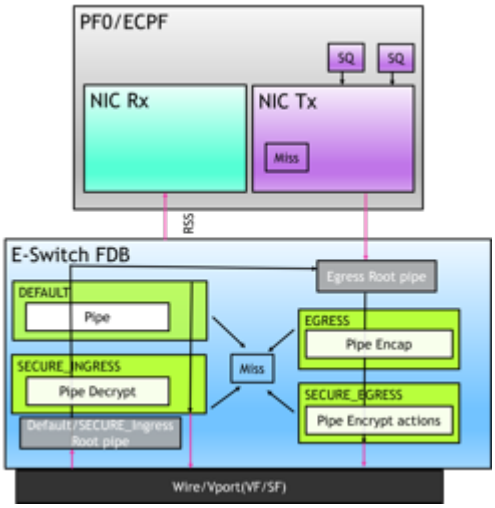
Domain	Description
DOCA_FLOW_PIPE_DOMAIN_DEFAULT	• Default domain for actions on ingress traffic • Secure actions are not allowed here • The next milestone is queue or pipe in the EGRESS domain • Miss action is: Port Default Miss (kernel by default, see Port Default Miss)
DOCA_FLOW_PIPE_DOMAIN_SECURE_INGRESS	• For secure actions on ingress traffic • Encapsulation and encrypting actions not allowed here • The only allowed domain for decrypting secure actions • The next milestone is queue or pipe in the DEFAULT or EGRESS domain • Only meta register is preserved • Miss action is: Port Default Miss (kernel by default, see Port Default Miss)
DOCA_FLOW_PIPE_DOMAIN_EGRESS	• Domain for actions on egress traffic • Secure actions are not allowed here • The next milestone is wire/representor or pipe in SECURE_EGRESS domain • Miss action is: Port Default Miss (kernel by default, see Port Default Miss). In non-switch mode, send to wire/representor.
DOCA_FLOW_PIPE_DOMAIN_SECURE_EGRESS	• Domain for secure actions on egress traffic • Decapsulation actions are not allowed here • The only allowed domain for encrypting secure action • The next milestone is wire/representor • Miss action is: Port Default Miss (kernel by default, see Port Default Miss). In non-switch mode, send to wire/representor.

5.4.2.4.2 Domains in VNF Mode



5.4.2.4.3 Domains in Switch Mode

In switch mode, forwarding from a pipe with the default domain to the egress domain root pipe is permitted, while forwarding from the egress domain back to the default domain is prohibited.



Traffic originating from Software Transmit (SW Tx) automatically forwards to the egress root pipe.

A pipe with RSS forwarding strictly follows these cross-domain rules.

NVIDIA recommends performing encapsulation from the egress domain.

5.4.2.4.3.1 Domain Details

Domain	Direction	Traffic Entry	Allowed Actions	Prohibited Actions	Root Pipe Slot
`DEFAULT`	Ingress	Packet arrives at E-Switch from wire or vport.	match, modify, monitor, forward	crypto (encrypt/ decrypt)	Ingress slot (shared with `SECURE_INGRESS`)
`SECURE_INGRESS`	Ingress	Forwarded from `DEFAULT`.	match, decrypt, monitor	encap, encrypt	Ingress slot (shared with `DEFAULT`)
`EGRESS`	Egress	1. SW Tx auto-enters egress root pipe. 2. Forwarded from ingress domains.	match, modify, encap, monitor	crypto (encrypt/ decrypt)	Egress slot (shared with `SECURE_EGRESS`)
`SECURE_EGRESS`	Egress	Forwarded from `DEFAULT` or `EGRESS`, or as egress root pipe receiving SW Tx.	match, encrypt, monitor	decap	Egress slot (shared with `EGRESS`)

Only one root pipe is allowed per direction slot. `DEFAULT` and `SECURE_INGRESS` are mutually exclusive as ingress root; `EGRESS` and `SECURE_EGRESS` are mutually exclusive as egress root.

5.4.2.4.3.2 Cross-Domain Forwarding Rules

↓ Source \ Target →	DEFAULT	SECURE_INGRESS	EGRESS	SECURE_EGRESS
DEFAULT	Yes	Yes	Yes (root only)	Yes (root only)
SECURE_INGRESS	Yes	Yes	Yes (root only)	No
EGRESS	No	No	Yes	Yes
SECURE_EGRESS	No	No	Yes	Yes

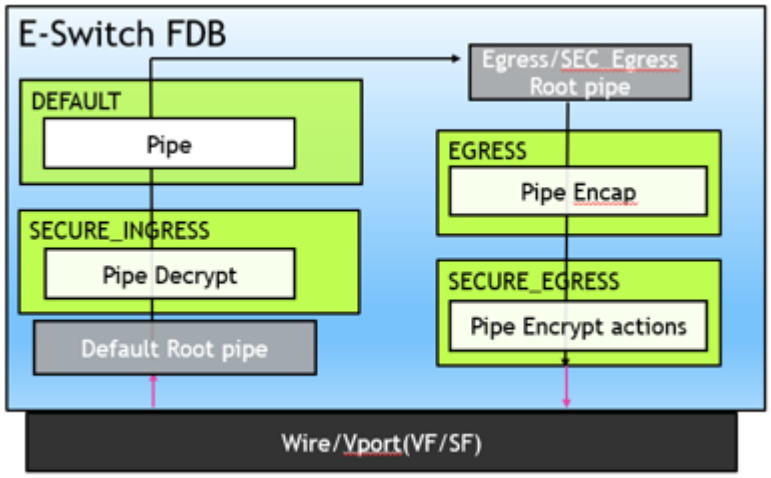
- (root only): When forwarding from an ingress domain to an egress domain, the target must be a root pipe.
- Forwarding from egress domains back to ingress domains is not allowed.
- Cross-direction forwarding (ingress → egress) requires **hairpinq_num** to be configured.

5.4.2.4.3.3 Traffic Models & Domain Mapping

The following models illustrate common traffic paths utilized in switch mode. Square brackets **[...]** indicate optional stages (e.g., cryptographic operations) that can be appended based on specific application requirements.

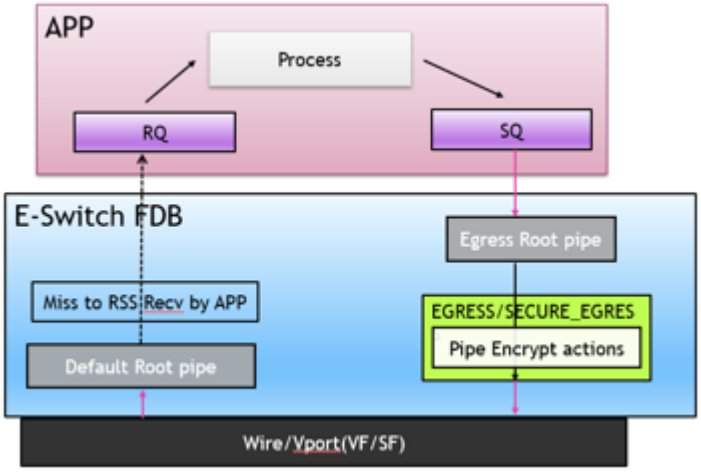
Fast Path (Full Hardware Forwarding, No Software Involvement)

#	Path	Domain Chain	Description
FP1	VF → Wire	**DEFAULT** → **[EGR/SEC_EGR (root)]** → **fwd_port** → wire	Egress direction: The **DEFAULT** root matches and forwards directly to the wire via **fwd_port**. It can optionally pass through **SECURE_EGRESS** for encryption.
FP2	Wire → VF	**DEFAULT** → **[SEC_ING -> EGRESS(root)]** → **fwd_port** → VF	Ingress direction: The **DEFAULT** root matches and forwards directly to the VF via **fwd_port**. It can optionally pass through **SECURE_INGRESS** for decryption before utilizing **EGRESS** for dispatch.



Slow Path (Miss to Software, Software Processes, Re-enters e-Switch)

#	Path	Domain Chain	Description
SP1	* → RSS → SW → Tx → Wire	`DEFAULT` → `RSS` → `SW` → `EGRESS(root)` → `[SEC_EGR]` → wire	Egress slow path: Unmatched packets miss to RSS for Software (SW) processing. The SW then transmits the packet (with `pkt_meta`), and the `EGRESS` root dispatches it through the encrypt pipeline or directly to the wire.
SP2	* → RSS → SW → Tx → VF	`DEFAULT` → `RSS` → `SW` → `EGRESS(root)` → `fwd_port` → VF	Ingress slow path: Unmatched packets miss to RSS for SW processing. The SW then transmits the packet (with `pkt_meta`), and the `EGRESS` root dispatches it via `fwd_port` to the VF.



Architectural Design Patterns

- The `DEFAULT` root serves as the primary traffic entry point.
- The `EGRESS` root acts as the unified re-entry point for software-transmitted (SW Tx) traffic.
- The Fast Path completely bypasses RSS/Software, relying entirely on full hardware processing.
- The second half of the Slow Path (post-SW Tx) directly reuses the Fast Path's `EGRESS` → output chain.
- `SECURE_INGRESS` and `SECURE_EGRESS` are optional cryptographic processing layers added strictly on demand.

Reference Applications & Samples

Use Case	Paths	Reference
Simple L2/L3 switch	FP1 + FP2	`flow_switch`, `flow_switch_single`
Bidirectional wire ↔ VF	FP1 + FP2	`flow_switch_to_wire`
IPsec/PSP Security Gateway (full offload)	FP1[SEC_EGR] + FP2[SEC_ING → EGRESS]	`ipsec_security_gw`
IPsec/PSP Security Gateway (partial offload)	SP1[SEC_EGR] + SP2	`ipsec_security_gw`, `psp_gateway`

PSP GW on-demand tunnel setup	SP1	**psp_gateway**

Important Routing Constraints

- Mutually exclusive root pipes: **DEFAULT** and **SECURE_INGRESS** share a single ingress root slot. **EGRESS** and **SECURE_EGRESS** share a single egress root slot.
- Cross-domain forwarding: Forwarding traffic from an ingress domain to an egress domain requires the target to be an egress root pipe.
- Miss cascade: If a pipe lacks a defined **fwd_miss** handler, unmatched traffic automatically falls through to the port default miss (which defaults to the kernel).
- Software Tx auto-entry: Packets transmitted via **rte_eth_tx_burst ()** automatically enter the egress root pipe.
- One-way transitions: Egress domains are prohibited from forwarding back to ingress domains. Similarly, **SECURE_INGRESS** cannot forward directly to **SECURE_EGRESS**.

5.4.2.5 API

DOCA API is available through the DOCA API Reference.

> The pkg-config (***.pc** file) for the DOCA Flow library is **doca-flow** .

5.4.2.6 Flow Life Cycle

5.4.2.6.1 Initialization Flow

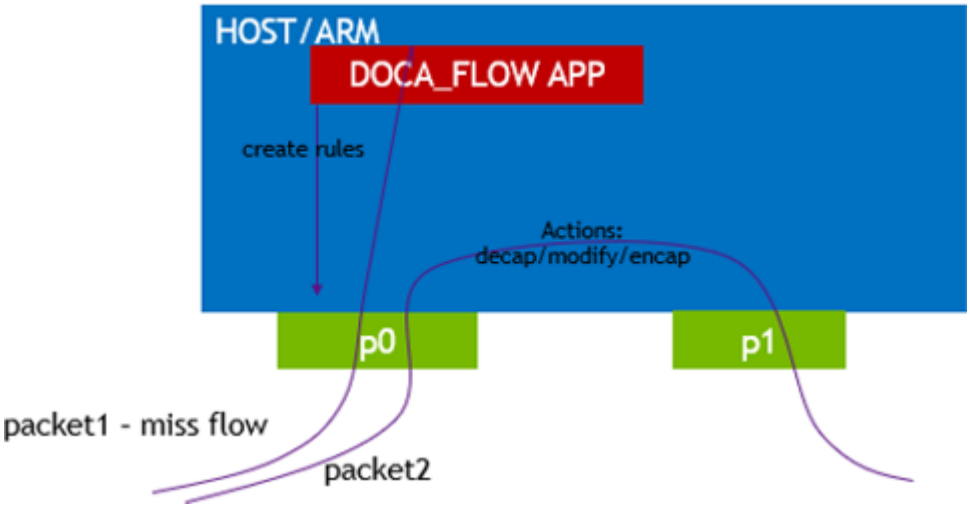
Before using any DOCA Flow function, it is mandatory to call **doca_flow_init()** . This function initializes all resources required by the DOCA Flow library.

5.4.2.6.1.1 Pipe Mode

The pipe mode (set via **mode_args**) defines the fundamental traffic topology and default "miss" behavior for a DOCA port. When a port is initialized, this mode creates the initial miss rules.

This mode (**mode_args**) defines the basic traffic in DOCA. It creates some miss rules when a DOCA port initializes. Currently, DOCA supports 2 modes:

- **vnf**
  A packet arriving from one of the device's ports is processed, and can be sent to another port. By default, missed packets go to RSS.
  The following diagram shows the basic traffic flow in **vnf** mode. Packet1 firstly misses and is forwarded to host RSS. The app captures this packet and decides how to process it and then creates a pipe entry. Packet2 will hit this pipe entry and do the action, for example, for VXLAN, will do decap, modify, and encap, then is sent out from P1.



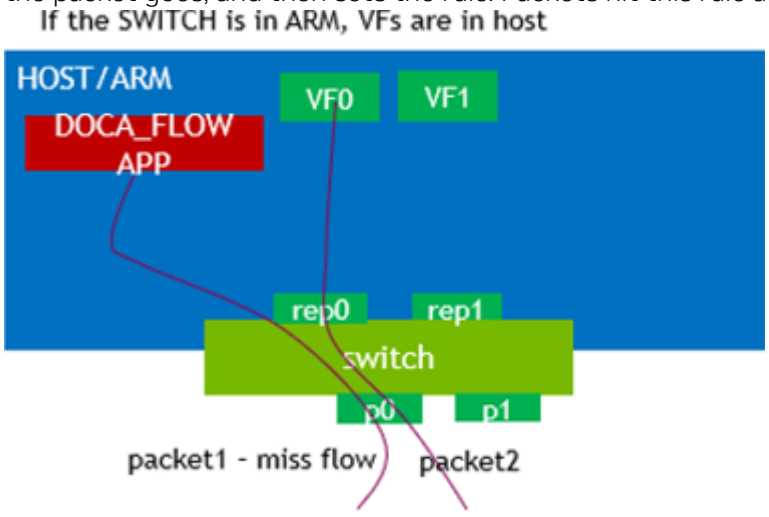
To forward packets between ports, use `doca_flow_port_pair` to establish a port pair. Port pairing is unidirectional, meaning that pairing port X with port Y allows forwarding from X to Y, but not the reverse. To enable forwarding from Y to X, you must pair Y with X using `doca_flow_port_pair` again. After pairing ports, packets can be forwarded to either the paired port directly or the paired port's egress root pipe, which sends packets through the egress routine.

> ⚠ Forwarding from an ingress pipe to an egress pipe is only allowed to the `egress_root` pipe.

- `switch`

  Used for internal switching, only representor ports are allowed, for example, uplink representors and SF/VF representors. Packet is forwarded from one port to another. If a packet arrives from an uplink and does not hit the rules defined by the user's pipe, then the packet is received on all RSS queues of the representor of the uplink.

  The following diagram shows the basic flow of traffic in `switch` mode. Packet1 firstly misses to host RSS queues. The app captures this packet and decides to which representor the packet goes, and then sets the rule. Packets hit this rule and go to representor0.



`doca_dev` field is mandatory in `doca_flow_port_cfg` (using `doca_flow_port_cfg_set_dev()`).

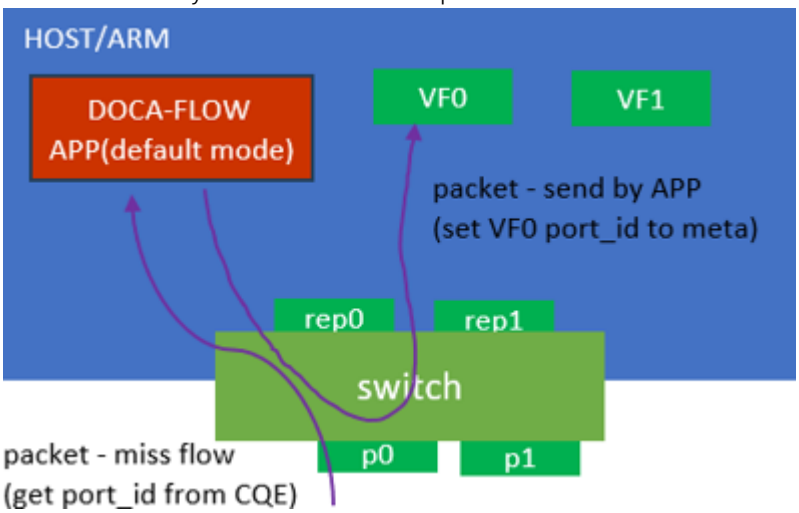
> **⚠** The application must avoid initialization of the VF/SF representor ports in DPDK API (i.e., the following functions **rte_eth_dev_configure()**, **rte_eth_rx_queue_setup()**, **rte_eth_dev_start()** must not be called for VF/SF representor ports).

DOCA Flow switch mode unifies all the ports to the switch manager port for traffic management. This means that all the traffic is handled by switch manager port. Users only must create an RSS pipe on the switch manager port to get the missed traffic, and they should only manage the pipes on the switch manager port.
To obtain the switch manager port, call **doca_flow_port_switch_get()**:

```
/* Pass any started port to get its eswitch's manager port */
struct doca_flow_port *sw_port = doca_flow_port_switch_get(ports[0]);

/* Or pass NULL when there is only one eswitch */
struct doca_flow_port *sw_port = doca_flow_port_switch_get(NULL);
```

All pipe creation, entry addition, and **doca_flow_entries_process()** calls must use this switch manager port, not the individual ports returned by **doca_flow_port_start()**.
Switch mode can work with two different **mode_args** configurations: With or without **expert**. The way to retrieve the miss traffic source's **port_id** depends on this configuration:

- If **expert** is not set, the traffic misses to software would be tagged with **port_id** information in the mbuf CQE field to allow users to deduce the source **port_id**. Meanwhile, users can set the destination **port_id** to mbuf meta and the packet is sent out directly to the destination port based on the meta information.



- If **expert** is set, the **port_id** is not added to the packet. Users can configure the pipes freely to implement their own solution.

  > **⚠** Traffic cloned from the VF to the RSS pipe misses its **port_id** information due to firmware limitation.

5.4.2.6.1.2 Port Capabilities

DOCA Flow provides a port capabilities API (`doca_flow_port_capabilities_create()` , `doca_flow_port_capabilities_destroy()`) to query hardware-supported features before pipeline configuration. The whole cycle of querying port capabilities (create -> query -> destroy) should happen after `doca_flow_init()` and before port start. The device passed to create must remain open until destroy.

Currently supported queries include CT IPv6 steering. For hardware requirements, API usage, and an example, see Querying CT IPv6 Support in the DOCA Flow Connection Tracking guide.

5.4.2.6.2 Start Point

DOCA Flow API serves as an abstraction layer API for network acceleration. The packet processing in-network function is described from ingress to egress and, therefore, a pipe must be attached to the origin port. Once a packet arrives to the ingress port, it starts the hardware execution as defined by the DOCA API.

5.4.2.6.2.1 Port identifiers

`doca_flow_port` is an opaque object, as the DOCA Flow API is designed to be independent of any specific packet I/O framework (e.g., DPDK). To begin using a DOCA Flow port, the user must call `doca_flow_port_start()` . This function associates the application's underlying port with a DOCA Flow port.

> ⚠ DOCA Flow performs all steering in hardware, operating on the port's `doca_dev` independently of the packet I/O framework. When the port is managed through DPDK, the application must probe the device with the `dv_flow_en=2` device argument, which puts the DPDK PMD into hardware steering mode (see the "Running the Sample" section). Paths that do not use DPDK do not require this argument.

During this setup, the user must assign a logical port ID using the `doca_flow_port_cfg_set_port_id()` API. This `port_id` is later referenced in:

- `doca_flow_parser_meta.port_id` - when matching packets received on a specific port. In switch mode, the special value `UINT16_MAX` (0xffff) can be used to match software-originated traffic — i.e., packets sent by the application via `rte_eth_tx_burst()` . This is particularly useful in EGRESS root pipes to distinguish SW-injected packets from hardware-forwarded traffic.
- `doca_flow_fwd.port_id` - when forwarding packets to a specific port

For DPDK-based applications, the DPDK `port_id` can be used directly as the logical port identifier.

5.4.2.6.2.2 DOCA Device

A DOCA device must be specified during `doca_flow_port_start()` to provide the necessary hardware context for the port.

- For physical, virtual, or subfunction devices (PF, VF, or SF), use the `doca_flow_port_cfg_set_dev()` API to associate a `doca_dev` object with the port.
- For representor ports, use the `doca_flow_port_cfg_set_dev_rep()` API to associate a `doca_dev_rep` object instead.

Providing a `doca_dev_rep` is what explicitly indicates that the port is a representor.

5.4.2.6.2.3 Actions Memory

The use of DOCA Flow actions mandates that sufficient memory will be allocated for them, in advance, per port. This is done using the `doca_flow_port_cfg_set_actions_mem_size()` API function. The recommended way of getting precise information about the expected actions memory consumption is the following:

1. During development, allocate a fixed high amount of actions memory, regardless of your actual pipeline.
2. Once the steering pipeline is ready, examine the program's actual consumption using the DOCA Flow Tune tool.
3. Using DOCA Flow Tune for calibration, update the parameter passed to the setter function.

Please note that in the hot upgrade use case, the total actions memory size set for all instances should not exceed the maximum resource limit of this hardware port.

5.4.2.6.2.4 Port devargs

Set specific configurations per port type using `doca_flow_port_cfg_set_devargs()`.

- `no_wire_to_wire` – Acts as a hint telling DOCA Flow that no wire-to-wire traffic path exists on the port. Enable this when the environment exclusively processes Wire-to-VF or VF-to-Wire traffic. This reduces latency for these specific forwarding paths and significantly improves port startup times.

5.4.2.6.3 Port Operation State

DOCA Flow ports can be initialized multiple times from different instances. Each instance prepares its pipeline, but only one actively receives port traffic at a time. The instance actively handling the port traffic depends on the operation state set by the `doca_flow_port_cfg_set_operation_state()` function:

- `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE` – The instance actively handles incoming and outgoing traffic
- `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE_READY_TO_SWAP` – The instance handles traffic actively when no other active instance is available
- `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY` – The instance handles traffic only when no active or `active_ready_to_swap` instance is available
- `DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED` – The instance does not handle traffic, regardless of the state of other instances

If the `doca_flow_port_cfg_set_operation_state()` function is not called, the default state `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE` is applied.

> ⚠ When a port is configured with a state that expects to handle traffic, it takes effect only after root pipes are created for this port.

When the active port is closed, either gracefully or due to a crash, the standby instance automatically becomes active without any action required.

The port operation state can be modified after the port is started using the `doca_flow_port_operation_state_modify()` function.

5.4.2.6.3.1 Use Case Examples

Hot Upgrade

This operation state mechanism allows upgrading the DOCA Flow program without losing any traffic.

To upgrade an existing DOCA Flow program with ports started in `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE` state (Instance A):

1. Open a new Instance B and start its ports in `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY` state.
2. Modify Instance A's ports from `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE` to `DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED` state. At this point, Instance B starts receiving traffic.
3. Close Instance A.
4. Open a new Instance C with `DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED` state. Instance C is the upgraded version of Instance A.
5. Create the entire pipeline for Instance C.
6. Change Instance C's state from `DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED` to `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE`. At this point, Instance B stops receiving traffic and Instance C starts.
7. Instance B can either be closed or kept as a backup should Instance C crash.

Swap Existing Instances

This mechanism also facilitates swapping two different DOCA Flow programs without losing any traffic.

To swap between two existing DOCA Flow programs with ports started in `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE` and `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY` states (Instance A and Instance B, respectively):

1. Modify Instance A's ports from `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE` to `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE_READY_TO_SWAP`.
2. Modify Instance B's ports from `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY` to `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE`. At this point, Instance B starts receiving traffic.

3. Modify Instance A's ports from `DOCA_FLOW_PORT_OPERATION_STATE_ACTIVE_READY_TO_SWAP` to `DOCA_FLOW_PORT_OPERATION_STATE_STANDBY`.

5.4.2.6.3.2 Limitations

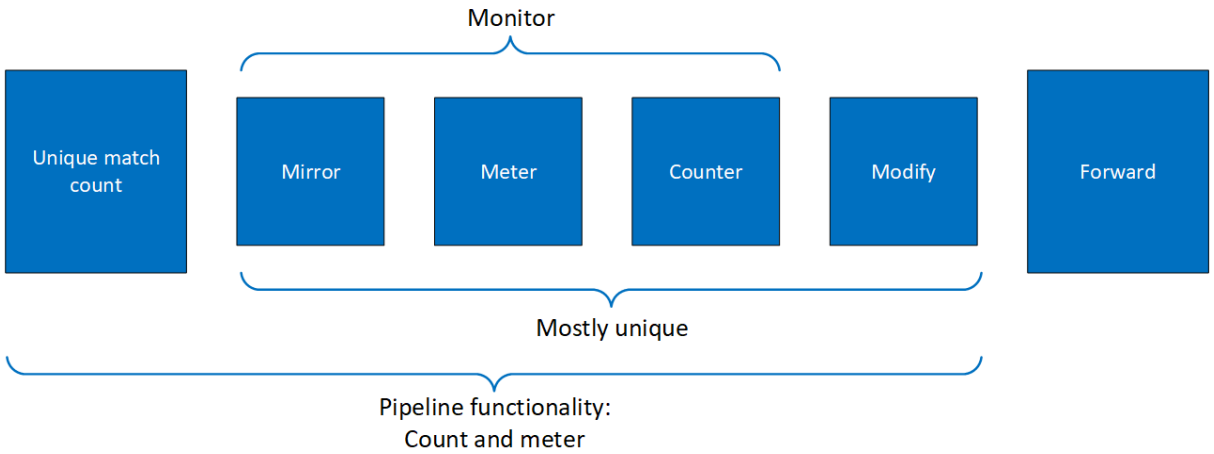
- Supported only in switch mode – the `mode_args` string must include `"switch"`.
- Only the switch port supports states; its representors are affected by its state. Starting a representor port or calling the modify function with a non-active operation state should fail.
- Two instances cannot be in the same operation state simultaneously, except for `DOCA_FLOW_PORT_OPERATION_STATE_UNCONNECTED`. If a new instance initializes a port while another instance with the same operation state exists, a `DOCA_ERROR_ALREADY_EXIST` error is returned either during port startup or root pipe creation.
- Due to an underlying driver limitation, each ASO object type (e.g., `METER`, `COUNTER`) is restricted to a maximum of 64 million (64M) objects per port.

5.4.2.6.4 Create Pipe and Pipe Entry

Pipe is a template that defines packet processing without adding any specific hardware rule. A pipe consists of a template that includes the following elements:

- Match
- Monitor
- Actions
- Forward

The following diagram illustrates a pipe structure.



The creation phase allows the hardware to efficiently build the execution pipe. After the pipe is created, specific entries can be added. A subset of the pipe may be used (e.g., skipping the monitor completely, just using the counter, etc).

5.4.2.6.4.1 Pipe Matching or Action Applying

DOCA Flow allows defining criteria for matching on a packet or for taking actions on a matched packet by modifying it. The information defining these criteria is provided through the following pointers:

- Match or action pointer – given at pipe or entry creation
- Mask pointer – optionally given at pipe creation

Defining criteria for matching or actions on a packet can be done at the pipe level, where it applies to all packets of a pipe, or specified on a per entry basis, where each entry defines the operation on either the match, actions, or both.

In DOCA Flow terminology, when a field is identified as CHANGEABLE at pipe creation, this means that the actual criterion of the field is deferred to entry creation. Different entries can provide different criteria for a CHANGEABLE field.

A match or action field can be categorized, during pipe creation, as one of the following:

- IGNORED – Ignored in either the match or action taking process
- CHANGEABLE – When the actual behavior is deferred to the entry creation stage
- SPECIFIC – Value is used as is in either match or action process

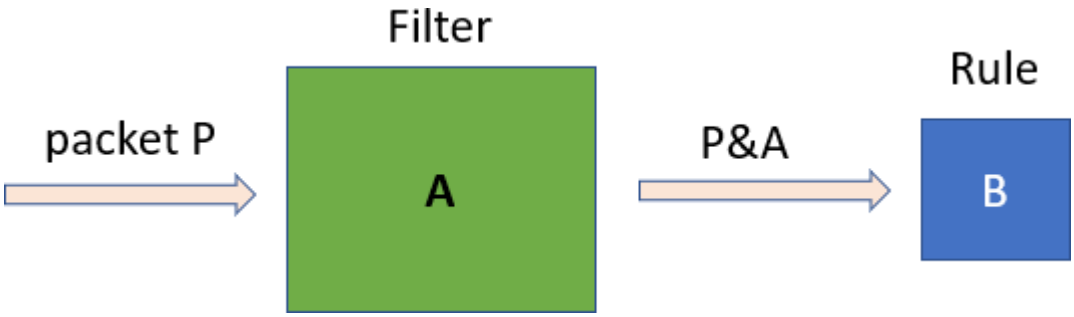
A mask field can either be provided, in which case it is called explicit matching, or action applying. If the mask pointer is NULL, we call it implicit matching or action applying. The following subsections provide the logic governing matching and action applying.

When a field value is specified as `0xffff` it means that all the field's bits are set (e.g., for TTL it means `0xff` and for IPv4 address it means `0xffffffff`).

> ⚠ Creating multiple entries with the same match criteria is not supported. Instead, use the update entry API to modify the existing entry.

Matching

Matching is the process of selecting packets based on their fields' values and steering them for further processing. Processing can either be further matching or actions applying.



The packet enters the green filter which modifies it by masking it with the value A. The output value, P&A, is then compared to the value B, and if they are equal, then that is a match.

The values of A and B are evaluated according to the values of the pipe configuration and entry configuration fields, according to the tables in sections "Implicit Matching" and "Explicit Matching".

Implicit Matching

Match Type	Pipe Match Value (V)	Pipe Match Mask (M)	Entry Match Value (E)	Filter (A)	Rule (B)
Ignore	0	NULL	N/A	0	0
Constant	0<V<0xffff	NULL	N/A	0xffff	V
Changeable (per entry)	0xffff	NULL	0≤E≤0xffff	0xffff	E

Explicit Matching

Match Type	Pipe Match Value (V)	Pipe Match Mask (M)	Entry Match Value (E)	Filter (A)	Rule (B)
Constant	V!=0xffff	0<M≤0xffff	0≤E≤0xffff	M	M&V
Changeable	V==0xffff	0<M≤0xffff	0≤E≤0xffff	M	M&E
Ignored	0≤V<0xffff	M==0	0≤E≤0xffff	0	0

Action Applying

Implicit Action Applying

Action Type	Pipe Action Value (V)	Pipe Action Mask (M)	Entry Action Value (E)	Action on the Field
Ignore	0	NULL	N/A	None
Constant	0 < V < 0xffff	NULL	N/A	Set to V
Changeable	0xffff	NULL	E	Set to E

Implicit action applying example:

- Destination IPv4 address is 255.255.255.255
- No mask provided
- Entry value is 192.168.0.1
- Result – The action field is changeable. Therefore, the value is provided by the entry. If a match on the packet occurs, the packet destination IPv4 address is changed to 192.168.0.1.

Meta only supports changeable action.

Explicit Action Applying

Assume P is packet's field value.

Action Type	Pipe Action Value (V)	Pipe Action Mask (M)	Entry Action Value (E)	Action on the Field
constant	V!=0xffff	0≤M≤0xffff	0≤E≤0xffff	set to (~M & P) | (M & V) In words: modify only bits that are set on the mask to the values in V
Changeable	V==0xffff	0<M≤0xffff	0≤E≤0xffff	set to (~M & P) | (M & E)
Ignored	0≤V<0xffff	M==0	0≤E≤0xffff	none

Explicit action applying example:

- Destination IPv4 address is 192.168.10.1
- Mask is provided and equals 255.255.0.0
- Entry value is ignored
- Result – If a match on the packet occurs, the packet destination IPv4 value changes to 192.168.0.0

5.4.2.6.4.2 Setting Pipe Match or Action

Match is a mandatory parameter when creating a pipe. Using the **doca_flow_match** struct, users must define the packet fields to be matched by the pipe.

For each **doca_flow_match** field, users select whether the field type is:

- Ignore (match any) – the value of the field is ignored in a packet. In other words, match on any value of the field.
- Constant (specific) – all entries in the pipe have the same value for this field. Users should not put a value for each entry.
- Changeable – the value of the field is defined per entry. Users must provide it upon adding an entry.

> ⚠ L4 type, L3 type, and tunnel type cannot be changeable.

> ⚠ **gtp_next_ext_hdr_type** supports only **psc** type (0x85).

The match field type can be defined either implicitly or explicitly using the **doca_flow_pipe_cfg_set_match(struct doca_flow_pipe_cfg *cfg, const doca_flow_match *match, const doca_flow_match *match_mask)** function. If **match_mask == NULL**, then it is done implicitly. Otherwise, it is explicit.

In the tables in the following subsections, an example is used of a 16-bit field (such as layer-4 destination port) where:

> ⚠ The same concept would apply to any other field (such as an IP address occupying 32 bits).

- P stands for the packet field value

- V stands for the pipe match field value
- M stands for the pipe mask field value
- E stands for the match entry field value

Implicit Match

Match Type	Pipe Match Value (V)	Pipe Match Mask (M)	Entry Match Value (E)	Filter (A)	Rule (B)
Ignore	0	NULL	N/A	0	0
Constant	0<V<0xffff	NULL	N/A	0xffff	V
Changeable (per entry)	0xffff	NULL	0≤E≤0xffff	0xffff	E

To match implicitly, the following considerations should be taken into account.

- Ignored fields:
  - Field is zeroed
  - Pipeline has no comparison on the field
- Constant fields – These are fields that have a constant value among all entries. For example, as shown in the following, the tunnel type is VXLAN:

  ```
  match.tun.type = DOCA_FLOW_TUN_VXLAN;
  ```

  These fields must only be configured once at pipe build stage, not once per new pipeline entry.
- Changeable fields – These are fields whose value may change per entry. For example, the following shows match on a destination IPv4 address of variable per-entry value (outer 5-tuple):

  ```
  match.outer.ip4.dst_ip = 0xffffffff;
  ```

- The following is an example of a match, where:
  - Outer 5-tuple
    - L3 type is IPv4 – constant among entries by design
    - L4 type is UDP – constant among entries by design
    - Tunnel type is **DOCA_FLOW_TUN_VXLAN** – constant among entries by design
    - IPv4 destination address varies per entry
    - UDP destination port is always **DOCA_FLOW_VXLAN_DEFAULT_PORT**
    - VXLAN tunnel ID varies per entry
    - The rest of the packet fields are ignored
  - Inner 5-tuple
    - L3 type is IPv4 – constant among entries by design
    - L4 type is TCP – constant among entries by design
    - IPv4 source and destination addresses vary per entry
    - TCP source and destination ports vary per entry
    - The rest of the packet fields are ignored

```
// filter creation
static void build_underlay_overlay_match(struct doca_flow_match *match)
{
    //outer
```

```
    match->outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;
    match->outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;
    match->tun.type = DOCA_FLOW_TUN_VXLAN;
    match->outer.ip4.dst_ip = 0xffffffff;
    match->outer.udp.l4_port.dst_port = DOCA_FLOW_VXLAN_DEFAULT_PORT;
    match->tun.vxlan_tun_id = 0xffffffff;

    //inner
    match->inner.l3_type = DOCA_FLOW_L3_TYPE_IP4;
    match->inner.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_TCP;
    match->inner.ip4.dst_ip = 0xffffffff;
    match->inner.ip4.src_ip = 0xffffffff;
    match->inner.tcp.l4_port.src_port = 0xffff;
    match->inner.tcp.l4_port.dst_port = 0xffff;
}

// create entry specifying specific values to match upon
doca_error_t add_entry(struct doca_flow_pipe *pipe, struct doca_flow_port *port,
                       struct doca_flow_pipe_entry **entry)
{
    struct doca_flow_match match = {};
    struct doca_flow_actions actions = {};
    struct entries_status status = {};
    doca_error_t result;

    match.outer.ip4.dst_ip = BE_IPV4_ADDR(7, 7, 7, 1);
    match.tun.vxlan_tun_id = RTE_BE32(9876);
    match.inner.ip4.src_ip = BE_IPV4_ADDR(8, 8, 8, 1);
    match.inner.ip4.dst_ip = BE_IPV4_ADDR(9, 9, 9, 1);
    match.inner.tcp.l4_port.src_port = rte_cpu_to_be_16(5678);
    match.inner.tcp.l4_port.dst_port = rte_cpu_to_be_16(1234);
    result = doca_flow_pipe_basic_add_entry(0, pipe, &match, 0, &actions, NULL, NULL, 0, &status, entry);
}
```

The fields of the **doca_flow_meta** struct inside the match are not subject to implicit match rules and must be paired with explicit mask values.

Explicit Match

Match Type	Pipe Match Value (V)	Pipe Match Mask (M)	Entry Match Value (E)	Filter (A)	Rule (B)
Constant	V!=0xffff	0<M≤0xffff	0≤E≤0xffff	M	M&V
Changeable	V==0xffff	0<M≤0xffff	0≤E≤0xffff	M	M&E
Ignored	0≤V<0xffff	M==0	0≤E≤0xffff	0	0

In this case, there are two **doca_flow_match** items, the following considerations should be considered:

- Ignored fields
  - M equals zero. This can be seen from the table where the rule equals 0. Since mask is also 0, the resulting packet after the filter is0. Thus, the comparison always succeeds.

```
match_mask.inner.ip4.dst_ip = 0;
```

- Constant fields
  These are fields that have a constant value. For example, as shown in the following, the inner 5-tuple match on IPv4 destination addresses belonging to the **0.0.0.0/24** subnet, and this match is constant among all entries:

```
// BE_IPV4_ADDR converts 4 numbers A,B,C,D to a big endian representation of IP address A.B.C.D
match.inner.ip4.dst_ip = 0;
match_mask.inner.ip4.dst_ip = BE_IPV4_ADDR(255, 255, 255, 0);
```

  For example, as shown in the following, the inner 5-tuple match on IPv4 destination addresses belonging to the **1.2.0.0/16** subnet, and this match is constant among all

entries. The last two octets of the **match.inner.ip4.dst_ip** are ignored because the **match_mask** of **255.255.0.0** is applied:

```
// BE_IPV4_ADDR converts 4 numbers A,B,C,D to a big endian representation of IP address A.B.C.D
match.inner.ip4.dst_ip = BE_IPV4_ADDR(1, 2, 3, 4);
match_mask.inner.ip4.dst_ip = BE_IPV4_ADDR(255, 255, 0, 0);
```

Once a field is defined as constant, the field's value cannot be changed per entry.

> Users should set constant fields to zero when adding entries for better code readability.

A more complex example of constant matches may be achieved as follows:

```
match_mask.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(0xf0f0);
match.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(0x5020)
```

The following ports would be matched:

- 0x5020 - 0x502f
- 0x5120 - 0x512f
- ...
- 0x5f20 - 0x5f2f

Changeable fields

The following example matches on either FTP or TELNET well known port numbers and forwards packets to a server after modifying the destination IP address and destination port numbers. In the example, either FTP or TELNET are forwarded to the same server. FTP is forwarded to port 8000 and TELNET is forwarded to port 9000.

```
// at Pipe creation
doca_flow_pipe_cfg_set_name(pipe_cfg, "PORT_MAPPER");
doca_flow_pipe_cfg_set_type(pipe_cfg, DOCA_FLOW_PIPE_BASIC);
match.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(0xffff); // v
match_mask.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(0xffff); // M
doca_flow_pipe_cfg_set_match(pipe_cfg, &match, &match_mask);
actions_arr[0] = &actions;
doca_flow_pipe_cfg_set_actions(pipe_cfg, actions_arr, NULL, NULL, 1);
doca_flow_pipe_cfg_set_is_root(pipe_cfg, true);

// Adding entries
// FTP
match.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(20); // E
actions.outer.ip4.src_ip = server_addr;
actions.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(8000);
result = doca_flow_pipe_basic_add_entry(0, pipe, &match, 0, &actions, NULL, NULL, 0, &status, entry);

// TELNET
match.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(23); // E
actions.outer.ip4.src_ip = server_addr;
actions.outer.tcp.l4_port.dst_port = rte_cpu_to_be_16(9000);
result = doca_flow_pipe_basic_add_entry(0, pipe, &match, 0, &actions, NULL, NULL, 0, &status, entry);
```

5.4.2.6.4.3 Relaxed Match

Relaxed matching mode grants users full control on matching fields and guarantees that no fields are implicitly added by DOCA Flow.

Relaxed Matching and Pipeline Design Considerations

Relaxed matching mode provides full control to the DOCA application developer over the match design, without adding implicit match logic by the DOCA Flow library. This approach increases user responsibility to prevent unintended side effects caused by packet layout similarities (e.g.,

between UDP and TCP source/destination ports) or by skipping header type validation before matching on header fields. For instance, matching solely on a UDP destination port does not verify the presence of an L4 header or confirm that the L4 header is of UDP type.

To ensure effective design, early-stage pipes should classify packet types to filter out undesired packets. As the pipeline progresses, later stages can focus on more specific packet types based on prior matches. Examples include:

- Match on VXLAN VNI:
  - For Early-stage pipes, ensure the packet contains a VXLAN header (this can be achieved in a single pipe):
    - The first pipe verifies the packet has a UDP header by matching the L4 packet type to UDP or the L3 **next_proto** field to UDP
    - The second pipe matches the UDP destination port to the commonly used VXLAN value (4789)
  - For later-stage pipes, match on the VXLAN VNI field
- Match on UDP destination port:
  - For early-stage pipes, verify the packet contains a UDP header by matching the L4 packet type to UDP or the L3 **next_proto** field to UDP
  - For later-stage pipes, match on the UDP destination port field

Relaxed Matching Memory Footprint and Performance Impact

Consider the following strict matching mode example. There are three pipes:

- Basic pipe **A** with **match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_TCP** and **match.outer.tcp.flags = 1**
- Basic pipe **B** with **match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP** and **match.outer.udp.l4_port.src_port = 8080**
- Control pipe **C** with two entries to direct TCP traffic to pipe **A** and UDP traffic to pipe **B**. The first entry has **match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_TCP** while the second has **match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP**.

As a result, the hardware matches on the L4 header type twice:

- When the packet enters the filter in control pipe **C** to decide the next pipe
- When the packet enters the filter of pipe **A** or **B** to match on the L4 header fields

With particularly large pipelines, such double matches decrease performance and increase the memory footprint in hardware. Relaxed matching mode gives the user greater control of the match logic to eliminate the implicitly added matches, consequently reducing hardware memory footprint and improving performance as well.

Parser Meta Usage with Relaxed Match

Parser meta matching is particularly useful when it comes to matching on a specific packet type. In relaxed mode, type selectors in the **outer**, **inner**, and **tun** parts of the **doca_flow_match** structs are used only for the type cast of the underlying unions. Header-type (packet type) matches are available using the **parser_meta** API.

For example, the scenario from the previous section may be overwritten by changing the match of control pipe **C** (with the same **A** and **B** pipes):

- Basic pipe **A** with **match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_TCP** and **match.outer.tcp.flags = 1**
- Basic pipe **B** with **match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP** and **match.outer.udp.l4_port.src_port = 8080**
- Control pipe **C** with two entries to direct TCP traffic to pipe **A** and UDP traffic to pipe **B**. The first entry has **match.parser_meta.outer_l4_type = DOCA_FLOW_L4_META_TCP** while the second has **match.parser_meta.outer_l4_type = DOCA_FLOW_L4_META_UDP**.

As a result, the hardware performs the L4 header-type match only once, when the packet enters the filter of control pipe. Basic pipes' **match.outer.l4_type_ext** are used only for the selection of the **match.outer.tcp** or **match.outer.udp** structures during the inspection of match struct.

Examples

The following code snippets are used to demonstrate the redesign of a pipeline with relaxed matching for non-tunnel match cases.

The following is the code before the redesign:

```
static void pipe_match_build(struct doca_flow_match *match)
{
    match->outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;
    match->outer.ip4.dst_ip = 0xffffffff;
    match->outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;
    match->outer.udp.l4_port.src_port = 22;
}
```

Without relaxed matching, the following hardware matches were performed for the code snippet above:

- L3 header type is IPv4 – constant among entries by design
- L4 header type is UDP – constant among entries by design
- IPv4 destination address varies per entry
- UDP source port is constant among entries
- The rest of the packet fields are ignored

With relaxed matching the following pipeline stages, where **pipe1** forwards packets to **pipe2**, should be considered to achieve a similar match as above:

```
static void pipe1_match_build(struct doca_flow_match *match)
{
    // Classifier logic. Only IPv4, UDP packets are to be forwarded to pipe2
    match.parser_meta.outer_l3_type = DOCA_FLOW_L3_META_IPV4;
    match.parser_meta.outer_l4_type = DOCA_FLOW_L4_META_UDP;
}

static void pipe2_match_build(struct doca_flow_match *match)
{
    // Main logic. Match on the specific packet fields
    match->outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;
    match->outer.ip4.dst_ip = 0xffffffff;
    match->outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;
    match->outer.udp.l4_port.src_port = 22;
}
```

The following code snippet demonstrates the redesign of a pipeline with relaxed matching for tunnel match cases:

```
static void pipe1_match_build(struct doca_flow_match *match)
{
    match->outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;
    match->outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;
```

```
    match->tun.type = DOCA_FLOW_TUN_VXLAN;
    match->outer.ip4.dst_ip = 0xffffffff;
    match->outer.udp.l4_port.src_port = 0x22;
    match->tun.vxlan_tun_id = 0xffffffff;
}
```

Without relaxed matching, the following hardware matches were performed for the code snippet above:

- L3 type is IPv4 – constant among entries by design
- L4 type is UDP – constant among entries by design
- Tunnel type is **DOCA_FLOW_TUN_VXLAN** – constant among entries by design
- IPv4 destination address varies per entry
- UDP source port is always 22
- VXLAN tunnel ID varies per entry
- The rest of the packet fields are ignored

With relaxed matching, the following pipeline stages, where **pipe1** forwards packets to **pipe2**, should be considered to achieve a similar match as above:

```
static void pipe1_match_build(struct doca_flow_match *match)
{
    // Classifier logic. Only IPv4, UDP packets are to be forwarded to pipe2
    match.parser_meta.outer_l3_type = DOCA_FLOW_L3_META_IPV4;
    match.parser_meta.outer_l4_type = DOCA_FLOW_L4_META_UDP;
}

static void pipe2_match_build(struct doca_flow_match *match)
{
    // Main logic. Match on the specific packet fields
    match->outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;
    match->outer.udp.l4_port.src_port = 22;
    match->outer.udp.l4_port.dst_port = DOCA_FLOW_VXLAN_DEFAULT_PORT;
    match->outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;
    match->outer.ip4.dst_ip = 0xffffffff;
    match->tun.type = DOCA_FLOW_TUN_VXLAN;
    match->tun.vxlan_tun_id = 0xffffffff;
}
```

Relaxed Matching Considerations

With relaxed matching, header type fields within **outer**, **inner**, or **tun** structs no longer indicate a match on the specific protocol. They are used solely as a selector for the relevant header fields. For example, to match on **outer.ip4.dst_ip**, users must set **outer.l3_type = DOCA_FLOW_L3_TYPE_IP4**. With this match, the L3 header is checked for the IPv4 destination address, however there is no check that the L3 header is of IPv4 type. It is user responsibility to make sure that packets arriving to such a filter indeed have an L3 header of type IPv4.

To match on a specific protocol/tunnel type, consider the following:

- To match on an inner/outer L3/L4 protocol type, users can use relevant **doca_flow_parser_meta** fields as explained above. For example, for outer protocols, **parser_meta.outer_l[3,4]_type** fields can be used.
- To match on a specific tunnel type, users should match on a tunnel according to its specification. For example, for a VXLAN tunnel, a match on UDP destination port 4789 can be used. Another option is to use the L3 next protocol field. For example, for IPv4 with next header GRE, one can match on the IPv4 headers' next protocol field value to match GRE IP protocol number 47.

More relaxed matching design best practices can be found in the samples Flow Drop, Flow VXLAN Encap, and LPM with Exact Match Logic.

⚠ With relaxed matching, to achieve a match-all functionality, either one of the following methods can be used during pipe creation:

- Set the **match_mask** structure to NULL and set the match structure to all zeroes
- Set the **match_mask** structure to all zeroes while the match structure have any setting

⚠ With relaxed matching, if any of the selectors is used without setting a relevant field, the pipe/entry creation would fail with the following error message:

```
failed building active opcode - active opcode <opcode number> is protocol only
```

5.4.2.6.4.4 Setting Pipe Actions

Pipe Actions Execution Order

When setting actions, they are executed in the following order:

1. Crypto (decryption)
2. Decapsulation
3. Pop
4. Meta
5. Outer
6. Tun
7. Push
8. Encapsulation
9. Crypto (encryption)

The **desc_array** field in **struct doca_flow_action_descs** is executed by order of the array after the above action execution.

⚠ A "next protocol" field selects the protocol of the header that follows it. For example:

- an EtherType (**eth.type**) — selects the L3 protocol (IPv4, IPv6, ...);
- an IPv4 protocol (**ipv4.next_proto**) or IPv6 next-header (**ipv6.next_proto**) — selects the L4 protocol (TCP, UDP, ...);
- a tunnel next-protocol (**vxlan_gpe.next_proto** , **geneve.next_proto**) — selects the protocol carried inside the tunnel.

Such a field can be set in several ways:

- directly, by a modify action on the field;
- indirectly, by pushing a header (e.g. a VLAN, whose EtherType selects what follows);
- indirectly, by encapsulating a header.

If a next-protocol field is set to an unsupported value, the following header can no longer be identified. No error is raised and the entry still reports success, but for that header and any after it:

- matching is not supported — matches will not hit;
- actions are unpredictable — they may be silently skipped, applied to the wrong location, or happen to work by chance.

Auto-modification

Similarly to setting pipe match, actions also have a template definition.

Similarly to `doca_flow_match` in the creation phase, only the subset of actions that should be executed per packet are defined. This is done in a similar way to match, namely by classifying a field of `doca_flow_actions` to one of the following:

- Ignored field – field is zeroed, modify is not used.
- Constant fields – when a field must be modified per packet, but the value is the same for all packets, a one-time value on action definitions can be used
- Changeable fields – fields that may have more than one possible value, and the exact values are set by the user per entry

```
actions.outer.ip4.dst_ip = 0xffffffff
```

⚠ The `action_mask` should be set as `0xffffffff` and action as 0 if the user wants to configure 0 to this field.

⚠ Changeable shared encap/decap actions must use shared resources that have been configured in advance.

⚠ A single pipe cannot be created with both a changeable shared encap/decap action and a changeable FWD action at the same time. This configuration is not supported and will fail.

⚠ To use both a changeable shared encap/decap action and a changeable FWD action, implement them in sequential pipes. Use one pipe for the encap/decap action, and then use a subsequent pipe for the FWD action.

Explicit Modification Type

It is possible to force constant modification or per-entry modification with action mask. For example:

```
static void
create_constant_modify_actions(struct doca_flow_actions *actions,
                               struct doca_flow_actions *actions_mask,
                               struct doca_flow_action_desc *desc)
{
        actions->outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_UDP;
        actions->outer.udp.src_port = 0x1234;
        actions_mask->outer.udp.src_port = 0xffff;
}
```

Copy Field

The action descriptor can be used to copy between the packet field and metadata. For example:

```
#define META_U32_BIT_OFFSET(idx) (offsetof(struct doca_flow_meta, u32[(idx)]) << 3)

static void
create_copy_packet_to_meta_actions(struct doca_flow_match *match,
                                   struct doca_flow_action_desc *desc)
{
        desc->type = DOCA_FLOW_ACTION_COPY;
        desc->field_op.src.field_string = "outer.ipv4.src_ip";
        desc->field_op.src.bit_offset = 0;
        desc->field_op.dst.field_string = "meta.data";
        desc->field_op.dst.bit_offset = META_U32_BIT_OFFSET(1); /* Bit offset of meta.u32[1] */;
}
```

Multiple Actions List

Creating a pipe is possible using a list of multiple actions. For example:

```
static void
create_multi_actions_for_pipe_cfg()
{
    struct doca_flow_actions *actions_arr[2];
    struct doca_flow_actions actions_0 = {0}, actions_1 = {0};
    struct doca_flow_pipe_cfg *pipe_cfg;

    /* input configurations for actions_0 and actions_1 */

    actions_arr[0] = &actions_0;
    actions_arr[1] = &actions_1;
    doca_flow_pipe_cfg_set_actions(pipe_cfg, actions_arr, NULL, NULL, 2);
}
```

Summary of Action Types

Pipe Creation				Entry Creation	Behavior
action_desc		Pipe Actions	Pipe Actions Mask	Entry Actions	
doca_flow_action_type	Configuration				
DOCA_FLOW_ACTION_AUTO/ action_desc = NULL	No specific config	**0**	**0**	N/A	Field ignored, no modification
		0	**mask** != 0	N/A	Apply **0** and **mask** to all entries
		val != 0 && **val** != 0xFF	**mask** != 0	N/A	Apply **val** and **mask** to all entries
		val = 0xFF	**mask** = 0	N/A	Apply **0xFF** to all entries
		val = 0xFF	**mask** != 0	Define **val** per entry	Apply entry's **val** and **mask**
DOCA_FLOW_ACTION_ADD Add field value or from **src**	Define only the **dst** field and width	**val** != 0	N/A	N/A	Add to dst this **val** to all entries
		val == 0	N/A	Define **val** per entry	Add to dst entry's **val**

Pipe Creation				Entry Creation	Behavior
action_desc		Pipe Actions	Pipe Actions Mask	Entry Actions	
doca_flow_action_type	Configuration				
	Define the `src` and `dst` fields and width	Define the source and destination fields. • Meta field → header field • Header field → meta field • Meta field → meta field	N/A	N/A	Add data from `src` fields to `dst` for all entries
`DOCA_FLOW_ACTION_COPY` Copy field to another field	N/A	Define the source and destination fields. • Meta field → header field • Header field → meta field • Meta field → meta field	N/A	N/A	Copy data between fields for all entries

Queue Balancing for Non-shared Actions

For non-shared actions, each action resource handle is implicitly bound to a specific entry. To enable faster, lock-free access during entry creation, these action resource handles are cached on a per-queue basis, mirroring how entries are added.

Therefore, applications should distribute entry additions evenly across all available queues. This practice prevents cache fragmentation and ensures efficient use of action resources.

If entries are distributed unevenly, it can lead to a worst-case scenario where the number of successfully attached action handles is slightly lower than the total number of defined resources, even if those resources are technically available system-wide.

5.4.2.6.4.5 Setting Pipe Monitoring

If a meter policer should be used, then it is possible to have the same configuration for all policers on the pipe or to have a specific configuration per entry. The meter policer is determined by the FWD action. If an entry has NULL FWD action, the policer FWD action is taken from the pipe.

The monitor also includes the aging configuration, if the aging time is set, this entry ages out if timeout passes without any matching on the entry.

For example:

```
static void build_entry_monitor(struct doca_flow_monitor *monitor, void *user_ctx)
{
    monitor->aging_sec = 10;
}
```

Refer to Pipe Entry Aged Query for more information.

5.4.2.6.4.6 Setting Pipe Forwarding

The FWD (forwarding) action is the last action in a pipe, and it directs where the packet goes next. Users may configure one of the following destinations:

- Send to software (representor)
- Send to wire
- Jump to next pipe
- Drop packets

The FORWARDING action may be set for pipe create, but it can also be unique per entry.

Pipe forwarding can be set either at creation time or be deferred to entry addition:

- If the **fwd.type** type is not **DOCA_FLOW_FWD_CHANGEABLE**, any match on the pipe, on any entry, is forwarded to the specified target
- If the **fwd.type** type is **DOCA_FLOW_FWD_CHANGEABLE**, the target would match on whatever is defined in the **fwd.type** field of the specific entry

Putting this logic in a table look like this:

Pipe Fwd.type	Entry Fwd.type	Actual Forward	Comment
Equals **DOCA_FLOW_FWD_CHANGEABLE**	X	X	X must not equal **DOCA_FLOW_FWD_CHANGEABLE**
X != **DOCA_FLOW_FWD_CHANGEABLE**	Does not care	X	

When a pipe includes meter monitor **<cir, cbs>**, it must have **fwd** defined as well as the policer.

The following is an RSS forwarding example:

```
fwd.type = DOCA_FLOW_FWD_RSS;
fwd.rss_type = DOCA_FLOW_RESOURCE_TYPE_NON_SHARED;
fwd.rss.queues_array = queues;
fwd.rss.outer_flags = DOCA_FLOW_RSS_IPV4 | DOCA_FLOW_RSS_UDP;
fwd.rss.nr_queues = 4;
```

Queues point to the **uint16_t** array that contains the queue numbers. When a port is started, the number of queues is defined, starting from zero up to the number of queues minus 1. RSS queue

numbers may contain any subset of those predefined queue numbers. For a specific match, a packet may be directed to a single queue by having RSS forwarding with a single queue.

Changeable RSS forwarding is supported. When creating the pipe, the **num_of_queues** must be set to **0xffffffff**, then different forwarding RSS information can be set when adding each entry.

```
fwd->num_of_queues = 0xffffffff;
```

The packet is directed to the port. In many instances the complete pipe is executed in the hardware, including the forwarding of the packet back to the wire. The packet never arrives to the software.

Example code for forwarding to port:

```
struct doca_flow_fwd *fwd = malloc(sizeof(struct doca_flow_fwd));
memset(fwd, 0, sizeof(struct doca_flow_fwd));
fwd->type = DOCA_FLOW_FWD_PORT;
fwd->port_id = port_id; // this should be the same port_id as set in doca_flow_port_cfg_set_port_id()
```

The type of forwarding is **DOCA_FLOW_FWD_PORT** and the only data required is the **port_id** as defined in **DOCA_FLOW_PORT**.

The destination port for any FWD action (identified by **fwd->port_id**) must be started in advance by calling **doca_flow_port_start()**.

A single pipe cannot be created with both a changeable shared encap/decap action and a changeable FWD action simultaneously. This configuration is not supported.

Changeable port forwarding is also supported. When creating the pipe, the **port_id** must be set to **0xffff**, then different forwarding **port_id** values can be set when adding each entry.

```
fwd->port_id = 0xffff;
```

5.4.2.6.4.7 Shared Resources

DOCA Flow supports several types of resources that can be shared. The supported types of resources can be:

- Meters
- Counters
- RSS queues
- PSPs
- Encap
- Decap
- IPsec SA
- Trim

Shared resources can be used by several pipes and can save device and memory resources while promoting better performance.

To create and configure shared resource, the user should go through the steps detailed in the following subsections.

> ⚠ The per-port shared-resource APIs used throughout this section (`doca_flow_port_cfg_set_nr_resources()`, `doca_flow_port_shared_resource_set_cfg()`, `doca_flow_port_shared_resource_get()`, `doca_flow_port_shared_resource_put()`, and `doca_flow_port_shared_resources_query()`) are currently experimental and may change until they are promoted to stable. The deprecated system-wide APIs they replace are stable, so callers migrating now should account for the experimental status of the replacements.

Creating Shared Resource Configuration Object

Create a DOCA Flow port configuration object by calling `doca_flow_port_cfg_create(&port_cfg)`.

> ⚠ Global DOCA Flow initialization via `doca_flow_cfg_create()` and `doca_flow_init()` is still required before starting ports. Shared-resource sizing and binding are configured per port via `struct doca_flow_port_cfg` and `struct doca_flow_port`.

Before using the per-port shared-resource APIs in this section, enable per-port resource mode by calling `doca_flow_cfg_set_resource_mode(flow_cfg, DOCA_FLOW_RESOURCE_MODE_PORT)` on the global configuration object before `doca_flow_init()`. DOCA Flow defaults to system resource mode; enabling port mode is a required precondition for these APIs, so call them only after port mode is enabled. Do not select `DOCA_FLOW_RESOURCE_MODE_SYSTEM`.

> ⚠ `doca_flow_cfg_set_resource_mode()` is experimental and planned for deprecation once per-port is the only supported model.

Setting Number of Shared Resources per Shared Resource Type

Set the number of shared resources for each type on the port configuration object by calling `doca_flow_port_cfg_set_nr_resources()` with the matching `enum doca_flow_resource` value (for example, `DOCA_FLOW_RESOURCE_COUNTER` or `DOCA_FLOW_RESOURCE_METER`). Refer to the API documentation for details on the configuration process.

Start the port by calling `doca_flow_port_start()` with the configured `struct doca_flow_port_cfg`.

> ⚠ The legacy system-wide sizing APIs `doca_flow_cfg_set_nr_shared_resource()`, `doca_flow_cfg_set_nr_counters()`, and `doca_flow_cfg_set_nr_meters()` are deprecated. Use `doca_flow_port_cfg_set_nr_resources()` instead.

Getting a Shared Resource ID

Obtain a shared-resource ID on the port by calling `doca_flow_port_shared_resource_get()`. These per-port APIs require per-port resource mode to be enabled first (see the prerequisite above); they are not supported in the default system mode. Each call returns the next available ID for that

resource type on the port, and returns **DOCA_ERROR_EMPTY** once the port's configured resources of that type are exhausted. When the ID is no longer needed, return it with **doca_flow_port_shared_resource_put()**.

> ⚠ Each resource type has its own ID space. The first IDs handed out for a type start at 0 and increase incrementally, so getting two shared counters and three meters yields identifiers 0..1 and 0..2 respectively. IDs released with **doca_flow_port_shared_resource_put()** are recycled, so a later **doca_flow_port_shared_resource_get()** may return a previously released ID — do not assume IDs are monotonic or never reused.

> ⚠ The legacy global binding API **doca_flow_shared_resources_bind()** is deprecated. Use **doca_flow_port_shared_resource_get()** and **doca_flow_port_shared_resource_put()** instead.

Configuring a Shared Resource

Configure a shared resource on the port by calling **doca_flow_port_shared_resource_set_cfg()**, passing an ID previously obtained from **doca_flow_port_shared_resource_get()**.

Using Shared Resources

After a resource has been configured, it can be used by referring to its ID.

In the case of meters, counters, they are referenced through **struct doca_flow_monitor** during pipe creation or entry addition.

Querying Shared Resource

Querying shared resources can be done by calling **doca_flow_port_shared_resources_query()**. The function accepts the port, resource type, and an array of resource numbers and returns an array of **struct doca_flow_resource_query** with the results.

> ⚠ The legacy global query API **doca_flow_shared_resources_query()** is planned for deprecation. Use **doca_flow_port_shared_resources_query()** instead.

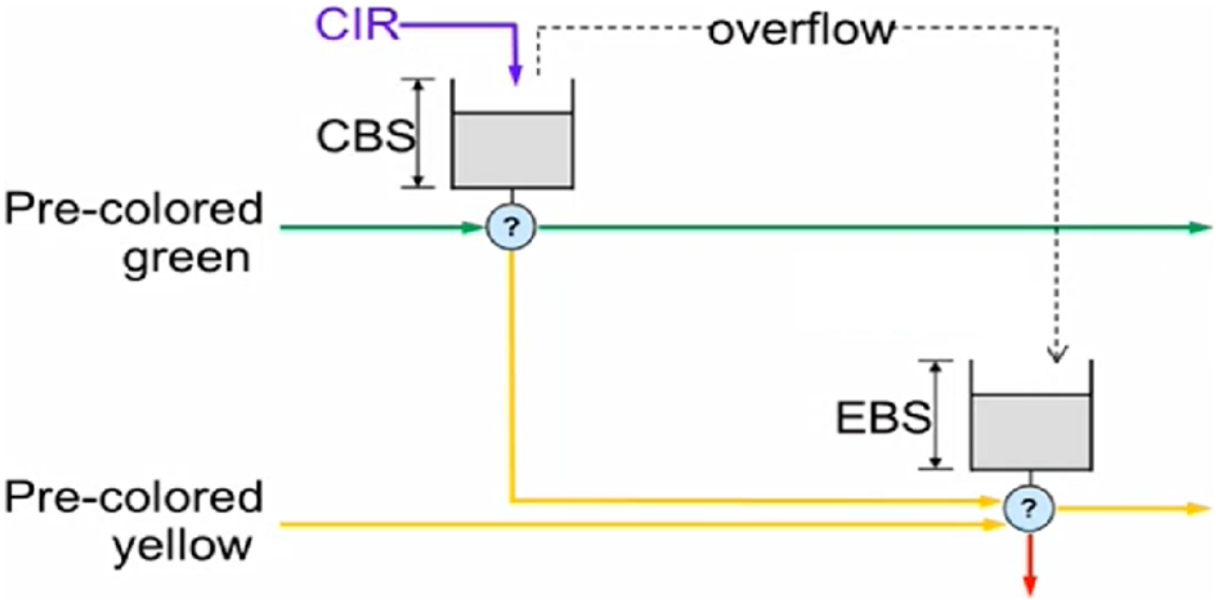
Shared Meter Resource

A shared meter can be used in multiple pipe entries (hardware steering mode support only).

The shared meter action marks a packet with one of three colors: Green, Yellow, and Red. The packet color can then be matched in the next pipe, and an appropriate action may be taken. For example, packets marked in red are usually dropped. So, the next pipe to meter action may have an entry which matches on red and has fwd type **DOCA_FLOW_FWD_DROP**.

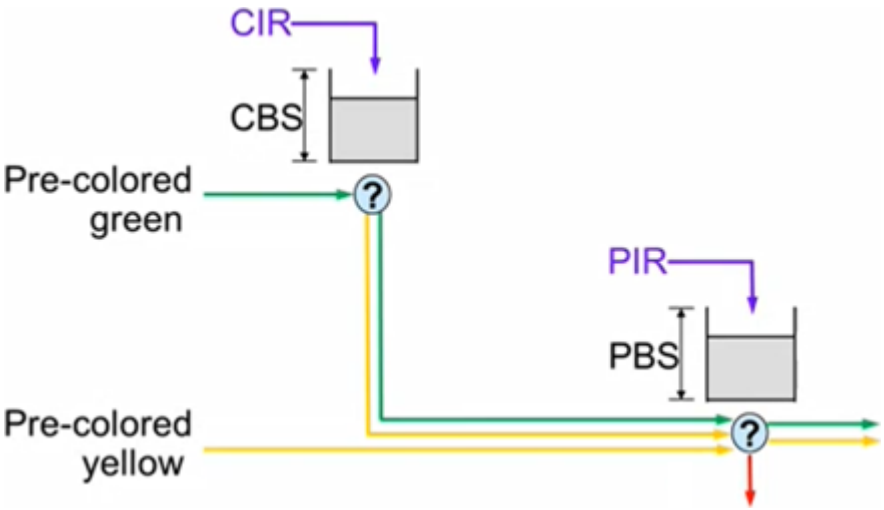
DOCA Flow supports three marking algorithms based on RFCs: 2697, 2698, and 4115.

RFC 2697 – Single-rate Three Color Marker (srTCM)



CBS (committed burst size) is the bucket size which is granted credentials at a CIR (committed information rate). If CBS overflow occurs, credentials are passed to the EBS (excess burst size) bucket. Packets passing through the meter consume credentials. A packet is marked green if it does not exceed the CBS, yellow if it exceeds the CBS but not the EBS, and red otherwise. A packet can have an initial color upon entering the meter. A pre-colored yellow packet will start consuming credentials from the EBS.

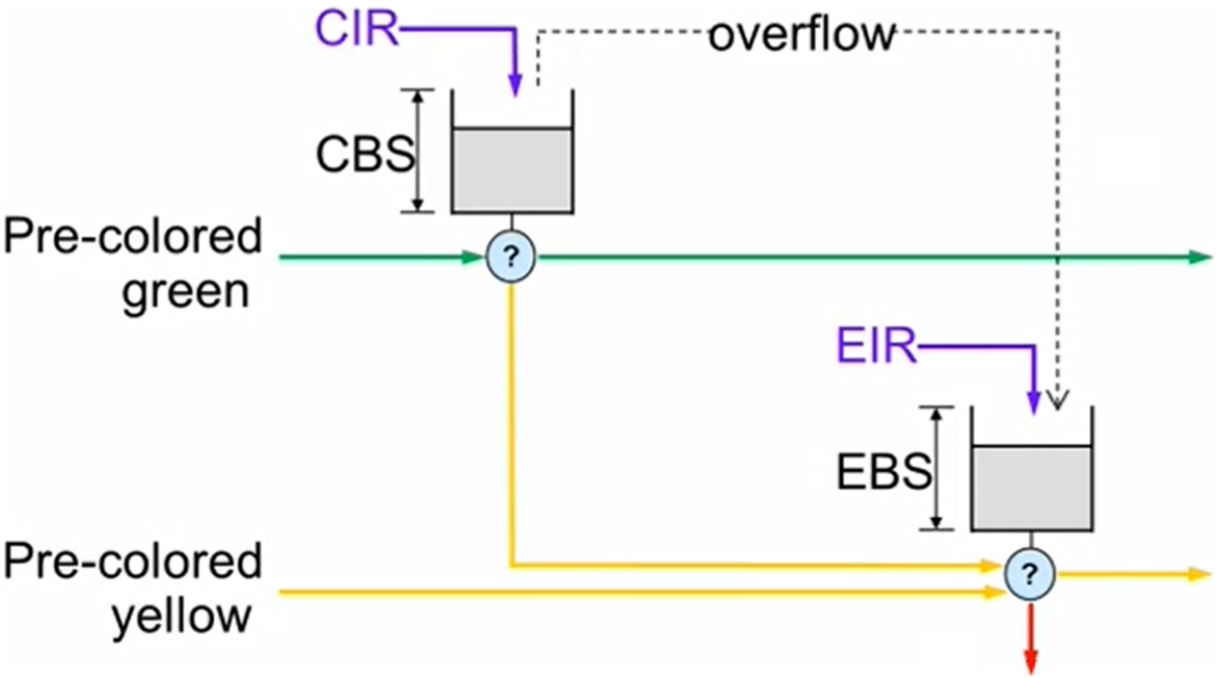
RFC 2698 – Two-rate Three Color Marker (trTCM)



CBS and CIR are defined as in RFC 2697. PBS (peak burst size) is a second bucket which is granted credentials at a PIR (peak information rate). There is no overflow of credentials from the CBS bucket to the PBS bucket. The PIR must be equal to or greater than the CIR. Packets consuming CBS credentials consume PBS credentials as well. A packet is marked red if it exceeds the PIR.

Otherwise, it is marked either yellow or green depending on whether it exceeds the CIR or not. A packet can have an initial color upon entering the meter. A pre-colored yellow packet starts consuming credentials from the PBS.

RFC 4115 – trTCM without Peak-rate Dependency



EBS is a second bucket which is granted credentials at a EIR (excess information rate) and gets overflowed credentials from the CBS. For the packet marking algorithm, refer to RFC 4115.

All information rates (i.e., CIR, PIR, EIR) are measured in one of two units:

- Per bytes (**DOCA_FLOW_METER_LIMIT_TYPE_BYTES**); or
- Per packets (**DOCA_FLOW_METER_LIMIT_TYPE_PACKETS**)

The following sections present the steps for configuring and using shared meters to mark packets.

Bandwidth Measurement

When testing a DOCA Flow meter, it is important to understand that the rate-limiting parameters (CIR, PIR, EIR) apply to the information rate, not the full L2 Ethernet frame. The information rate typically refers to the inner IP layer of the packet.

Because of this, to accurately measure the metered bandwidth and verify that it matches your settings, you must perform a mathematical adaptation to exclude the non-IP layer overhead from your byte count.

Measurement Method

The recommended method is to take two measurements, 1 second apart, after the traffic stream has been stable for at least 5 seconds.

Terminology:

- **pckts1** and **bytes1**: The total number of packets and bytes measured at time **X** (where **X** ≥ 5 seconds).
- **pckts2** and **bytes2**: The total number of packets and bytes measured at time **X+1** second.

The true metered bandwidth (in bytes per second) is the adapted byte count, which is calculated by subtracting the total overhead from the total bytes measured during that one-second interval:

```
Adapted Byte Count = (bytes2 - bytes1) - (overhead_bytes * (pckts2 - pckts1))
```

Here, **overhead_bytes** is the size of all non-IP-layer headers for your specific packet type.

Overhead Examples

The **overhead_bytes** value depends on the packet's structure. Common examples include:

- Simple IP packets: **18 bytes**
  - (14-byte MAC header + 4-byte FCS)
- VLAN packets: **22 bytes**
  - (14-byte MAC header + 4-byte VLAN tag + 4-byte FCS)
- MPLS tunnel packets: **50 bytes**
  - (14-byte MAC header + 4-byte MPLS + 20-byte Tunnel IP + 8-byte Tunnel UDP + 4-byte FCS)

Verifying the Rate

You can then compare the calculated bandwidth against your meter's settings:

- For byte-based limits (**DOCA_FLOW_METER_LIMIT_TYPE_BYTES**): The expected bandwidth is the Adapted Byte Count calculated above.
- For packet-based limits (**DOCA_FLOW_METER_LIMIT_TYPE_PACKETS**): The expected bandwidth is the packet delta: **(pckts2 - pckts1)**.

Shared IPsec SA Resource

The Shared IPsec Security Association (SA) resource is used to manage the IPsec ESP protocol for both encryption and decryption. This resource is referenced from the **doca_flow_actions** struct via the **crypto** field.

Default Hardware Offload Behavior

By default, the resource manages the sequence number (SN) state in hardware (i.e., **sn_offload** is enabled). This includes:

- Encryption: Automatically incrementing the sequence number (SN) for each packet.
- Decryption: Performing anti-replay protection. The anti-replay status (syndrome) is stored in **meta.u32[0]**.

Manual Software SN Control

To manage the sequence number in software, the **sn_offload** feature must be disabled in the port configuration passed to **doca_flow_port_start** (see the DOCA API References for details).

> ⚠ When **sn_offload** is disabled, the following hardware offload fields in the resource configuration are ignored:
>
> - **sn_offload_type**
> - **win_size**
> - **sn_initial**
> - **lifetime_threshold**

Querying the Resource

When **sn_offload** is enabled, you can query the shared IPsec SA resource to retrieve its current state:

- For an encryption resource: The query returns the current sequence number (SN).
- For a decryption resource: The query returns the lower bound of the anti-replay window.

> ⚠ Querying the resource is not supported when **sn_offload** is disabled.

State Maintenance

To ensure a valid resource state during operation, the **doca_flow_crypto_ipsec_resource_handle** function should be called periodically.

Shared Encap Resource

The shared encap resource defines a specific, reusable encapsulation configuration. Each unique configuration, primarily defined by its reformat type (**is_l2**) and size, is represented by a shared encap ID.

This ID allows the same encapsulation to be applied across multiple pipes and entries.

Key characteristics:

- This feature is supported only in hardware steering mode.
- Once a shared encap resource is configured and assigned an ID, its settings (like type and size) cannot be modified.
- The "shared encap action" in a pipe uses this resource ID to encapsulate a packet with the pre-configured tunnel information.

Shared Decap Resource

The shared decap resource defines a specific, reusable decapsulation configuration. Each unique configuration, primarily defined by its reformat type (**is_l2**) and size, is represented by a shared decap ID.

This ID allows the same decapsulation to be applied across multiple pipes and entries.

Key characteristics:

- This feature is supported only in hardware steering mode.
- Once a shared decap resource is configured and assigned an ID, its settings (like type and size) cannot be modified.
- If the **is_l2** flag is set to false (indicating a non-L2 decap), Ethernet information must be provided in the configuration.
- The "shared decap action" in a pipe uses this resource ID to decapsulate the packet according to the pre-configured rules.

Shared PSP Resource

The PSP shared resource is used for PSP encryption. The resource should be pointed to from the **doca_flow_crypto_action** struct in **doca_flow_actions**.

The resource should be configured with a key to encrypt the packets. See DOCA API References for PSP key generation for a reference about key handling on decrypt side.

Shared Trim Resource

The trim shared resource lets a pipe entry trim its matched packet to a fixed length and continue processing the trimmed copy at a user-chosen target pipe.

Configure the resource with **struct doca_flow_resource_trim_cfg**:

- **size** — number of bytes to keep on the trimmed copy (counted from the start of the packet), must be at least 64 bytes.
- **target** — **struct doca_flow_fwd** describing where the trimmed packet continues. The target must be a pipe (**DOCA_FLOW_FWD_PIPE**) on a port that itself owns at least one trim shared resource. The **target.next_pipe** field must be non-NULL.

Reference the configured resource from a pipe by setting the entry's forward to **DOCA_FLOW_FWD_TRIM** and filling **fwd.trim.trim_id** with the resource id allocated at configure time. Both fixed and **DOCA_FLOW_FWD_CHANGEABLE** forwarding are supported, so the same pipe can route different entries to different trim resources.

Key characteristics:

- Supported in hardware steering mode, in both VNF mode and Switch mode.
- Multiple trim shared resources may coexist on the same port.
- Once a trim shared resource is configured and assigned an id, its **size** and **target** cannot be modified, must destroy and recreate the resource to change it.

Limitations:

- In VNF mode, the trim resource's **target** pipe must be created on the egress domain (**DOCA_FLOW_PIPE_DOMAIN_EGRESS**).
- After trimming, none of the metadata fields are preserved. All metadata scratch area is set to 0 and must not be matched or read post-trim.
- Packet ordering is not preserved in the following cases: between packets sent to different trim resources; between trimmed and non-trimmed packets.

5.4.2.6.4.8 Creating a Pipe

Once all pipe parameters are defined, call `doca_flow_pipe_create` to create the pipe.

> Refer to the DOCA API documentation to see which fields are optional and may be skipped. It is typically recommended to set optional fields to 0 when not in use. See Miss Pipe and Control Pipe for more information.

On success, this function returns a handle to the pipe, which is required when adding new entries. If creation fails, the function returns `NULL`. An error struct, if provided by the user, will be populated with the error details.

A pipe acts as a container for its entries. When a pipe is destroyed, all entries bound to it are automatically removed. For more details, see the Pipe Entry section.

DOCA Flow Connection Tracking Pipe

CT pipes use additional `doca_flow_pipe_cfg` parameters beyond standard pipe creation. For configuration details (session pools, zone limits, duplication filter, etc), see CT Pipe Configuration in the DOCA Flow Connection Tracking guide.

Implementing Priority

DOCA Flow does not have an explicit priority field for pipes or entries. Priority is managed through the order of operation and rule insertion.

- Chaining (Recommended) – The most explicit way to implement priority is to create a chain. Create a "miss" rule in a high-priority pipe that forwards traffic to the next, lower-priority pipe (e.g., match `PIPE1`, and on miss, jump to `PIPE2`).
- Insertion Order (Overlapping Entries) – Priority can also be determined by the order in which entries are added to pipes.
  - If two pipes have overlapping match criteria, the pipe whose entries were added last has the higher priority.
  - For example, to make `PIPE1` (high-priority) take precedence over `PIPE2` (low-priority), you must add all entries for `PIPE2` first, *followed* by the overlapping entries for `PIPE1`.

Pipe Sizing and Caching

When you define the number of entries for a pipe, this number must account for both active entries and entries held in internal caches.

This means the actual number of entries your application can add will be less than the total defined size. The total number of cached entries is limited to (1K * number of queues), and this cache size cannot exceed 2% of the total pipe size.

5.4.2.6.4.9 Pipe Entry (doca_flow_pipe_basic_add_entry)

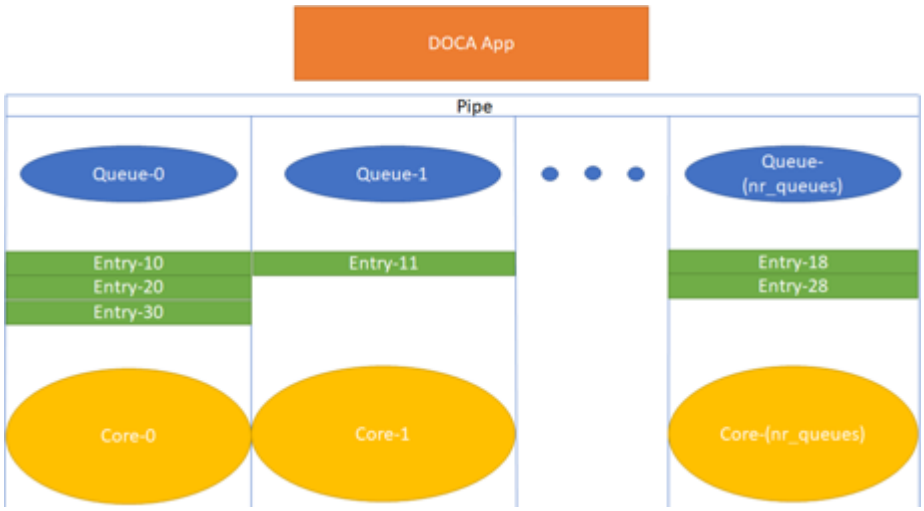
An entry is a specific instance inside of a pipe. When defining a pipe, users define match criteria (subset of fields to be matched), the type of actions to be done on matched packets, monitor, and, optionally, the FWD action.

When a user calls `doca_flow_pipe_basic_add_entry()` to add an entry, they must define the values for the match criteria and action parameters. If a Forward (FWD) action was not defined as constant for the entire pipe, it is mandatory to specify the FWD action for each entry.

Concurrency and Queue Affinity

DOCA Flow is designed to support millions of new entries per second. To achieve this, its architecture requires that entry-addition operations be parallelized.

This is managed by assigning a dedicated `pipe_queue` ID to each CPU core that will be adding entries. Having a unique queue ID per core prevents the need for data-structure locking and enables efficient, parallel interaction with the hardware.



Each core must use its own dedicated `pipe_queue` ID when calling `doca_flow_pipe_basic_add_entry`. Using the same `pipe_queue` ID from different cores simultaneously will cause a race condition and lead to undefined behavior.

Usage Notes

- Applications must not add, remove, or update pipe entries from within a `doca_flow_entry_process_cb` callback.
- Applications should distribute entry additions evenly across all queues. This prevents cache fragmentation and ensures efficient resource use. In a worst-case scenario where entries are not balanced, the number of successfully inserted entries may be slightly lower than the total number defined during pipe creation.

Pushing Entries to Hardware

When calling `doca_flow_pipe_basic_add_entry()`, the `flags` parameter controls when the entry is pushed to hardware:

- `DOCA_FLOW_ENTRY_FLAGS_WAIT_FOR_BATCH` – the entry waits to be pushed to hardware together with subsequent entries. Batching entries reduces the number of pushes and increases the insertion rate.

- `DOCA_FLOW_ENTRY_FLAGS_NO_WAIT` – the entry is pushed to hardware immediately.

Entry Processing and Completion

`doca_flow_entries_process()` processes all the pending entries on a given queue. This covers add, update, and remove operations. Once an operation has been handled and its status is known, the completion callback (set via `doca_flow_cfg`) is invoked with the resulting status, the operation type (add/update/remove), and the entry's `usr_ctx` . The `usr_ctx` is the pointer supplied when the entry was added (the add APIs take a `usr_ctx` parameter; update and remove reuse the context associated with the entry).

If no callback is set in `doca_flow_cfg` , the application must still call `doca_flow_entries_process()` on the queue to progress operations, and can read an entry's status with `doca_flow_pipe_entry_get_status()` (which only reports the currently known status) to check whether it has completed offloading. This applies only to live entries — `doca_flow_pipe_remove_entry()` only schedules an asynchronous removal, and the entry is destroyed while its remove completion is processed; once removal has been requested the entry handle becomes invalid and must not be used or queried.

`doca_flow_pipe_queue_drain()` complements `doca_flow_entries_process()` as a control-step API: it pushes any pending entry operations on the given queue to hardware and then waits for all in-progress operations on that queue to complete, optionally reporting the number of operations drained. The wait is bounded by an internal timeout: the call returns `DOCA_ERROR_DRIVER` if the queue does not empty in time or the underlying flush/poll fails. Use `doca_flow_entries_process()` for periodic processing (bounded by a number of operations or a timeout) and for triggering pipe resize relocation; use `doca_flow_pipe_queue_drain()` when the application needs to complete all outstanding operations on a queue in a single control step. As with `doca_flow_entries_process()` , per-entry results are still delivered through the status callback (or `doca_flow_pipe_entry_get_status()`) — a successful drain reports that the queue was emptied, not that every entry succeeded. Connection tracking (CT) queues are not supported by this API; use `doca_flow_entries_process()` for those.

Failure Path

Entry insertion can fail in two places, `add_entry` and `add_entry_cb` .

- When `add_entry` fails, no cleanup is required.
- When `add_entry` succeeds, a handle is returned to the user. If the subsequent `add_entry_cb` fails, the user is responsible for releasing the handle through a `rm_entry` call. This `rm_entry` call is expected to return `DOCA_SUCCESS` and is expected to invoke `doca_rm_entry_cb` with a successful return code.

Pipe Entry Counting

By default, no counter is added. If defined in monitor, a unique counter is added per entry.

Having a counter per entry affects performance and should be avoided if it is not required by the application.

The retrieved statistics are stored in struct **doca_flow_resource_query**.

> ⚠ Counters have a granularity of 1 second.

Pipe Entry Aged Query

When a user calls **doca_flow_aging_handle()**, this query is used to get the aged-out entries by the time quota in microseconds. The user callback is invoked by this API with the aged entries.

Since the number of flows can be very large, the query of aged flows is limited by a quota in microseconds. This means that it may return without all flows and requires the user to call it again. When the query has gone over all flows, a full cycle is done.

5.4.2.6.4.10 Pipe Entry (doca_flow_pipe_basic_update_entry)

Updating an entry requires available space in the pipe. If the pipe is full, the function returns **DOCA_ERROR_AGAIN**. To proceed with the update, you may need to release an existing entry to free up space.

5.4.2.6.4.11 Pipes with Multiple Actions

DOCA Flow allows you to define a pipe with an array of multiple, distinct actions. This feature is useful for creating multiple flow entries that share the same match criteria but require different actions.

When adding or updating an entry (e.g., with **doca_flow_pipe_basic_add_entry**), you select which action to execute by providing its index in the **action_idx** argument.

> ⚠ Even if a pipe has multiple actions defined, only one action will be executed for any given packet that matches an entry.

Consider a case where two flows are needed with identical match criteria but different actions:

1. Pipe creation: A pipe is created with two actions defined in its **actions** array:
   - Index 0: An action to **modify_mac**.
   - Index 1: An action to **modify_ip**.
2. Entry addition:
   - Entry A is added (e.g., **match L4_PORT=80**) with its **action_idx** set to **0**. Packets matching this entry will have their MAC modified.
   - Entry B is added (e.g., **match L4_PORT=80**) with its **action_idx** set to **1**. Packets matching this entry will have their IP modified.

5.4.2.6.4.12 Pipe Chaining, Priority, and Miss Handling

This section describes how to chain pipes together to implement priority, how to use Control Pipes to resolve complex matching conflicts, and how the Port Default Miss handler works.

Implementing Priority with fwd_miss (Miss-Pipe Chaining)

DOCA Flow implements priority by chaining pipes together. A pipe's **fwd_miss** handler defines what to do with a packet if it does not match any entry within that pipe, allowing you to create a "miss-pipe" chain:

1. A packet is looked up in **Pipe 1** (the highest priority table).
2. If it matches an entry, the action is taken.
3. If it does not match, the **fwd_miss** handler forwards the packet to **Pipe 2**.
4. The lookup process repeats in **Pipe 2**.

This behavior is configured in the **struct doca_flow_fwd** of a pipe. When creating a pipe, the **fwd_miss** struct's **next_pipe** field specifies where to send non-matching packets.

The **fwd_miss** handler supports three forwarding types:

- **DOCA_FLOW_FWD_PIPE**: Forwards the packet to another pipe for further processing.
- **DOCA_FLOW_FWD_DROP**: Drops the packet.
- **DOCA_FLOW_FWD_TARGET**: Forwards to a target such as the kernel (see **doca_flow_get_target()**).

Other forwarding types, such as forwarding to a port (**DOCA_FLOW_FWD_PORT**) or an RSS queue (**DOCA_FLOW_FWD_RSS**), are not supported in a **fwd_miss** handler.

Root Pipes vs. Chained Pipes

A pipe's operational role within the DOCA Flow architecture is defined by the **is_root** flag within its **struct doca_flow_pipe_cfg**:

Pipe Type	**is_root** Flag	Execution Trigger	Characteristics & Constraints
Root Pipe	**true**	Executed immediately upon packet arrival.	• Limited strictly per direction (not per domain). • Ingress slot: **DEFAULT** and **SECURE_INGRESS** share a single root slot; only one can act as the root. • Egress slot: **EGRESS** and **SECURE_EGRESS** share a single root slot. • To route traffic to multiple pipelines simultaneously, you must create a single root control pipe that delegates packets to subsequent non-root pipes.
Chained Pipe (Non-Root)	**false**	Executed strictly when targeted by a forwarding action.	• Never executed directly upon initial packet arrival. • Must be reached via a **DOCA_FLOW_FWD_PIPE** action (e.g., routed from another pipe's explicit flow entry or triggered by a **fwd_miss** fallback handler).

Control Pipes and Prioritized Entries

Internally, DOCA Flow implements the **fwd_miss** handler as a flow entry with the lowest possible priority. This can create conflicts.

For example, consider a pipe that:

1. Matches specific VXLAN flows and sends them to RSS.

2. Has a **fwd_miss** handler to hairpin all other (non-matching) packets.

If a new VXLAN packet arrives for an IP that does not have a specific flow entry, it will not match (1) and will incorrectly fall through to (2), where it is hairpinned instead of being sent to RSS.

The solution is to use a Control Pipe. A Control Pipe allows you to add entries with *explicit* priorities, giving you fine-grained control to resolve these conflicts.

- You add prioritized entries to a Control Pipe by calling **doca_flow_pipe_control_add_entry()**.
- This function includes a **priority** field (where 0 is the highest and 7 is the lowest), which overrides the default insertion-order priority.

Example for Resolving Conflicts with Control Entries

Using the scenario above, you can fix the conflict with a Control Pipe:

1. Configure **fwd_miss**: Set the pipe's **fwd_miss** handler to hairpin packets. This creates an internal "hairpin" rule at the lowest priority (e.g., priority 7).
2. Add a control entry: Call **doca_flow_pipe_control_add_entry()** to add a new, higher-priority rule:
    - Priority: **6** (higher than the **fwd_miss** rule's priority of 7).
    - Match: Match *all* VXLAN packets (not just specific IPs).
    - Forward: Forward to the RSS pipe.

As a result, the new VXLAN packet no longer matches a specific IP, but it *will* match the new "all VXLAN" control entry at priority 6. It is correctly forwarded to RSS and never falls through to the lowest-priority (7) **fwd_miss** rule.

Updating fwd_miss at Runtime

A pipe's **fwd_miss** handler can be updated dynamically after creation using **doca_flow_pipe_update_miss()**, without needing to recreate the pipe:

```
doca_error_t doca_flow_pipe_update_miss(struct doca_flow_pipe *pipe, const struct doca_flow_fwd *fwd_miss);
```

The same forwarding type restrictions apply as during pipe creation (**DOCA_FLOW_FWD_PIPE**, **DOCA_FLOW_FWD_DROP**, or **DOCA_FLOW_FWD_TARGET**).

Example:

```
struct doca_flow_fwd new_fwd_miss = {.type = DOCA_FLOW_FWD_PIPE, .next_pipe = new_target_pipe};
doca_flow_pipe_update_miss(pipe, &new_fwd_miss);
```

Port Default Miss

This is the final fallback action for the port, used when a pipe's **fwd_miss** is not set (i.e., **NULL**).

By default, this forwards all non-matched traffic to the kernel and is active as soon as the port is started. The target of this default miss can be controlled using the following API:

```
enum doca_flow_port_default_miss_type {
    DOCA_FLOW_PORT_DEFAULT_MISS_KERNEL = 0,
    DOCA_FLOW_PORT_DEFAULT_MISS_RSS,
    DOCA_FLOW_PORT_DEFAULT_MISS_DROP,
};
```

```
doca_error_t doca_flow_port_update_default_miss(
    struct doca_flow_port *port,
    enum doca_flow_port_default_miss_type default_miss,
    bool counter_en
);

doca_error_t doca_flow_port_query_default_miss(
    struct doca_flow_port *port,
    struct doca_flow_resource_query *data
);
```

- The target of the default miss (`KERNEL`, `RSS`, or `DROP`) can be updated at any time.
- The counter (`counter_en`) can only be enabled *before* the first use of or update to the default miss, and it cannot be disabled once set.
- The `DOCA_FLOW_PORT_DEFAULT_MISS_RSS` option distributes packets over all existing RSS queues, hashing each packet using its L3/L4 information. Packets without a hashable L3/L4 header will receive a constant hash value.

5.4.2.6.4.13 doca_flow_pipe_lpm

The `doca_flow_pipe_lpm` pipe performs Longest Prefix Match (LPM) matching.

LPM Logic

LPM matching is performed on a single field specified at pipe creation, such as the outer destination IP. Each entry consists of a value and a mask (e.g., `10.0.0.0/8`, `10.10.0.0/16`). The LPM match is defined as the entry with the maximum number of matching bits.

For example, given two entries, `10.0.0.0/8` and `10.7.0.0/16`:

- An incoming IP of `10.1.9.2` matches `10.0.0.0/8`.
- An incoming IP of `10.7.9.2` matches `10.7.0.0/16`, as its 16-bit prefix is the longest match.

Exact Match (EM) Logic

In addition to LPM, the pipe can simultaneously apply Exact Match (EM) logic. The EM result is combined with the LPM result using a logical AND. If a packet matches the LPM logic but *fails* the EM logic, it is considered a pipe miss.

EM logic is supported on a combination of the following fields:

- `meta.u32[1]`
- `inner.eth.dst_mac`
- `tun.vxlan_tun_id`

A field (e.g., inner destination MAC) cannot be used for both LPM and EM at the same time.

The primary benefit of EM logic is to allow multiple entries to share the same LPM but have different EM values. For example, an application can have multiple entries for the same IP prefix `1.1.1.1/32` that match different meta values, such as `42`, `555`, and `1020`.

> Support for `meta.u32[1]` allows for EM on any field, as its value can be copied to `meta.u32[1]` in a preceding pipe.

Enabling EM Logic

Enabling EM logic is a two-step process:

1. At pipe creation:
    - Specify the LPM field using the **match** parameter.
    - Enable the EM fields by fully masking them in the **match_mask** parameter (e.g., set **match_mask.meta.u32[1]** to all **0xFF**s, and/or **inner.eth.dst_mac**, and/or **tun.vxlan_tun_id**).
    - If using **tun.vxlan_tun_id**, **match_mask.tun.type** must be set to **DOCA_FLOW_TUN_VXLAN**.
2. At entry addition:
    - Provide the LPM value and its corresponding prefix mask (e.g., **1.1.1.1** and **/32**) in the **match** and **match_mask** structures, respectively.
    - Provide the EM values (e.g., **meta.u32[1] = 42**) in the **match** structure. The **match_mask** is *only* for the LPM field's prefix mask.

General Behavior and Limitations

- Actions and forwarding: The **actions** and **fwd** configurations for an LPM pipe work the same as for a basic DOCA Flow pipe.
- Root pipe: A **doca_flow_pipe_lpm** cannot be a root pipe. A root pipe must be created to forward packets to the LPM pipe.
- LPM field support: LPM matching can only be performed on inner/outer IP addresses and inner/outer MAC addresses.
- Monitoring: The monitor (**doca_flow_monitor**) for an LPM pipe only supports non-shared counters. Other monitoring capabilities are not supported.
- Insertion latency: Entry insertion for an LPM pipe can, in some cases, take milliseconds.
    - Best practice: Entries should be added from a control path rather than a fast data path.
    - Performance: To get the best insertion performance, add entries in large batches.

5.4.2.6.4.14 doca_flow_pipe_acl

doca_flow_pipe_acl uses access-control list (ACL) matching. ACL matching is five tuple of the **doca_flow_match**. Each entry consists of a value and a mask (e.g., 10.0.0.0/8, 10.10.0.0/16, etc.) for IP address fields, port range, or specific port in the port fields, protocol, and priority of the entry.

ACL entry port configuration:

- Mask port is 0 ==> Any port
- Mask port is equal to match port ==> Exact port. Port with mask 0xffff.
- Mask port > match port ==> Match port is used as port from and mask port is used as port to

Monitor actions are not supported in ACL. FWD of the DOCA Flow ACL pipe works the same as the basic DOCA Flow pipe.

ACL supports the following types of FWD:

- **DOCA_FLOW_FWD_PORT**

- **DOCA_FLOW_FWD_PIPE**
- **DOCA_FLOW_FWD_DROP**

doca_flow_pipe_acl insertion max latency can be measured in milliseconds in some cases and, therefore, it is better to insert it from the control path. To get the best insertion performance, entries should be added in large batches.

> ⚠ An ACL pipe is not currently supported for IPv6.

> ⚠ An ACL pipe can be a root pipe.

> ⚠ An ACL pipe can be in ingress and egress domain.

> ⚠ The monitor (**doca_flow_monitor**) for an ACL pipe only supports non-shared counters. Other monitoring capabilities are not supported.

5.4.2.6.4.15 doca_flow_pipe_ordered_list

doca_flow_pipe_ordered_list runs a fixed sequence of actions per entry, so the user can control ordering and reuse the same kind of step multiple times (for example, meter and encap/decap ordering).

An ordered list pipe is defined by an array of actions (i.e., sequences of actions). Each entry can be an instance of one of these sequences.

The pipe is built from one or more list templates (doca_flow_pipe_cfg_set_ordered_lists). Each template is an ordered array of typed elements (enum doca_flow_ordered_list_element_type):

- 'DOCA_FLOW_ORDERED_LIST_ELEMENT_ACTIONS' - a 'struct doca_flow_actions' with optional action descriptors and mask.

-'DOCA_FLOW_ORDERED_LIST_ELEMENT_MONITOR' -a 'struct doca_flow_monitor'

-'DOCA_FLOW_ORDERED_LIST_ELEMENT_NAT64 - a 'struct doca_flow_nat64_action'

-'DOCA_FLOW_ORDERED_LIST_ELEMENT_CRYPTO' - a 'struct doca_flow_crypto'

-'DOCA_FLOW_ORDERED_LIST_ELEMENT_EXTERNAL_ACTIONS' - an external action handle. At pipe creation, the 'external_actions' field points to the pipe-level action handle. At entry creation, it points to the per-entry data structure (e.g. SRH content for SRv6 PUSH).

5.4.2.6.4.16 doca_flow_pipe_hash

The **doca_flow_pipe_hash** API allows for creating a pipe where entries are matched by an index, which is typically the result of a hash calculation on the packet.

Key characteristics of a hash pipe:

- A **doca_flow_match** structure is provided only at pipe creation.

- This `doca_flow_match` mask defines which packet fields are included in the hash calculation.
- Other features (e.g., `monitor`, `actions`, `fwd`) behave identically to a basic DOCA Flow pipe.
- The `nb_flows` (number of entries) defined in `doca_flow_pipe_attr` must be set to a power of 2.

Hash Map Algorithms

The hash pipe's behavior is defined by its map algorithm, which can be set using `doca_flow_pipe_cfg_set_hash_map_algorithm()`.

> If no algorithm is specified, the default `DOCA_FLOW_PIPE_HASH_MAP_ALGORITHM_HASH` is used.

The available algorithms are:

- `DOCA_FLOW_PIPE_HASH_MAP_ALGORITHM_HASH` – The default algorithm. Uses a standard hash function on the masked fields to calculate the index.
- `DOCA_FLOW_PIPE_HASH_MAP_ALGORITHM_RANDOM` – Directs packets to entries randomly. No match mask is required.
- `DOCA_FLOW_PIPE_HASH_MAP_ALGORITHM_IDENTITY` – Uses an identity function for mapping. This creates a one-to-one relationship between the hash value and the index.
- `DOCA_FLOW_PIPE_HASH_MAP_ALGORITHM_FLOODING` – Duplicates the packet to match the number of entries, causing the packet to hit all entries in the pipe.
  - Best practices for this design are shown in the "Flow Hash Flooding Pipe" sample.
  - The pipe is limited to a maximum of 254 entries.
  - This algorithm is not supported in the VNF egress direction.
  - Packet order is only guaranteed for the destination of the first entry.
  - `SECURE_EGRESS` limitation (pre-ConnectX-8) – If a hash pipe is in the `SECURE_EGRESS` domain, its flooding destination pipe cannot be a `SECURE_EGRESS` or `SECURE_INGRESS` pipe.
- `DOCA_FLOW_PIPE_HASH_MAP_ALGORITHM_SELECT_ENABLED` – Consistently directs packets to the same, single index within the pipe.

Dynamic Algorithm Selection

It is possible to specify the hash algorithm dynamically when forwarding a packet from another pipe.

When `DOCA_FLOW_FWD_HASH_PIPE` is used as a forwarding action, the application can specify which algorithm to apply. This allows a single hash pipe to be used in multiple ways, processing packets with different logic as needed.

5.4.2.6.4.17 Pipe Resize

The move to HWS improves performance because rule insertion is implemented in hardware rather than software. However, this move imposes additional limitations, such as the need to commit in advance on the size of the pipes (the number of rule entries). For applications that

require pipe sizes to grow over time, a static size can be challenging: Committing to a pipe size too small can cause the application to fail once the number of rule entries exceeds the committed number, and pre-committing to an excessively high number of rules can result in memory over-allocation.

This is where pipe resizing comes in handy. This feature allows the pipe size to increase during runtime with support for all entries in a new resized pipe.

> Pipe resizing is supported in a basic pipe and a control pipe.

Increasing Pipe Size

It is possible to set a congestion level by percentage (**CONGESTION_PERCENTAGE**). Once the number of entries in the pipe exceeds this value, a callback is invoked. For example, for a pipe with 1000 entries and a **CONGESTION_PERCENTAGE** of 80%, the **CONGESTION_REACHED** callback is invoked after the 800th entry is added.

Following the **CONGESTION_REACHED** callback, the application should call the pipe resize API (**resize()**). The following are optional callbacks during the resize callback:

- A callback on the new number of entries allocated to the pipe
- A callback on each entry that existed in the smaller pipe and is now allocated to the resized pipe

> The pipe pointer remains the same for the application to use even after being resized.

Upon completion of the internal transfer of all entries from the small pipe to the resized pipe, a **RESIZED** callback is invoked.

A **CONGESTION_REACHED** callback is received exactly once before the **RESIZED** callback. Receiving another **CONGESTION_REACHED** only happens after calling **resize()** and receiving its completion with a **RESIZED** callback.

List of Callbacks

- **CONGESTION_REACHED** – on the updated number of entries in the pipe (if pipe is resizable)

  > Receiving a **CONGESTION_REACHED** callback can occur after adding a small number of entries and for moving entries from a small to resized pipe. The application must always call pipe resize after receiving the **CONGESTION_REACHED** callback to handle such cases.

- **RESIZED** – upon completion of the resize operation

  > Calling pipe resize returns immediately. It starts an internal process that ends later with the **RESIZED** callback.

- **NR_ENTRIES_CHANGED** (optional) – on the new max number of entries in the pipe

- **ENTRY_RELOCATE** (optional) – on each entry moved from the small pipe to the resized pipe

Order of Operations for Pipe Resizing

1. Set a process callback on flow configuration:

```
struct doca_flow_cfg *flow_cfg;
doca_flow_cfg_create(&flow_cfg);
doca_flow_cfg_set_cb_pipe_process(flow_cfg, <pipe-process-callback>);
```

> This informs on **OP_CONGESTION_REACHED** and **OP_RESIZED** operations when applicable.

2. Set the following pipe attribute configurations:

```
struct doca_flow_pipe_cfg *pipe_cfg;
doca_flow_pipe_cfg_create(&pipe_cfg, port);
doca_flow_pipe_cfg_set_nr_entries(pipe_cfg, <initial-number-of-entries>);
doca_flow_pipe_cfg_set_is_resizable(pipe_cfg, true);
doca_flow_pipe_cfg_set_congestion_level_threshold(pipe_cfg, <CONGESTION_PERCENTAGE>);
doca_flow_pipe_cfg_set_user_ctx(pipe_cfg, <pipe-user-context>);

# Optional for each pipe queue number to be excluded from entries process:
doca_flow_pipe_cfg_set_excluded_queue(pipe_cfg, pipe_queue);
```

3. Start adding entries:

```
/* Basic pipe */
doca_flow_pipe_basic_add_entry()
/* Control pipe */
doca_flow_pipe_control_add_entry()
```

4. Once the number of entries in the pipe crosses the congestion threshold, an **OP_CONGESTION_REACHED** operation callback is received.
5. Mark the pipe's congestion threshold event and, upon return, call **doca_flow_pipe_resize()**. For this call, add the following parameters:
   - The new threshold percentage for calculating the new size.
   - A callback on the new pipe size (optional):

     ```
     doca_flow_pipe_resize_nr_entries_changed_cb nr_entries_changed_cb
     ```

   - A callback on the entries to be transferred to the resized pipe:

     ```
     doca_flow_pipe_resize_entry_relocate_cb entry_relocation_cb
     ```

6. Call **doca_flow_entries_process()** per queue in a loop to trigger the transfer of entries (per queue). It is relevant for both a basic pipe and a control pipe.
7. At this phase, adding new entries to the pipe is permitted. The entries are added directly to the resized pipe and therefore do not need to be transferred.
8. Once all entries are transferred on all queues, an **OP_RESIZED** operation callback is received (in the context of the last thread calling entries process. Also, at this point a new **OP_CONGESTION_REACHED** operation callback can be received again.

At this point calling **doca_flow_entries_process()** can be stopped for a control pipe. For a basic pipe an additional call is required to complete the call to **doca_flow_pipe_basic_add_entry()**.

doca_flow_entries_process() has the following roles:

- Triggering entry transfer from the smaller to the bigger pipe (until an **OP_RESIZED** callback is received)
- Follow up API on previous **add_entries** API (basic pipe relevance only)
- If a specific queue is excluded on the resized pipe - calling entries process on that queue is not a must in order to receive the **OP_RESIZED** operation callback.
- The **OP_RESIZED** callback can only be received on a pipe non-excluded queue.

5.4.2.6.4.18 Excluded Queues

There are cases where an application uses a subset of queues for rules operations such as add/remove/update. On the other hand, resizing a pipe requires calling doca_flow_entries_process() on all queues to get an **OP_RESIZED** operation callback. This is where the excluded queues feature becomes handy: it enables excluding the pipe's queues where rule operations do not take part of.

- When calling pipe resize it is sufficient to process entries only on the non-excluded queues to get an **OP_RESIZED** operation callback for that pipe. Still - it is allowed to process entries of any queue (for any other pipe with this queue included).
- Before and during pipe destroy it is allowed to process entries (in parallel) on all pipe's excluded queues.

However, there are limitations when configuring a pipe's excluded queues:

- It is not allowed to add/remove/update entries on a pipe's excluded queues. There is no validity check or enforcement on this and it is up to the application to maintain this restriction. It is allowed to process entries of all queues at all times since **doca_flow_entries_process()** API is not per pipe.

Excluded queue syntax

```
struct doca_flow_pipe_cfg *pipe_cfg;

doca_flow_pipe_cfg_create(&pipe_cfg, port);
...
/* The next configuration should be repeated for each excluded queue number individually */
doca_error_t doca_flow_pipe_cfg_set_excluded_queue(struct doca_flow_pipe_cfg *pipe_cfg, uint16_t pipe_queue);

# pipe_cfg - pointer to DOCA Flow pipe configuration struct.
# pipe_queue - the pipe queue number to exclude.
# Return code:
# - DOCA_SUCCESS - in case of success.
# - DOCA_ERROR_INVALID_VALUE - in case of invalid input.
```

- An excluded queue number must be between 0 to (max queues - 1).
- It is not allowed to exclude all queues of a pipe. At least one of the pipe's queues must remain non-excluded.

5.4.2.6.4.19 Hairpin Configuration

In switch mode, if `dev` is set in struct `doca_flow_port_cfg` (using `doca_flow_port_cfg_set_dev()`), then an internal hairpin is created for direct wire-to-wire fwd. Users may specify the hairpin configuration using `mode_args`. The supported options as follows:

- `hairpinq_num=[n]` – the hairpin queue number
- `use_huge_mem` – determines whether the Tx buffer uses hugepage memory
- `lock_rx_mem` – locks Rx queue memory

5.4.2.6.5 Teardown

5.4.2.6.5.1 Pipe Entry Teardown

When an entry is terminated by the user application or ages-out, the user should call the entry destroy function, `doca_flow_pipe_remove_entry()`. This frees the pipe entry and cancels hardware offload.

5.4.2.6.5.2 Pipe Teardown

When a pipe is terminated by the user application, the user should call the pipe destroy function, `doca_flow_pipe_destroy()`. This destroys the pipe and the pipe entries that match it.

When all pipes of a port are terminated by the user application, the user should call the pipe flush function, `doca_flow_port_pipes_flush()`. This destroys all pipes and all pipe entries belonging to this port.

> During `doca_flow_pipe_destroy()` execution, the application must avoid adding/removing entries or checking for aged entries of any other pipes.

5.4.2.6.5.3 Port Teardown

When the port is not used anymore, the user should call the port stop function, `doca_flow_port_stop()`. This stops the DOCA port, disables the traffic, destroys the port and frees all resources of the port.

5.4.2.6.5.4 Flow Teardown

When the DOCA Flow is not used anymore, the user should call the flow destroy function, `doca_flow_destroy()`. This releases all the resources used by DOCA Flow.

5.4.2.7 Metadata

> In DOCA Flow, a scratch area of size up to **DOCA_FLOW_META_MAX** bytes is available throughout the pipeline. This area functions like transient memory, allowing applications to store, modify, and match metadata values across multiple pipes.

You can set metadata values manually or copy them from packet fields. These values can then be matched in later pipes, with support for bit masking in both matching and modification operations.

5.4.2.7.1 Metadata Actions

Metadata can be modified using the following action types depending on their masks or descriptors:

- **SET** – Sets a metadata value from a pipe action or a specific entry action. The width of the value is defined by the action descriptor.
- **COPY** – Copies a value from a packet field, which can include other metadata fields. The width is also specified by the descriptor.

> Refer to DOCA API References for details on **struct doca_flow_meta**.

5.4.2.7.2 Behavior and Usage Considerations

Metadata fields behave similarly to variables in programming languages but have specific characteristics within DOCA Flow:

- Some fields may be internally used by DOCA Flow actions and can be overwritten with opaque values
- Most fields do not retain their values when the packet crosses between NIC and FDB domains (i.e., if there is a transfer between two pipes belonging to the NIC and FDB domains the field data might be lost)
- Some fields support software datapath integration, allowing promotion to or from the software Rx/Tx paths.

5.4.2.7.3 Metadata Field Breakdown

Field	Description
pkt_meta	Intended exclusively for application use. Preserves its content across NIC/FDB domain boundaries. Available in the software Rx datapath. Values set in Tx can be observed in Rx.
u32[0-3]	Always available for application use. Content may not be preserved across NIC/FDB domain boundaries. May be used internally by DOCA Flow for certain pipe types or actions.
u32[4]	Always available for application use

`u32[5-max]`	Optionally available, depending on the hardware model, firmware configuration, and DOCA Flow operation mode
`access_index`	32-bit big-endian field (`doca_be32_t`), set through `actions.meta.access_index` and usable as the target of `SET` , `COPY` , and `ADD` actions. Holds an index used to address external-action resource offset (the actual external-action data is fetched from that memory location using this index). Availability depends on the hardware. On ConnectX-9 this field overlaps the u32[8], u32[9] scratch fields and cannot be used together with them (see the note).

ConnectX-9 limitation: On ConnectX-9, even though access_index is a single 32-bit field, enabling it consumes either u32[8] and/or u32[9], so access_index and u32[8],u32[9] cannot be used simultaneously.

5.4.2.7.4 Pipe Design Consideration

Some pipe types or actions use specific bytes in the scratch area for internal operations. So, if:

1. PIPE-1 sets metadata
2. PIPE-A (intermediate) modifies the same bytes internally
3. PIPE-2 reads those bytes

Then the values may be overwritten by PIPE-A, leading to incorrect results in PIPE-2.

When designing a pipe tree, ensure metadata fields used across multiple pipes are not also internally consumed by intermediate pipes.

Some DOCA pipe types (or actions) use several bytes in the scratch area for internal usage. So, if the user has set these bytes in PIPE-1 and read them in PIPE-2, and between PIPE-1 and PIPE-2 there is PIPE-A which also uses these bytes for internal purposes, then these bytes are overwritten by the PIPE-A. This must be considered when designing the pipe tree.

5.4.2.7.5 Scratch Area Usage by Pipe Type

The following table shows which scratch area bytes (metadata fields) are used internally by different DOCA Flow pipe types.

If the application writes to any of these fields before or after they are used internally by a pipe, the values may be overwritten.

Pipe Type/Action	Elements Used in Scratch
Hash Pipe	`u32[3]` bits [31:16]
LPM Pipe	`u32[0]`
LPM Exact Match Pipe	`u32[0]` , `u32[1]`
ACL	`u32[0]` – `u32[3]`
IPsec Anti-Replay	`u32[0]` bits [7:0], `u32[1]`
METER COLOR	`u32[1]` bits [7:0]

NAT64	`u32[0]` – `u32[2]`

> When designing the pipeline, avoid reading or writing to metadata fields that are used internally by intermediate pipes.

5.4.2.7.6 Supported Metadata Fields per Hardware and Mode

The number of available user metadata fields (starting from `u32[5]`) depends on the hardware generation and operating mode. The following table details which fields are available for application use.

> Attempting to use an unavailable metadata field during pipe creation will cause the operation to fail.

Hardware Type and Mode	Available Fields
Before ConnectX-7, VNF mode, over PF (switchdev mode)	`u32[5]`
Before ConnectX-7, VNF mode, over VF / PF (non-switchdev)	`u32[5]` – `u32[6]`
Before ConnectX-7, SWITCH mode	None
ConnectX-7 and above, VNF mode, over PF (switchdev mode)	`u32[5]` – `u32[8]`
ConnectX-7 and above, VNF mode, over VF / PF (non-switchdev)	`u32[5]` – `u32[9]`
ConnectX-7 and above, SWITCH mode	`u32[5]` – `u32[7]`

Scroll conditional content

ConnectX-7 and above, PRDMA configured	u32[5]

hide

> The number of available fields listed above is the maximum. This number can be reduced by specific host configurations, firmware settings, or other enabled features.
>
> Always verify hardware capabilities and operating mode when designing pipelines that rely on these extended metadata fields. An API to query these capabilities will be provided in a future release.

5.4.2.8 Custom Header

Network protocols evolve rapidly, often outpacing hardware support. To address the need for flexible packet processing, DOCA Flow provides support for custom headers. This allows applications to define and parse proprietary or new network protocols (such as GENEVE or eCPRI) directly in hardware without waiting for native firmware support.

5.4.2.8.1 Network Header Properties

A network header can vary in format, field composition, and length. The Custom Header feature categorizes these properties into the following groups:

- Header length: Indicates the size of the header. It can be a fixed value or dynamic (determined by a specific field within the header). For example, the IPv4 header uses a 4-bit IHL field to indicate its length.
- Next protocol: Defines how the parser identifies the subsequent header. This can be a fixed protocol type or determined dynamically by a dedicated field. For instance, the **Protocol** field in an IPv4 header dictates whether the next header is TCP, UDP, etc.
- Data fields: Specific segments of the header exposed for application logic. Applications can match on or modify these fields.
- TLV options: Describes variable-length options (Type-Length-Value), including overall option length, starting offset, option type, and individual length fields.
- Preceding protocols: Specifies valid preceding headers. For example, a GENEVE header typically follows a UDP header; an eCPRI header may follow either UDP or L2 headers.
- Following protocols: Specifies valid subsequent headers. For example, a tunnel header might be followed by an inner L2 or L3 header.
- Special attributes: Defines protocol characteristics, such as whether the protocol acts as a tunnel (a split point between outer and inner layers).

5.4.2.8.2 IPv4 Header Definition Example

The following table illustrates the structure of an IPv4 header. This layout serves as a reference for defining similar protocols using the Custom Header API.

Byte	0								1								2								3							
Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
IPv4	Version				Header Length				TOS								Total length MSB								Total length LSB							
Offset	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
IPv4	Identification																-	DF	MF	Offset												
Offset	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
IPv4	Time to live								Next Protocol								Header checksum															

Offset	96	97-126	127
IPv4	Source IP address		
Offset	128	129-258	159
IPv4	Destination IP address		

- Header Length: This is a dynamic length determined by the 4-bit `IHL` field (bits 4-7). The value in this field is multiplied by 32 to calculate the total length in bits.
    - Next Protocol: This is determined by the 8-bit **Protocol** field (bits 72-79). This value tells the parser which protocol header follows IPv4 (e.g., TCP, UDP).
    - Data Fields: The application may define fields of interest for matching or modification, such as **Source IP** (32 bits at offset 96) and **Destination IP** (32 bits at offset 128).

5.4.2.8.3 GENEVE Header Definition Example

The following table illustrates the structure of a GENEVE header, emphasizing variable-length options.

Byte	0								1								2								3							
Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Geneve	Version			Options length					O	C	Reserved						Next protocol															
Offset	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
Geneve	Virtual Network Identifier (VNI)																								Reserved							

- Option Length: The 6-bit `Opt Len` field (bits 2-7) specifies the length of the options following the base header. The total length is calculated as `(Opt Len * 32) + 64` bits.
    - Type: Hardware can match on the **Type** field (bits 72-79) to identify specific options, which is necessary because GENEVE options do not have a fixed order.
    - Sample Fields**:** Applications can match or modify specific option data, such as **Option Class** and **Option Value**.

5.4.2.8.4 GENEVE Option Definition Example

The following list describes the properties of a GENEVE header, specifically focusing on how the hardware handles variable-length options.

Byte	0								1								2								3							
Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Option	Option Class																Type								Rsvd				Length			
Offset	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
Option	Option value																															

- Option Length: This is a 5-bit field located at an offset of 27 bits. It specifies the length of the subsequent option data in 32-bit words. The hardware calculates the total header length by multiplying this value by 32 and adding the fixed header size (64 bits).

- Type: Located at an offset of 16 bits with a length of 8 bits. The hardware uses this value to identify specific options and capture the correct data into the configured sampler. This identification is essential because the order of GENEVE options is not fixed by the protocol; the hardware must match the type to locate the data.
- Sample Fields (**Option Class** , **Option Value**): These are the specific data fields the application intends to match or modify. For example, an application might define a sampler for the **Option Value** , located at an offset of 32 bits with a length of 32 bits.

5.4.2.8.5 UDP Header Definition Example

The following table illustrates the structure of a UDP header.

Byte	0								1								2								3							
Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
UDP	Source Port																Destination Port															
Offset	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

UDP	Header + Payload length	Checksum

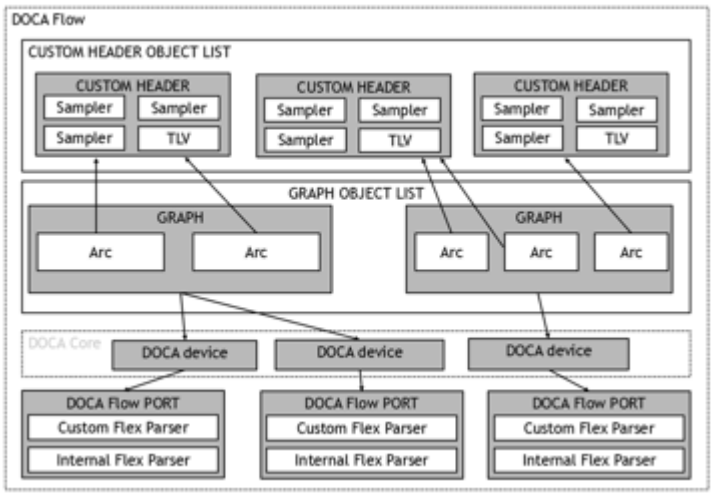
- Source/Destination Port: 16-bit fields typically used for matching or modification.
  - Header Length: Fixed at 64 bits (8 bytes).
  - Next Protocol: The UDP header does not contain an explicit "Next Protocol" field. The payload is treated as generic data unless a specific context (like a tunneling destination port, e.g., VXLAN on port 4789) implicitly defines the next layer.

5.4.2.8.6 Custom Header Lifecycle

The following workflow describes the standard operation for custom headers:

1. Define Structure: The application defines the header layout (fields and lengths) based on the protocol specification.
2. Create Parser Object: The application creates a **doca_flow_parser** object. This object represents the new protocol header and is available for use across the DOCA instance. One object is created per protocol.
3. Create Samplers: The application creates "samplers" within the parser object. These identify specific data fields to be exposed for matching or modification.
4. Set Attributes: The application configures attributes such as header length calculation logic, next protocol identification, and tunnel behavior.
5. Create Parse Graph: The application creates one or more parse graphs. These graphs define the sequence of headers the hardware should expect.
6. Define Arcs: The application creates "arcs" to link native and custom headers within the graph, establishing valid transitions between protocols.
7. Configure Match/Actions: The application uses the **definition** structure to map new header fields to DOCA match and action APIs.
8. Initialize: The application initializes DOCA devices and binds the required parse graphs. Graphs can be bound to multiple devices; all ports on a device (VFs, SFs, representors) share the bound graphs.
9. Teardown: Upon exit, the application must destroy all custom header objects, parse graphs, and devices as part of the standard cleanup procedure.
10. application defines the header structures according to the protocol specification, or just the structure defining the necessary fields only
11. application creates the **custom header** object, which is DOCA global visible and represents the new protocol header applications wish to support. Application can support multiple new protocols and create multiple custom header objects, one per protocol
12. on top of the custom header object application creates the **samplers** - the local objects representing the data field to match on or modify with actions. Sampler can be configured to handle TLV option with a matched type value
13. application sets the custom header object attributes specifying how the header length should be calculated, whether there is a next protocol field, and whether the header is a tunnel header

14. application creates one or multiple **parse graph** objects, that is DOCA global visible object and represents the set of custom headers configured on one or more DOCA devices in hardware
15. on top of large graph application creates an **arc** , the local object to link the native or custom headers in the header parsing chain. It tells the relations between headers and the following order
16. application configures *definitions* to represent the new header fields in DOCA match and actions structures. The required by **definitions** strings are provided from the created samplers
17. application initializes the **DOCA devices** and binds the required parse graphs. The parse graph can be bound to multiple devices, and multiple graphs can be bound to the same device. All network ports spawned on the top of the same device (VFs, SFs, representors) share the bound parse graphs
18. On the DOCA Flow, the library implicitly configures the hardware according to the configuration of the bound parse graphs, some firmware/hardware imposed errors might occur on this stage
19. application uses the match and action structure extensions provided by definitions to build the pipes matching on and modifying the fields in the custom headers
20. on exit application stops the DOCA Flow, destroys the devices, parse graphs, and custom headers – the usual cleanup procedure



5.4.2.8.7 Match and Actions Fields

Custom header fields are presented to DOCA match and action APIs as byte arrays via the **definition** structure. If a field width is not byte-aligned, it is right-aligned within the array, occupying the lower bits.

The **valid_bit** field in the metadata indicates whether the custom header was successfully identified and parsed in the current packet. This ensures matches are only applied when the header is present.

5.4.2.8.8 Known Limitations

There are some known limitations:

- Field alignment: Both the **next protocol** and dynamic **header length** fields must reside entirely within a single 32-bit word, aligned to the start of the header. They cannot cross a dword boundary due to hardware constraints.
- Header length field width: The supported width for a dynamic header length field is limited to 6 or 9 bits, depending on the hardware generation.
- Sampling limit: The total amount of data sampled for matching and modification cannot exceed eight 32-bit words (32 bytes) per header.
- Dummy fields: To maintain the correct header structure in software without exceeding the sampling limit, fields that are not required for matching can be defined as dummy fields in the configuration.
- Multiplier restrictions: The hardware calculates header length in 32-bit words. Therefore, multipliers of 1 (bytes) or 2 (16-bit words) are not fully supported due to hardware alignment behaviors.
- Parse graph vs. flow match: The input and output arcs defined in the parse graph configuration only instruct the hardware on *when* and *how* to capture sample fields from a recognized header. They do not enforce a match in the flow rules. It is the application's responsibility to ensure the flow rules explicitly match the required packet pattern (e.g., ensuring a specific header sequence exists) if strict adherence to the graph structure is required.

5.4.2.8.9 NIC Configuration for Custom Header Support

The Flex Parser is a specialized hardware feature on BlueField DPUs that enables programmable parsing of protocol headers. While the NIC contains fixed, built-in parsers for standard protocols defined at the time of hardware development, the Flex Parser allows DOCA to define and handle new or modified protocols and options.

BlueField-2 and BlueField-3 models support a maximum of 8 concurrent Flex Parser units. However, the actual number available to the application may be lower if the firmware reserves specific units for internal use (e.g., to support newer network protocols).

Two non-volatile parameters control the Flex Parser configuration, which can be modified using the **mlxconfig** utility

- **PROG_PARSE_GRAPH** – Setting this value to **True** makes the Flex Parser feature available for host software usage.
- **FLEX_PARSER_PROFILE_ENABLE** – This setting configures how the firmware allocates internal Flex Parser units for specific protocols.

Configuration steps:

1. Check the current device settings:

```
mlxconfig -d /dev/mst/mt41686_pciconf0 query
```

2. Enable the Flex Parser feature for the host:

```
mlxconfig -d /dev/mst/mt41686_pciconf0 set PROG_PARSE_GRAPH=1
```

3. Set the profile enable bit (e.g., to 4) to allocate resources for the custom header feature.

```
mlxconfig -d /dev/mst/mt41686_pciconf0 set FLEX_PARSER_PROFILE_ENABLE=4
```

4. A cold reboot of the NIC is required for these non-volatile configuration changes to take effect.

```
mlxfwreset -d /dev/mst/mt41686_pciconf0 -yes r
```

5.4.2.9 Segment Routing over IPv6 (SRv6)

DOCA Flow provides hardware-accelerated SRv6 support for IPv6 Segment Routing. SRv6 is implemented as an external action; that is, a plugin that extends the standard DOCA Flow action set.

This feature provides the following capabilities:

- Insert an SRH (Segment Routing Header) after the IPv6 base header with a user-provided segment list.
- Match on the **segments_left** field of an existing SRH.
- Decrement the **segments_left** value to advance to the next segment.
- Copy the active SID (determined by the current **segments_left** value) to the IPv6 destination address.

5.4.2.9.1 SRv6 Prerequisites

SRv6 is supported in both **vnf** and **switch** modes with hardware steering (HWS) enabled. The SRv6 external action must be registered before calling **doca_flow_init()**:

```
doca_flow_external_action_srv6_register();
/* Then proceed with doca_flow_init() */
```

Calling this function after **doca_flow_init()** results in undefined behavior.

5.4.2.9.2 SRv6 Actions Memory

SRv6 actions require additional actions memory beyond the standard per-entry allocation. Each distinct SRH configuration consumes **DOCA_FLOW_SRV6_ACTION_MEM_SIZE** (128) bytes. This must be added to the total actions memory size set via **doca_flow_port_cfg_set_actions_mem_size()**.

For example, for a port with 1024 entries and up to 16 distinct SRH configurations:

```
uint32_t mem_size = ACTIONS_MEM_SIZE(1024) + SRV6_ACTIONS_MEM_SIZE(16);
doca_flow_port_cfg_set_actions_mem_size(port_cfg, mem_size);
```

5.4.2.9.3 SRv6 PUSH Operation

The PUSH operation inserts an SRH immediately after the IPv6 base header. Because this action encapsulates several operations, it is considered a "complex" action. The hardware will:

- Insert a user-inputted SRH (header fields that are not set by HW must contain valid values).
- Set the IPv6 **next_header** field to **43**.
- Set the SRH **next_header** field to the original IPv6 packet's **next_header** value.

To use SRv6 PUSH in a pipe, create a pipe action handle. The configuration specifies the operation type, the port ID, and the expected SRH size in bytes (base header + all segments):

```
#define NB_SIDS 2
struct doca_flow_external_action_srv6_pipe_action_cfg push_cfg = {0};
void *push_action = NULL;

push_cfg.op = DOCA_FLOW_EXT_ACT_SRV6_OP_PUSH;
push_cfg.port_id = port_id;
push_cfg.srh_size = sizeof(struct doca_flow_header_ipv6_srh_base) + NB_SIDS * sizeof(struct doca_flow_ipv6_addr);

result = doca_flow_external_action_srv6_pipe_action_create(&push_cfg, &push_action);
```

When the pipe action is no longer needed, destroy it:

```
doca_flow_external_action_srv6_pipe_action_destroy(push_action);
```

5.4.2.9.3.1 Creating/Destroying the Pipe Action

To utilize the SRv6 PUSH operation within a pipe, the application must first create a pipe action handle. The configuration specifies the operation type, the port ID, and the expected SRH size in bytes (which includes the base header plus all segments):

```
#define NB_SIDS 2
struct doca_flow_external_action_srv6_pipe_action_cfg push_cfg = {0};
void *push_action = NULL;

push_cfg.op = DOCA_FLOW_EXT_ACT_SRV6_OP_PUSH;
push_cfg.port_id = port_id;
push_cfg.srh_size = sizeof(struct doca_flow_header_ipv6_srh_base) + NB_SIDS * sizeof(struct doca_flow_ipv6_addr);

result = doca_flow_external_action_srv6_pipe_action_create(&push_cfg, &push_action);
```

- **srh_size**: Defines the total SRH header size in bytes. All entries added to this pipe action must provide an SRH of this exact size.

This handle is subsequently passed to an ordered list pipe element of type **DOCA_FLOW_ORDERED_LIST_ELEMENT_EXTERNAL_ACTIONS**.

When the pipe action is no longer required, it must be destroyed:

```
doca_flow_external_action_srv6_pipe_action_destroy(push_action);
```

5.4.2.9.3.2 Ordered List Integration

SRv6 PUSH functions as an ordered list pipe external action element. The **DOCA_FLOW_ORDERED_LIST_ELEMENT_EXTERNAL_ACTIONS** element serves a dual role:

- At pipe creation: The **external_actions** field points to the pipe action handle returned by **doca_flow_external_action_srv6_pipe_action_create()**.
- At entry creation: The **external_actions** field points to a **struct doca_flow_external_action_srv6_entry** containing the per-entry SRH data.

The PUSH element can be seamlessly combined with other ordered list elements, such as actions (e.g., copying the SID to the IPv6 destination) and monitors (e.g., counters), within the same list.

5.4.2.9.3.3 Providing Per-Entry SRH Data

Each entry must supply its SRH content via **struct doca_flow_external_action_srv6_entry**. The SRH is defined by **struct doca_flow_header_ipv6_srh**, consisting of a fixed 8-byte base header followed by a flexible array of 128-bit segment addresses:

```
#define NB_SIDS 2
uint32_t entry_len = sizeof(struct doca_flow_external_action_srv6_entry) + NB_SIDS * sizeof(struct
doca_flow_ipv6_addr);
struct doca_flow_external_action_srv6_entry *srv6_entry = calloc(1, entry_len);
struct doca_flow_header_ipv6_srh *srh = &srv6_entry->srh;

srh->base.hdr_ext_len = (NB_SIDS * sizeof(struct doca_flow_ipv6_addr)) / 8;
srh->base.routing_type = 4;
srh->base.segments_left = NB_SIDS - 1;
srh->base.last_entry = NB_SIDS - 1;

/* Segment 0 (active segment — copied to IPv6 DIP) */
SET_IPV6_ADDR(srh->segments[0].addr,
              DOCA_HTOBE32(0x20010db8), DOCA_HTOBE32(0x00000001),
              DOCA_HTOBE32(0x00000000), DOCA_HTOBE32(0x00000001));

/* Segment 1 */
SET_IPV6_ADDR(srh->segments[1].addr,
              DOCA_HTOBE32(0x20010db8), DOCA_HTOBE32(0x00000002),
              DOCA_HTOBE32(0x00000000), DOCA_HTOBE32(0x00000002));
```

- **hdr_ext_len**: Must be non-zero (at least one segment is required) and must be an even number, as each segment occupies 16 bytes (2 units of 8). DOCA Flow does not validate this field; the application must ensure it correctly reflects the total SRH size.
- **next_header** automatic handling: When a PUSH operation executes, DOCA Flow automatically saves the original IPv6 **next_header** value before overwriting it with **43** (IPv6 Routing Header). After the SRH is inserted and the packet is reparsed, the saved value is restored into the SRH **next_header** field. Any manual application settings applied to **srh->base.next_header** are ignored.

5.4.2.9.3.4 Matching/Actions on SRv6 Fields

DOCA Flow supports matching and modifying the following SRv6 fields:

- **segments_left**: The number of remaining route segments (8-bit).
- **segments**: The active segment identifier (128-bit IPv6 address).

These fields are accessed through the DOCA Flow Definitions API. Developers must register the SRv6 fields prior to calling **doca_flow_init()** by extending the match and actions structures:

```
struct srv6_match {
    struct doca_flow_match base;
    uint32_t segments_left;
};

struct srv6_actions {
    struct doca_flow_actions base;
    uint32_t segments_left;
};
```

```
/* Register match field */
doca_flow_definitions_add_field(definitions,
                                "match.packet.outer.srv6.segments_left",
                                offsetof(struct srv6_match, segments_left),
                                sizeof(((struct srv6_match *)0)->segments_left));

/* Register actions field */
doca_flow_definitions_add_field(definitions,
                                "actions.packet.outer.srv6.segments_left",
                                offsetof(struct srv6_actions, segments_left),
                                sizeof(((struct srv6_actions *)0)->segments_left));
```

The **base** member must be the first field in the extended structure.

SRv6 Field Strings

The following field strings are available for use in action descriptors:

String Field	Supported Actions	Description
outer.srv6.segments_left	Add, Copy (src/dst)	Segments remaining count
outer.srv6.segments	Copy (src)	Active segment (128-bit SID)

Examples:

- Copying the active SID into the IPv6 destination address:

```
desc.type = DOCA_FLOW_ACTION_COPY;
desc.field_op.src.field_string = "outer.srv6.segments";
desc.field_op.src.bit_offset = 0;
desc.field_op.dst.field_string = "outer.ipv6.dst_ip";
desc.field_op.dst.bit_offset = 0;
desc.field_op.width = 128;
```

- Decrementing **segments_left**: Use the **ADD** action with a per-entry value of **UINT8_MAX** (which is the equivalent of subtracting 1 in unsigned 8-bit arithmetic).

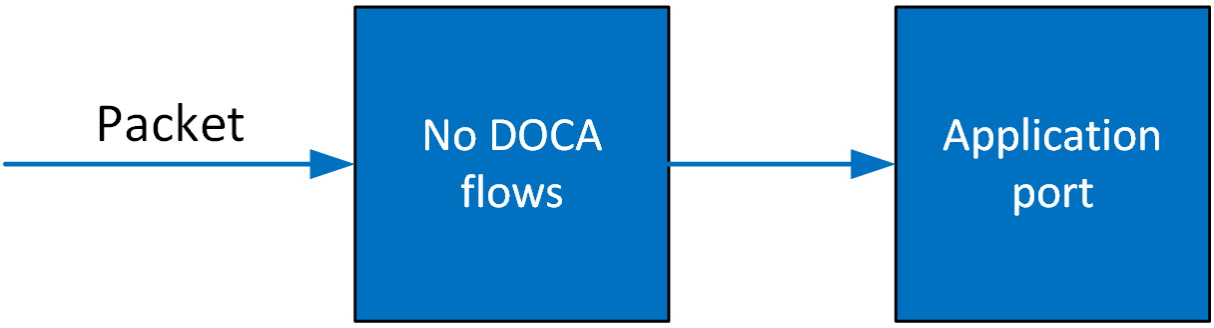
```
desc.type = DOCA_FLOW_ACTION_ADD;
desc.field_op.dst.field_string = "outer.srv6.segments_left";
desc.field_op.dst.bit_offset = 0;
desc.field_op.width = 8;

/* At entry creation: */
entry_actions.segments_left = UINT8_MAX; /* decrement by 1 */
```

5.4.2.9.3.5 SRv6 PUSH Limitations

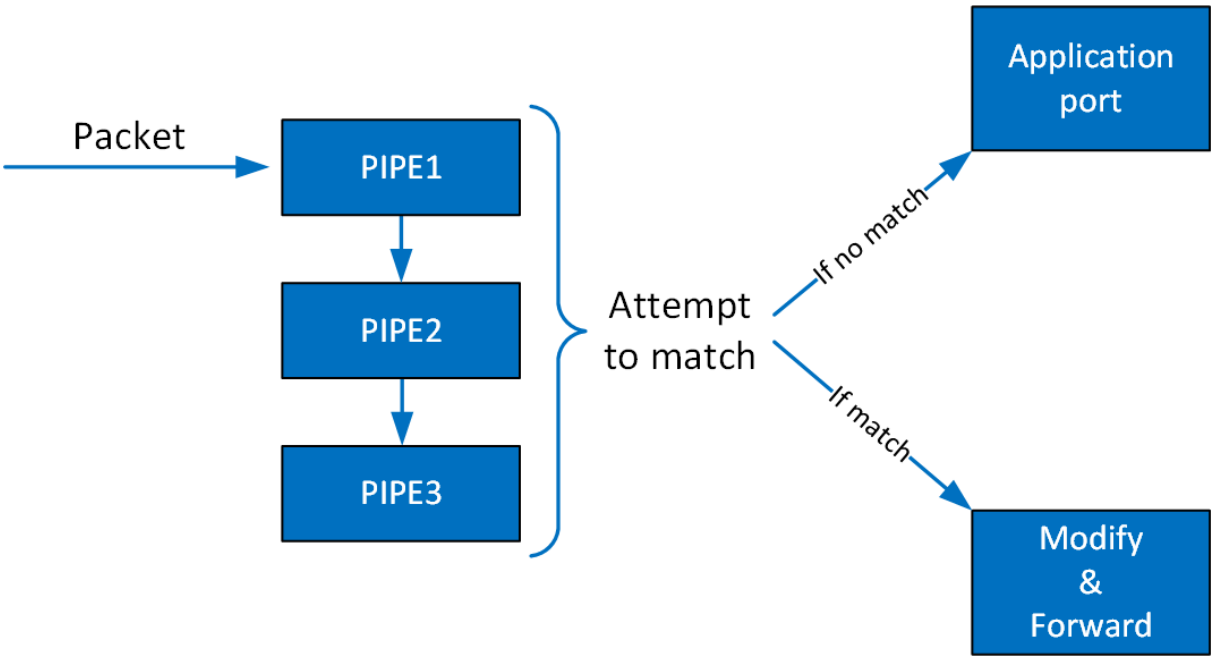
- Currently, only the PUSH operation is supported in hardware. SRH removal (POP) must be performed in software (future releases will introduce hardware POP support).
- The total SRH size (base header + segment list) must not exceed the hardware's insert-header maximum. The API will return a descriptive error message in such cases.
- **doca_flow_external_action_srv6_register()** MUST be called before **doca_flow_init()**.
- SRv6 PUSH strictly requires an internal scratch register on the port. If this register is unavailable (which is dependent on the specific hardware model and firmware configuration), the pipe creation will fail.

5.4.2.10 Packet Processing

In situations where there is a port without a pipe defined, or with a pipe defined but without any entry, the default behavior is that all packets arrive to a port in the software.



Once entries are added to the pipe, if a packet has no match then it continues to the port in the software. If it is matched, then the rules defined in the pipe are executed.



If the packet is forwarded in RSS, the packet is forwarded to software according to the RSS definition. If the packet is forwarded to a port, the packet is redirected back to the wire. If the packet is forwarded to the next pipe, then the software attempts to match it with the next pipe.

Note that the number of pipes impacts performance. The longer the number of matches and actions that the packet goes through, the longer it takes the hardware to process it. When there is a very large number of entries, the hardware must access the main memory to retrieve the entry context which increases latency.

5.4.2.11 Debug and Trace Features

DOCA Flow supports trace and debugging of DOCA Flow applications which enable collecting predefined internal key performance indicators (KPIs) and pipeline visualization.

5.4.2.11.1 Installation

The set of DOCA's SDK development packages include also a developer-oriented package that includes additional trace and debug features which are not included in the production libraries:

- **.deb** based systems – **libdoca-sdk-flow-trace**
- **.rpm** based systems – **doca-sdk-flow-trace**

These packages install the trace-version of the libraries under the following directories:

- **.deb** based systems – **/opt/mellanox/doca/lib/<arch>/trace**
- **.rpm** based systems – **/opt/mellanox/doca/lib64/trace**

5.4.2.11.2 Using Trace Libraries

5.4.2.11.2.1 Runtime Linking

The trace libraries are designed to allow a user to link their existing (production) program to the trace library without needing to recompile the program. To do so, one should simply update the matching environment variable so that the OS will prioritize loading libraries from the above trace directory.

The following is an example for such an update for the Ubuntu 22.04 BlueField image:

```
LD_LIBRARY_PATH=/opt/mellanox/doca/lib/aarch64-linux-gnu/trace:${LD_LIBRARY_PATH} doca_ipsec_security_gw <program
parameters>
```

5.4.2.11.2.2 Compilation

The trace-level development packages in the previous section provide additional compilation definitions (**doca-flow-trace.pc**) to be used in addition to the regular compilation definitions for the DOCA Flow SDK library (**doca-flow.pc**). It is recommended to use these compilation definitions for the following scenarios:

- Static linking of the trace-level DOCA Flow library into your program
- Regular (dynamic) linking of the trace-level DOCA Flow library into your program during development and testing

Although the latter could also be determined at runtime as explained in the previous section, many developers find it handy to compile directly against the trace version during initial development phases.

5.4.2.11.3 Trace Features

5.4.2.11.3.1 DOCA Log – Trace Level

DOCA's trace logging level (**DOCA_LOG_LEVEL_TRACE**) is compiled as part of this trace version of the library. That is, any program compiled against the library can activate this additional logging level through DOCA's API or even through DOCA's built-in argument parsing (ARGP) library:

```
LD_LIBRARY_PATH=/opt/mellanox/doca/lib/aarch64-linux-gnu/trace:${LD_LIBRARY_PATH} doca_ipsec_security_gw <program
parameters> --sdk-log-level 70
```

5.4.2.11.3.2 DOCA Flow – Additional Sanity Checks

When using the trace version of the library, additional input sanitation checks are added, at the cost of introducing minor performance implications. These checks are meant to assist developers in their early steps of using the library, as they provide early detection and improved logging for common coding mistakes.

5.4.2.12 DOCA Flow Samples

This section provides DOCA Flow sample implementation on top of the BlueField.

> All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

5.4.2.12.1 Sample Prerequisites

A DOCA Flow-based program can either run on the host machine or on the BlueField.

Flow-based programs require an allocation of huge pages, hence the following commands are required:

```
$ echo '1024' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
$ sudo mkdir /mnt/huge
$ sudo mount -t hugetlbfs -o pagesize=2M nodev /mnt/huge
```

5.4.2.12.2 Running the Sample

1. Refer to the following documents:
   - DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - DOCA Troubleshooting for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

   ```
   cd /opt/mellanox/doca/samples/doca_flow/<sample_name>
   meson /tmp/build
   ninja -C /tmp/build
   ```

   > The binary **doca_<sample_name>** will be created under **/tmp/build/``.**

3. Sample (e.g., **flow_aging**) usage:

   ```
   Usage: doca_flow_aging [DPDK Flags] -- [DOCA Flags]

   DOCA Flags:
     -h, --help                          Print a help synopsis
     -v, --version                       Print program version information
     -l, --log-level                     Set the (numeric) log level for the program <10=DISABLE,
   20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
   ```

```
  --sdk-log-level                         Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                       Parse command line flags from an input json file
```

4. For additional information per sample, use the `-h` option after the `--` separator:

```
/tmp/build/doca_<sample_name> -- -h
```

5. The following is an example from an execution on the DPU:
   - CLI example for running the VNF samples:

```
/tmp/build/doca_<sample_name> -- -a aux/2,dv_flow_en=2 -a aux/3,dv_flow_en=2 -l 60
```

> The `dv_flow_en=2` device argument puts the DPDK PMD into hardware steering mode, which DOCA Flow relies on when the port's Rx/Tx datapath is managed through DPDK. Supply it for each device probed via DPDK.

   - CLI example for running the switch samples:

```
/tmp/build/doca_<sample_name> -- -r pci/03:00.0,pf0sf[2-3] -l 60
```

> When running on the BlueField in switch mode, it is not necessary to configure the OVS.
>
> The DOCA switch sample hides the extra `dv_flow_en=2,fdb_def_rule_en=0,dv_xmeta_en=4` DPDK device arguments behind a simple `-r` to specify the PCIe ID and representor information.

> When running on the DPU using the command above, sub-functions must be enabled according to the BlueField Scalable Functions.

> When running on the host, virtual functions must be used according to the instructions in the DOCA Virtual Functions.

> For additional information regarding the command line syntax of the device and device representor identifiers, please refer to the matching section in the DOCA Arg Parser Guide.

5.4.2.13 Field String Mapping

5.4.2.13.1 Field String Supported Actions

The following is a list of all the API fields available for matching criteria and action execution.

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
meta.data (bit_offset < 32)	meta.pkt_meta	meta.pkt_meta	✔	✔	✔	✔	✔	✔	✔
meta.data (bit_offset ≥ 32)	meta.u32[i]	meta.u32[i]	✔	✔	✔	✔	✔	✔	✔
parser_meta.hash.result	None. See section "Copy Hash Result" for details.		N/A	N/A	✔	N/A	✔	✘	✘
parser_meta.port.id	parser_meta.port_id		N/A	N/A	✘	N/A	✘	✘	✘
parser_meta.ipsec.syndrome	parser_meta.ipsec_syndrome		N/A	N/A	✘	N/A	✘	✘	✘
parser_meta.psp.syndrome	parser_meta.psp_syndrome		N/A	N/A	✘	N/A	✘	✘	✘
parser_meta.random.value	parser_meta.random		N/A	N/A	✘	N/A	✘	✔	✘
parser_meta.meter.color	parser_meta.meter_color		N/A	N/A	✘	N/A	✘	✘	✘
parser_meta.packet_type.l2_outer	parser_meta.outer_l2_type		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.packet_type.l3_outer	parser_meta.outer_l3_type		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.packet_type.l4_outer	parser_meta.outer_l4_type		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.packet_type.l2_inner	parser_meta.inner_l2_type		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.packet_type.l3_inner	parser_meta.inner_l3_type		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.packet_type.l4_inner	parser_meta.inner_l4_type		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.outer_ip_fragmented.flag	parser_meta.outer_ip_fragmented		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.inner_ip_fragmented.flag	parser_meta.inner_ip_fragmented		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.outer_integrity.l3_ok	parser_meta.outer_l3_ok		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.outer_integrity.ipv4_checksum_ok	parser_meta.outer_ip4_checksum_ok		N/A	N/A	N/A	N/A	N/A	N/A	N/A

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
parser_meta.outer_integrity.l4_ok	parser_meta.outer_l4_ok		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.outer_integrity.l4_checksum_ok	parser_meta.outer_l4_checksum_ok		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.inner_integrity.l3_ok	parser_meta.inner_l3_ok		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.inner_integrity.ipv4_checksum_ok	parser_meta.inner_ip4_checksum_ok		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.inner_integrity.l4_ok	parser_meta.inner_l4_ok		N/A	N/A	N/A	N/A	N/A	N/A	N/A
parser_meta.inner_integrity.l4_checksum_ok	parser_meta.inner_l4_checksum_ok		N/A	N/A	N/A	N/A	N/A	N/A	N/A
outer.eth.dst_mac	outer.eth.dst_mac	outer.eth.dst_mac	✔	✘	✔	✔	✔	✘	✘
outer.eth.src_mac	outer.eth.src_mac	outer.eth.src_mac	✔	✘	✔	✔	✔	✘	✘
outer.eth.type	outer.eth.type	outer.eth.type	✔	✘	✔	✔	✔	✘	✘
outer.eth_vlan0.tci	outer.eth_vlan[0].tci	outer.eth_vlan[0].tci	✔	✘	✔	✔	✔	✘	✘
outer.eth_vlan1.tci	outer.eth_vlan[1].tci	outer.eth_vlan[1].tci	✘	✘	✘	✘	✘	✘	✘
outer.ipv4.src_ip	outer.ip4.src_ip	outer.ip4.src_ip	✔	✘	✔	✔	✔	✘	✘
outer.ipv4.dst_ip	outer.ip4.dst_ip	outer.ip4.dst_ip	✔	✘	✔	✔	✔	✘	✘
outer.ipv4.dscp_ecn	outer.ip4.dscp_ecn	outer.ip4.dscp_ecn	✔	✘	✔	✔	✔	✘	✘
outer.ipv4.next_proto	outer.ip4.next_proto	outer.ip4.next_proto	✔	✘	✔	✔	✔	✘	✘
outer.ipv4.ttl	outer.ip4.ttl	outer.ip4.ttl	✔	✔	✔	✔	✔	✘	✘
outer.ipv4.version_ihl	outer.ip4.version_ihl	outer.ip4.version_ihl	✔	✔	✔	✔	✔	✘	✘
outer.ipv4.total_len	outer.ip4.total_len	outer.ip4.total_len	✔	✔	✔	✔	✔	✘	✘

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
outer.ipv4.identification	outer.ip4.identification	outer.ip4.identification	✘	✘	✘	✘	✘	✘	✘
outer.ipv4.flags_fragment_offset	outer.ip4.flags_fragment_offset	outer.ip4.flags_fragment_offset	✘	✘	✘	✘	✘	✘	✘
outer.ipv6.src_ip	outer.ip6.src_ip	outer.ip6.src_ip	✔	✘	✔	✔	✔	✘	✘
outer.ipv6.dst_ip	outer.ip6.dst_ip	outer.ip6.dst_ip	✔	✘	✔	✔	✔	✘	✘
outer.ipv6.traffic_class	outer.ip6.traffic_class	outer.ip6.traffic_class	✔	✘	✔	✔	✔	✘	✘
outer.ipv6.flow_label	outer.ip6.flow_label	outer.ip6.flow_label	✔	✘	✔	✔	✔	✘	✘
outer.ipv6.next_proto	outer.ip6.next_proto	outer.ip6.next_proto	✔	✘	✔	✔	✔	✘	✘
outer.ipv6.hop_limit	outer.ip6.hop_limit	outer.ip6.hop_limit	✔	✔	✔	✔	✔	✘	✘
outer.ipv6.payload_len	outer.ip6.payload_len	outer.ip6.payload_len	✔	✔	✔	✔	✔	✘	✘
outer.udp.src_port	outer.udp.l4_port.src_port	outer.udp.l4_port.src_port	✔	✘	✔	✔	✔	✘	✘
outer.udp.dst_port	outer.udp.l4_port.dst_port	outer.udp.l4_port.dst_port	✔	✘	✔	✔	✔	✘	✘
outer.transport.src_port	outer.transport.src_port	outer.transport.src_port	✔	✘	✔	✔	✔	✘	✘
outer.transport.dst_port	outer.transport.dst_port	outer.transport.dst_port	✔	✘	✔	✔	✔	✘	✘
outer.tcp.src_port	outer.tcp.l4_port.src_port	outer.tcp.l4_port.src_port	✔	✘	✔	✔	✔	✘	✘
outer.tcp.dst_port	outer.tcp.l4_port.dst_port	outer.tcp.l4_port.dst_port	✔	✘	✔	✔	✔	✘	✘
outer.tcp.flags	outer.tcp.flags	outer.tcp.flags	✘	✘	✘	✘	✘	✘	✘
outer.tcp.data_offset	outer.tcp.data_offset	outer.tcp.data_offset	✘	✔	✔	✔	✔	✘	✘
outer.tcp.seq_num	outer.tcp.seq_num	outer.tcp.seq_num	✔	✔	✔	✘	✘	✔	✔

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
outer.tcp.ack_num	outer.tcp.ack_num	outer.tcp.ack_num	✔	✔	✔	✘	✘	✔	✔
outer.icmp4.type	outer.icmp.type	outer.icmp.type	✘	✘	✘	✘	✘	✘	✘
outer.icmp4.code	outer.icmp.code	outer.icmp.code	✘	✘	✘	✘	✘	✘	✘
outer.icmp4.ident	outer.icmp.ident	outer.icmp.ident	✘	✘	✘	✘	✘	✘	✘
outer.icmp6.type	outer.icmp.type	outer.icmp.type	✘	✘	✘	✘	✘	✘	✘
outer.icmp6.code	outer.icmp.code	outer.icmp.code	✘	✘	✘	✘	✘	✘	✘
outer.srv6.segments_left	outer.srv6.segments_left	outer.srv6.segments_left	✘	✔	✔	✔	✔	✔	✔
outer.srv6.segments		outer.srv6.segments	✘	✘	✘	✔	✔	✘	✘
tunnel.gre.protocol	tun.protocol	tun.protocol	✘	✘	✘	✘	✘	✘	✘
tunnel.gre_key.value	tun.gre_key	tun.gre_key	✘	✘	✘	✘	✘	✘	✘
tunnel.nvgre.protocol	tun.protocol	tun.protocol	✘	✘	✘	✘	✘	✘	✘
tunnel.nvgre.nvgre_vs_id	tun.nvgre_vs_id	tun.nvgre_vs_id	✘	✘	✘	✔	✔	✘	✘
tunnel.nvgre.nvgre_flow_id	tun.nvgre_flow_id	tun.nvgre_flow_id	✘	✘	✘	✘	✘	✘	✘
tunnel.vxlan.vni	tun.vxlan_tun_id	tun.vxlan_tun_id	✔	✘	✔	✔	✔	✘	✘
tunnel.vxlan_gpe.vni	tun.vxlan_tun_id	tun.vxlan_tun_id	✔	✘	✔	✔	✔	✘	✘
tunnel.vxlan_gbp.vni	tun.vxlan_tun_id	tun.vxlan_tun_id	✔	✘	✔	✔	✔	✘	✘
tunnel.vxlan_gpe.next_proto	tun.vxlan_gpe_next_protocol		✘	✘	✘	✘	✘	✘	✘
tunnel.vxlan_gpe.flags	tun.vxlan_gpe_flags		✘	✘	✘	✘	✘	✘	✘
tunnel.vxlan_gbp.policy_id	tun.vxlan_gbp_group_policy_id		✘	✘	✘	✘	✘	✘	✘
tunnel.vxlan.rsvd1		tun.vxlan_tun_rsvd1	✔	✘	✘	✔	✔	✘	✘
tunnel.vxlan_gpe.rsvd1		tun.vxlan_tun_rsvd1	✔	✘	✘	✔	✔	✘	✘

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
tunnel.vxlan_gbp.rsvd1		tun.vxlan_tun_rsvd1	✔	✘	✘	✔	✔	✘	✘
tunnel.gtp.teid	tun.gtp_teid	tun.gtp_teid	✔	✘	✔	✔	✔	✘	✘
tunnel.gtp_ext_hdr.next_ext	tun.gtp_next_ext_hdr_type	tun.gtp_next_ext_hdr_type	N/A	N/A	N/A	N/A	N/A	N/A	N/A
tunnel.gtp_psc.qfi	tun.gtp_ext_psc_qfi	tun.gtp_ext_psc_qfi	✔	✘	✔	✔	✔	✘	✘
tunnel.esp.spi	tun.esp_spi	tun.esp_spi	✔	✘	✔	✔	✔	✘	✘
tunnel.esp.sn	tun.esp_sn	tun.esp_sn	✔	✘	✔	✔	✔	✔	✔
tunnel.psp.nexthdr	tun.psp.nexthdr	tun.psp.nexthdr	✔	✘	✔	✔	✔	✘	✘
tunnel.psp.hdrextlen	tun.psp.hdrextlen	tun.psp.hdrextlen	✔	✘	✔	✔	✔	✘	✘
tunnel.psp.res_cryptofst	tun.psp.res_cryptofst	tun.psp.res_cryptofst	✔	✘	✔	✔	✔	✘	✘
tunnel.psp.s_d_ver_v	tun.psp.s_d_ver_v	tun.psp.s_d_ver_v	✔	✘	✔	✔	✔	✘	✘
tunnel.psp.spi	tun.psp.spi	tun.psp.spi	✔	✘	✔	✔	✔	✘	✘
tunnel.psp.iv	tun.psp.iv	tun.psp.iv	✔	✘	✔	✔	✔	✘	✘
tunnel.psp.vc	tun.psp.vc	tun.psp.vc	✔	✘	✔	✔	✔	✘	✘
tunnel.mpls[0].label	tun.mpls[0].label	tun.mpls[0].label	✘	✘	✔	✘	✔	✘	✘
tunnel.mpls[1].label	tun.mpls[1].label	tun.mpls[1].label	✘	✘	✔	✘	✔	✘	✘
tunnel.mpls[2].label	tun.mpls[2].label	tun.mpls[2].label	✘	✘	✔	✘	✔	✘	✘
tunnel.mpls[3].label	tun.mpls[3].label	tun.mpls[3].label	✘	✘	✔	✘	✔	✘	✘
tunnel.mpls[4].label	tun.mpls[4].label	tun.mpls[4].label	✘	✘	✔	✘	✔	✘	✘
tunnel.geneve.ver_opt_len	tun.geneve.ver_opt_len	tun.geneve.ver_opt_len	✘	✘	✘	✘	✘	✘	✘
tunnel.geneve.o_c	tun.geneve.o_c	tun.geneve.o_c	✘	✘	✘	✘	✘	✘	✘

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
tunnel.geneve.next_proto	tun.geneve.next_proto	tun.geneve.next_proto	✘	✘	✘	✘	✘	✘	✘
tunnel.geneve.vni	tun.geneve.vni	tun.geneve.vni	✔	✘	✔	✔	✔	✘	✘
tunnel.geneve_opt[i].type	None. See section "Copy Geneve Options" for details.		✔	✘	✔	✔	✔	✘	✘
tunnel.geneve_opt[i].class			✔	✘	✔	✔	✔	✘	✘
tunnel.geneve_opt[i].data			✔	✘	✔	✔	✔	✘	✘
inner.eth.dst_mac	inner.eth.dst_mac		✘	✘	✔	✘	✔	✘	✘
inner.eth.src_mac	inner.eth.src_mac		✘	✘	✔	✘	✔	✘	✘
inner.eth.type	inner.eth.type		✘	✘	✔	✘	✔	✘	✘
inner.eth_vlan0.tci	inner.eth_vlan[0].tci		✘	✘	✘	✘	✘	✘	✘
inner.eth_vlan1.tci	inner.eth_vlan[1].tci		✘	✘	✘	✘	✘	✘	✘
inner.ipv4.src_ip	inner.ip4.src_ip		✘	✘	✔	✘	✔	✘	✘
inner.ipv4.dst_ip	inner.ip4.dst_ip		✘	✘	✔	✘	✔	✘	✘
inner.ipv4.dscp_ecn	inner.ip4.dscp_ecn		✘	✘	✔	✘	✔	✘	✘
inner.ipv4.next_proto	inner.ip4.next_proto		✘	✘	✘	✘	✘	✘	✘
inner.ipv4.ttl	inner.ip4.ttl		✘	✘	✔	✘	✔	✘	✘
inner.ipv4.version_ihl	inner.ip4.version_ihl		✘	✘	✔	✘	✔	✘	✘
inner.ipv4.total_len	inner.ip4.total_len		✘	✘	✔	✘	✔	✘	✘
inner.ipv4.identification	inner.ip4.identification		✘	✘	✘	✘	✘	✘	✘
inner.ipv4.flags_fragment_offset	inner.ip4.flags_fragment_offset		✘	✘	✘	✘	✘	✘	✘
inner.ipv6.src_ip	inner.ip6.src_ip		✘	✘	✔	✘	✔	✘	✘
inner.ipv6.dst_ip	inner.ip6.dst_ip		✘	✘	✔	✘	✔	✘	✘
inner.ipv6.traffic_class	inner.ip6.traffic_class		✘	✘	✔	✘	✔	✘	✘
inner.ipv6.flow_label	inner.ip6.flow_label		✘	✘	✔	✘	✔	✘	✘
inner.ipv6.next_proto	inner.ip6.next_proto		✘	✘	✘	✘	✘	✘	✘
inner.ipv6.hop_limit	inner.ip6.hop_limit		✘	✘	✔	✘	✔	✘	✘
inner.ipv6.payload_len	inner.ip6.payload_len		✘	✘	✔	✘	✔	✘	✘

String Field	Path in The Structure		Set	Add		Copy		Condition	
	Match	Actions		Dst	Src	Dst	Src	A	B
inner.udp.src_port	inner.udp.l4_port.src_port		✘	✘	✔	✘	✔	✘	✘
inner.udp.dst_port	inner.udp.l4_port.dst_port		✘	✘	✔	✘	✔	✘	✘
inner.transport.src_port	inner.transport.src_port		✘	✘	✔	✘	✔	✘	✘
inner.transport.dst_port	inner.transport.dst_port		✘	✘	✔	✘	✔	✘	✘
inner.tcp.src_port	inner.tcp.l4_port.src_port		✘	✘	✔	✘	✔	✘	✘
inner.tcp.dst_port	inner.tcp.l4_port.dst_port		✘	✘	✔	✘	✔	✘	✘
inner.tcp.flags	inner.tcp.flags		✘	✘	✘	✘	✘	✘	✘
inner.tcp.data_offset	inner.tcp.data_offset		✘	✘	✔	✘	✔	✘	✘
inner.tcp.seq_num	inner.tcp.seq_num		✘	✘	✘	✘	✘	✘	✘
inner.tcp.ack_num	inner.tcp.ack_num		✘	✘	✘	✘	✘	✘	✘
inner.icmp4.type	inner.icmp.type		✘	✘	✘	✘	✘	✘	✘
inner.icmp4.code	inner.icmp.code		✘	✘	✘	✘	✘	✘	✘
inner.icmp4.ident	inner.icmp.ident		✘	✘	✘	✘	✘	✘	✘
inner.icmp6.type	inner.icmp.type		✘	✘	✘	✘	✘	✘	✘
inner.icmp6.code	inner.icmp.code		✘	✘	✘	✘	✘	✘	✘

We don't have a way to export this macro.

5.4.2.13.2 Non-Matchable Field Strings

Users can modify fields which are not included in `doca_flow_match` structure.

5.4.2.13.2.1 Copy Hash Result

You can copy the the matcher hash calculation into other fields using the `"parser_meta.hash.result"` string.

5.4.2.13.2.2 Copy UTC Timestamp Value

You can copy the NIC hardware timestamp value into other fields using `"parser_meta.utc.time"` string.

5.4.2.13.2.3 Copy GENEVE Options

User can copy GENEVE option type/class/data using the following strings:

- `"tunnel.geneve_opt[i].type"` – Copy from/to option type (only for option configured with `DOCA_FLOW_PARSER_GENEVE_OPT_MODE_MATCHABLE`).
- `"tunnel.geneve_opt[i].class"` – Copy from/to option class (only for option configured with `DOCA_FLOW_PARSER_GENEVE_OPT_MODE_MATCHABLE`).
- `"tunnel.geneve_opt[i].data"` – Copy from/to option data, the bit offset is from the start of the data.

`i` is the index of the option in `tlv_list` array provided in `doca_flow_parser_geneve_opt_create`

5.4.2.14 DOCA Flow Connection Tracking

This guide provides an overview and configuration instructions for the DOCA Flow Connection Tracking (CT) API.

5.4.2.14.1 Introduction

The DOCA Flow Connection Tracking (CT) module is a 5-tuple table designed to efficiently track network connections using hardware resources. It supports the following key features:

- Track zone and 5-tuple sessions – Track and manage network connections based on a 5-tuple (source IP, destination IP, source port, destination port, and protocol) along with zone-based separation
- IPv4 and IPv6 sessions – Track connections for both IP versions; each direction can independently use IPv4 or IPv6
- Zone-based virtual tables – Enable logical isolation using zones
- Aging support – Remove idle connections automatically using configurable timeouts
- Connection metadata – Set and manage metadata for tracked connections
- Bidirectional packet handling – Manage traffic in both directions of a connection
- High connection rate – Efficiently handle a high rate of connections per second (CPS)

The CT module makes it simple and efficient to track connections by leveraging hardware resources.

5.4.2.14.2 Architecture

The DOCA Flow CT Pipe is designed to handle non-encapsulated TCP and UDP packets. It supports two primary actions:

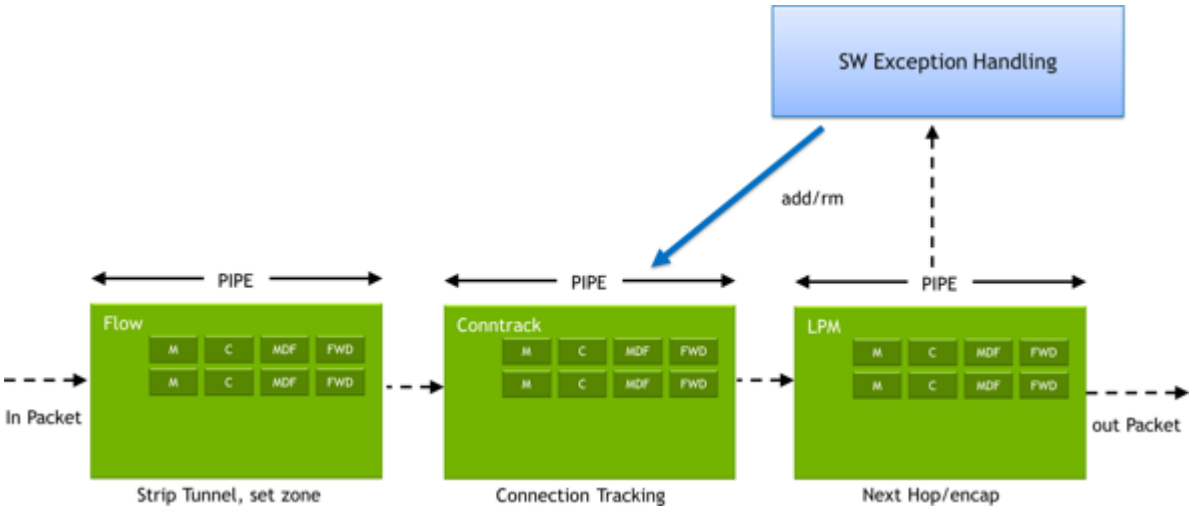
- Forward to next pipe – For packets that match a known 6-tuple connection (5-tuple + zone)
- Miss to next pipe – For packets without a matching connection entry

The application is responsible for handling packets based on these outcomes.

The DOCA Flow CT API consists of four major components:

- CT module manipulation – Configure and manage resources within the CT module
- CT connection entry manipulation – Add, remove, or update connection entries efficiently
- Callbacks – Handle asynchronous processing results for connection entries
- Pipe and entry statistics – Monitor connection tracking performance using pipe-level and entry-level statistics

These components provide flexible control over connection tracking and monitoring, allowing applications to adapt to various network scenarios effectively.



5.4.2.14.2.1 Aging

Aging time refers to the maximum duration (in seconds) a session can remain active without detecting any packets. If no packets are observed within this period, the session is terminated.

To support aging, aging checks run on the counter service thread, which polls and checks counters for all active connections each cycle, ensuring that stale sessions are removed efficiently.

> ⚠ When aging is enabled, either the **counter** flag or a non-zero **timeout** must be set for at least one connection entry to trigger session expiration.

5.4.2.14.2.2 Managed Mode

In Managed Mode, the application is responsible for:

- Managing worker threads
- Parsing and handling connection lifecycles

This mode utilizes DOCA Flow CT management APIs for creating and destroying connections.

The CT aging module automatically notifies the application of aged-out connections by invoking callbacks.

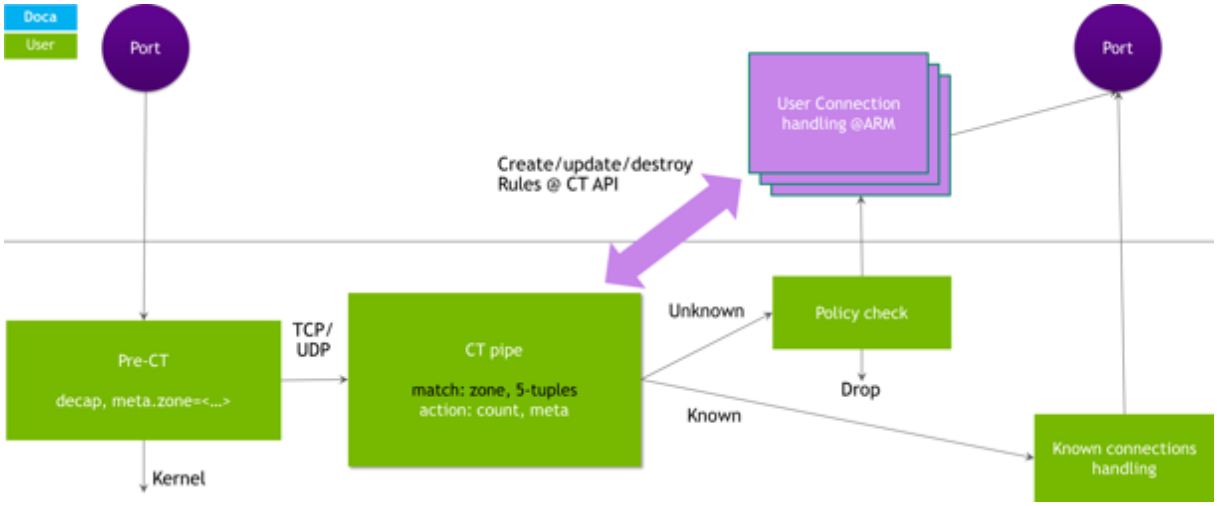
5.4.2.14.2.3 Connection Rules and Management

Users have the flexibility to create connection rules with different patterns, metadata (meta), or counters which can be applied separately for each packet direction.

> ⚠ Users must manually define the appropriate meta and mask values for matching (**match**) and modifying (**modify**) packets.

To create rules in stages:

1. Create one rule for a connection using the standard API.
2. Add a second rule for the opposite packet direction using the **doca_flow_ct_entry_add_dir()** API.



5.4.2.14.2.4 Processing CT Entries

DOCA Flow provides specialized APIs to process CT entries using a dedicated queue:

- **doca_flow_entries_process** – Processes pipe entries in the queue
- **doca_flow_aging_handle** – Handles the aging of pipe entries

> ℹ Some APIs, such as CT entry status queries and pipe miss queries, are not supported in Managed Mode.

5.4.2.14.3 CT Pipe Configuration

CT pipes extend standard pipe creation with additional optional **doca_flow_pipe_cfg** parameters:

Function	Description
doca_flow_pipe_cfg_set_ct_connections()	Sets the maximum number of IPv4, IPv6, and total CT connections.
doca_flow_pipe_cfg_set_ct_max_connections_per_zone()	Sets the maximum CT connections permitted per zone.
doca_flow_pipe_cfg_set_ct_dup_filter_size()	Sets the number of connections to cache in the duplication filter.

5.4.2.14.4 IPv6 Support

DOCA Flow CT tracks IPv6 connections as well as IPv4, including asymmetric flows where origin and reply use different IP versions (e.g. IPv4 origin, IPv6 reply). IPv4 and IPv6 sessions use separate pools (sized via **doca_flow_pipe_cfg_set_ct_connections()**); per entry, the IPv6 origin/reply flags in **doca_flow_ct.h** select which direction matches on IPv6.

CT IPv6 steering is hardware-dependent - supported on BlueField-3 / ConnectX-8 and above (BlueField-2 / ConnectX-7 are IPv4-only). Query support before allocating IPv6 sessions; if unsupported, set the IPv6 session count to 0 in **doca_flow_pipe_cfg_set_ct_connections()**.

5.4.2.14.4.1 Querying CT IPv6 Support

Use the port capabilities API (**doca_flow_port_capabilities_create** / **_get_ct_ipv6** / **_destroy**) to check device support before configuring CT. Perform the create -> query -> destroy cycle after **doca_flow_init()** and before port start (see Port Capabilities in the DOCA Flow programming guide). These APIs are **DOCA_EXPERIMENTAL** - build with **-DDOCA_ALLOW_EXPERIMENTAL_API**.

See the **doca_flow_ct_tcp_ipv4_ipv6** sample for an end-to-end example, including the capability query.

5.4.2.14.5 Prerequisites

5.4.2.14.5.1 DPU

To enable DOCA Flow CT on the DPU, perform the following on the Arm:

1. Enable **iommu.passthrough** in Linux boot commands (or disable SMMU from the DPU BIOS):
   a. Run:

   ```
   sudo vim /etc/default/grub
   ```

   Set **GRUB_CMDLINE_LINUX="iommu.passthrough=1"**.
   b. Run:

```
sudo update-grub
sudo reboot
```

2. Configure DPU firmware with **LAG_RESOURCE_ALLOCATION=1** :

```
sudo mlxconfig -d <device-id> s LAG_RESOURCE_ALLOCATION=1
```

> Retrieve **device-id** from the output of the **mst status -v** command. If, under the MST tab, the value is N/A, run the **mst start** command.

3. Update **/etc/mellanox/mlnx-bf.conf** as follows:

```
ALLOW_SHARED_RQ="no"
```

4. Perform power cycle on the host and Arm sides.
5. If working with a single port, set the DPU into e-switch mode:

```
sudo devlink dev eswitch set pci/<pcie-address> mode switchdev
sudo devlink dev param set pci/<pcie-address> name esw_multiport value false cmode runtime
```

> Retrieve **pcie-address** from the output of the **mst status -v** command.

6. If working with two PF ports, set the DPU into multi-port e-switch mode (for the 2 PCIe devices):

```
sudo devlink dev param set pci/<pcie-address> name esw_multiport value true cmode runtime
```

> Retrieve **pcie-address** from the output of the **mst status -v** command.

Define huge pages (see DOCA Flow prerequisites).

5.4.2.14.5.2 ConnectX

To enable DOCA Flow CT on the NVIDIA® ConnectX®, perform the following:

1. Configure firmware with **LAG_RESOURCE_ALLOCATION=1** :

```
sudo mlxconfig -d <device-id> s LAG_RESOURCE_ALLOCATION=1
```

> Retrieve **device-id** from the output of the **mst status -v** command. If, under the MST tab, the value is N/A, run the **mst start** command.

2. Perform power cycle.
3. If working with a single port:

```
sudo devlink dev eswitch set pci/<pcie-address> mode switchdev
sudo devlink dev param set pci/<pcie-address> name esw_multiport value false cmode runtime
```

Retrieve **pcie-address** from the output of the **mst status -v** command.

4. If working with two PF ports:

```
sudo devlink dev eswitch set pci/<pcie-address0> mode switchdev
sudo devlink dev eswitch set pci/<pcie-address1> mode switchdev
sudo devlink dev param set pci/<pcie-address0> name esw_multiport value true cmode runtime
sudo devlink dev param set pci/<pcie-address1> name esw_multiport value true cmode runtime
```

Retrieve **pcie-address** from the output of the **mst status -v** command.

Define huge pages (see DOCA Flow prerequisites).

5.4.2.14.6 Actions

DOCA Flow CT supports actions based on meta and NAT operations. Each action can be defined as either shared or non-shared.

Action descriptors are not supported.

5.4.2.14.6.1 Shared Actions

Actions that can be shared between entries. Shared actions are predefined and reused in multiple entries.

The user gets a handle per shared action created and uses this handle as a reference to the action where required.

It is user responsibility to track shared actions and to remove them when they become irrelevant.

Shared actions are defined using a control queue (see struct doca_flow_ct_cfg).

5.4.2.14.6.2 Non-shared Actions

Actions provided with their data during entry create/update.

These actions are completely managed by DOCA Flow CT and cannot be reused in multiple flows (i.e., NAT operations).

5.4.2.14.6.3 Action Sets in Pipe Creation

When creating a DOCA Flow CT pipe, users must define action sets, just as they would for any other pipe.

Fields in the CT pipe must be marked as **CHANGEABLE** during pipe creation. This allows the actual criteria for these fields to be specified later during entry creation.

Only actions related to meta and NAT, as defined in struct doca_flow_ct_actions, are supported.

During entry creation or update, different actions can be specified for each direction, allowing variations in action content and/or action type.

5.4.2.14.6.4 Feature Enable

To enable user actions, administrators must configure the following parameters:

- User action templates must be configured during the DOCA Flow CT pipe creation phase. The maximum memory allocated for user actions (**actions_mem_size**) must be defined during DOCA Flow CT initialization.

5.4.2.14.6.5 Using Actions in Managed Mode

Init

When calling **doca_flow_ct_init()** , you must configure the following parameters:

- **nb_ctrl_queues** : The total number of control queues dedicated to defining shared actions.
- **actions_mem_size** : The maximum amount of memory (in bytes) allocated for user actions. This value must be strictly 64-byte aligned, and NVIDIA highly recommends utilizing a power of 2.

Create DOCA Flow CT Pipe

Configure actions sets on **doca_flow_pipe_create()** .

Create Shared Actions

Use **doca_flow_ct_actions_add_shared()** with one of the control queues.

Shared actions can be added at any time before use.

Add Entry

Entry can be created in one of the following ways:

- Using an action handle of a predefined shared actionUsing action data, which is specific to the flow, not sharable (e.g., for NAT operations)

The entry can have different actions and/or different action types per direction.

Remove Entry

Non-shared actions associated with an entry are implicitly destroyed by DOCA Flow CT.

Shared actions are not destroyed. They can be used by the user until they decide to remove them.

Update Entry

Entry actions can be updated per direction. All combinations of shared/non-shared actions are applicable (e.g., update from shared to non-shared).

5.4.2.14.7 Changeable Forward

DOCA Flow CT permits the use of a different forward pipe for each flow direction. The module operates at one of two mutually exclusive forwarding levels:

- Pipe level – A single forward pipe is defined during DOCA Flow CT pipe creation and applies to all entries universally.
- Entry level – The forward pipe is defined dynamically during entry creation.

Entry-level forwarding characteristics:

- It exclusively supports **DOCA_FLOW_FWD_PIPE** and **DOCA_FLOW_FWD_ORDERED_LIST_PIPE** (fixed pipe, changeable index).
- It supports defining a distinct forward pipe per flow direction (both directions can utilize the same or different forward pipes).
- Because there is no default forward pipe, forwarding destinations must be explicitly set upon each entry creation.

5.4.2.14.7.1 Enabling Changeable Forwarding

To enable this feature, create the DOCA Flow CT pipe using one of the following configurations:

- Standard pipe:
  - Set forward type to **DOCA_FLOW_FWD_PIPE**
  - Set **next_pipe** to **NULL**
- Ordered list pipe:
  - Set forward type to **DOCA_FLOW_FWD_ORDERED_LIST_PIPE**
  - Set **ordered_list_pipe.pipe** to **<ol_pipe>**
  - Set **ordered_list_pipe.idx** to **UINT32_MAX**

5.4.2.14.7.2 Using Changeable Forward in Managed Mode

To utilize changeable forwarding in Managed Mode, execute the following sequence:

1. Initialize CT by calling **doca_flow_ct_init()**.
2. Create pipe by calling **doca_flow_pipe_create()** using the changeable forwarding configurations described above.
3. Add entry by calling **doca_flow_ct_add_entry()**. During this step, set **fwd_origin** and/or **fwd_reply** to your desired targets.
4. Update entry by calling **doca_flow_ct_update_entry()** to update the forwarding for a specific entry direction.

When updating the forward destination, you must explicitly pass all other parameters with their previously existing values.

5.4.2.14.8 Entry Iterator

When iterator support is enabled, DOCA Flow CT can traverse all entries on a CT pipe using a registered callback. For each invocation, the application retrieves full entry data (match, hash, flags) via **doca_flow_ct_get_entry()** and can recreate or mirror those entries.

A primary use case for this is High-Availability (HA) Synchronization: the application reads every active entry from the CT on the active node and programs matching entries on the standby node to preserve connection state during failovers. Iteration is incremental; the application drives progress by calling **doca_flow_ct_entries_process()** per queue, and the registered callback executes as entries are dispatched.

5.4.2.14.8.1 Enabling Iterator

1. Create or configure the CT with the **DOCA_FLOW_CT_FLAG_ITERATOR** flag.

   > The CT duplication filter is backed by a hash table of active entries to prevent duplicate insertions while iteration and forwarding run concurrently.

2. Start pipe-level iteration by calling **doca_flow_ct_pipe_iterate(ct_pipe, iterate_cb, iterate_usr_ctx)**. This schedules the iteration across all queues for the given pipe.
3. For each CT queue, call **doca_flow_ct_entries_process()** and pass the **max_processed_entries** limit. This processes the hardware steering queue and invokes **iterate_cb** as entries are delivered.
   - Inside the callback, read the entry details via **doca_flow_ct_get_entry()** to obtain the matcher, hash, and flags for standby replication.
   - If the number of processed entries returned is less than the requested **max_processed_entries**, the iteration for that specific queue has reached its end for the current pass.
4. Pipe iteration formally completes once all participating queues have finished the incremental processing steps and no further callbacks are pending.

5.4.2.14.8.2 Iterator Limitations

- Action exclusion: Entry actions are not exported through the iterator path. The application must manually retain or reconstruct CT entry actions on the standby node.
- Incomplete passes: New entries created during or immediately after a walk are not guaranteed to be captured in a single iterator pass. Applications should track new entries independently and not rely solely on the iterator for complete HA synchronization.

5.4.2.14.9 API

For the library API reference, refer to DOCA Flow and CT API documentation in the NVIDIA DOCA Library APIs.

> DOCA Flow CT is in the DOCA Flow library.

The following sections provide additional details about the library API.

5.4.2.14.9.1 enum doca_flow_ct_flags

Optional DOCA Flow CT configuration flags.

Flag	Description
`DOCA_FLOW_CT_FLAG_STATS`	Enables internal pipe counters for packet tracking. Call `doca_flow_pipe_dump(<ct_pipe>)` to dump the changed counter values.
`DOCA_FLOW_CT_FLAG_WORKER_STATS`	Enables the periodic dump of worker thread internal debug counters.
`DOCA_FLOW_CT_FLAG_NO_AGING`	Disables aging.
`DOCA_FLOW_CT_FLAG_ASYMMETRIC_TUNNEL`	Allows utilizing tunnel or non-tunnel configurations in different directions.
`DOCA_FLOW_CT_FLAG_NO_COUNTER`	Disables counters and aging entirely to save aging-thread CPU cycles.
`DOCA_FLOW_CT_FLAG_ITERATOR`	Enables the entry iterator.
`DOCA_FLOW_CT_FLAG_DUP_FILTER_UDP_ONLY`	Applies the connection duplication filter strictly for UDP connections.
`DOCA_FLOW_CT_FLAG_ORIGIN_WIRE`	Indicates origin traffic will arrive from the wire. If set, mark actions can be utilized in the origin direction.
`DOCA_FLOW_CT_FLAG_REPLY_WIRE`	Indicates reply traffic will arrive from the wire. If set, mark actions can be utilized in the reply direction.

5.4.2.14.9.2 enum doca_flow_ct doca_flow_ct_entry_flags

Optional DOCA Flow CT entry flags.

Flag	Description
`DOCA_FLOW_CT_ENTRY_FLAGS_NO_WAIT = (1 << 0)`	Entry is not buffered; send to hardware immediately
`DOCA_FLOW_CT_ENTRY_FLAGS_DIR_ORIGIN = (1 << 1)`	Apply flags to origin direction
`DOCA_FLOW_CT_ENTRY_FLAGS_DIR_REPLY = (1 << 2)`	Apply flags to reply direction
`DOCA_FLOW_CT_ENTRY_FLAGS_IPV6_ORIGIN = (1 << 3)`	Origin direction is IPv6; origin match union in struct `doca_flow_ct_match` is IPv6
`DOCA_FLOW_CT_ENTRY_FLAGS_IPV6_REPLY = (1 << 4)`	Reply direction is IPv6; reply match union in struct `doca_flow_ct_match` is IPv6

DOCA_FLOW_CT_ENTRY_FLAGS_COUNTER_ORIGIN = (1 << 5)	Apply counter to origin direction
DOCA_FLOW_CT_ENTRY_FLAGS_COUNTER_REPLY = (1 << 6)	Apply counter to reply direction
DOCA_FLOW_CT_ENTRY_FLAGS_COUNTER_SHARED = (1 << 7)	Counter is shared for both direction (origin and reply)
DOCA_FLOW_CT_ENTRY_FLAGS_FLOW_LOG = (1 << 8)	Enable flow log on entry removed
DOCA_FLOW_CT_ENTRY_FLAGS_ALLOC_ON_MISS = (1 << 9)	Allocate on entry not found when calling **doca_flow_ct_entry_prepare()** API
DOCA_FLOW_CT_ENTRY_FLAGS_DUP_FILTER_ORIGIN = (1 << 10)	Enable duplication filter on origin direction
DOCA_FLOW_CT_ENTRY_FLAGS_DUP_FILTER_REPLY = (1 << 11)	Enable duplication filter on reply direction

5.4.2.14.9.3 enum doca_flow_ct_rule_opr

Options for handling flows in autonomous mode with shared actions. The decision is taken on the first flow packet.

Operation	Description
DOCA_FLOW_CT_RULE_OK	Flow should be defined in the CT pipe using the required shared actions handles
DOCA_FLOW_CT_RULE_DROP	Flow should not be defined in the CT pipe. The packet should be dropped.
DOCA_FLOW_CT_RULE_TX_ONLY	Flow should not be defined in the CT pipe. The packet should be transmitted.

5.4.2.14.9.4 struct direction_cfg

Managed mode configuration for origin or reply direction.

Field	Description
bool match_inner	5-tuple match pattern applies to packet inner layer
struct doca_flow_meta *zone_match_mask	Mask to indicate meta field and bits to match
struct doca_flow_meta *meta_modify_mask	Mask to indicate meta field and bits to modify on connection packet match

5.4.2.14.9.5 doca_flow_ct_cfg

DOCA Flow CT configuration lifecycle manipulation:

```
struct doca_flow_ct_cfg *ct_cfg;
ret = doca_flow_ct_cfg_create(&ct_cfg);
doca_flow_ct_cfg_set_flags(ct_cfg, flags);
doca_flow_ct_cfg_set_queues(ct_cfg, n_queues);
/* ... */
ret = doca_flow_ct_init(ct_cfg);

final:
ret = doca_flow_ct_cfg_destroy(ct_cfg);
```

Configuration API methods:

Function	Description
doca_flow_ct_cfg_create	Creates the CT configuration object.
doca_flow_ct_cfg_destroy	Destroys the CT configuration object.
doca_flow_ct_cfg_set_flags	Sets the CT flags (refer to **enum doca_flow_ct_flags**).
doca_flow_ct_cfg_set_queues	Sets the number of hardware queues utilized to manipulate connections.
doca_flow_ct_cfg_set_queue_depth	Sets the queue depth (defaults to 512 rules).
doca_flow_ct_cfg_set_ctrl_queues	Sets the number of CT control queues used for defining shared actions.
doca_flow_ct_cfg_set_actions_mem_size	Sets the total CT actions memory size in bytes.
doca_flow_ct_cfg_set_entry_private_data_size	Sets the size of user private data allocated per connection.
doca_flow_ct_cfg_set_entry_finalize_cb	Sets the entry finalize callback to query final connection statistics.
doca_flow_ct_cfg_set_status_update_cb	Sets the status update callback to notify the application of counter changes.
doca_flow_ct_cfg_set_aging_core	Deprecated (removal planned); use **doca_flow_port_cfg_set_service_threads_core**
doca_flow_ct_cfg_set_aging_query_delay	Sets the CT aging query delay for newly created connections.
doca_flow_ct_cfg_set_aging_plugin_ops	Defines custom aging logic callbacks (falls back to default logic if omitted).
doca_flow_ct_cfg_set_direction	Configures the origin and reply directions.

Additional configuration notes:

- CT pipe session limits and duplication filter size are configured via **doca_flow_pipe_cfg** during pipe creation (see CT Pipe Configuration).

- CT counter configuration: DOCA Flow must be configured in per-port mode using **doca_flow_cfg_set_resource_mode(cfg, DOCA_FLOW_RESOURCE_MODE_PORT)**. Define the number of CT counters via **doca_flow_port_cfg_set_nr_resources(port_cfg, DOCA_FLOW_RESOURCE_COUNTER_CT, <n>)**.

5.4.2.14.9.6 struct doca_flow_ct_actions

This structure is used in the following cases:

- For defining shared actions. In this case, action data is provided by the user. The action handle is returned by DOCA Flow CT.For defining an entry with actions. The structure can be filled with two options:
  - With action handle of a previously created shared actionWith non-shared action data

DOCA Flow CT action structure.

```
enum doca_flow_resource_type  resource_type;
union {
                /* Used when creating an entry with a shared action. */
                uint32_t action_handle;

                /* Used when creating an entry with non-shared action or when creating a shared action. */
                struct {
                            uint32_t action_idx;
                            struct doca_flow_meta meta;
                            struct doca_flow_header_l4_port l4_port;
                            union {
                                      struct doca_flow_ct_ip4 ip4;
                                      struct doca_flow_ct_ip6 ip6;
                             };
                } data;
       };
```

Where:

Field	Description
enum doca_flow_resource_type resource_type	Shared/non-shared action
uint32_t action_handle	Shared action handle
uint32_t action_idx	Actions template index
struct doca_flow_meta meta	Modify meta values
struct doca_flow_header_l4_port l4_port	UDP or TCP source and destination port
struct doca_flow_ct_ip4 ip4	Source and destination IPv4 addresses
struct doca_flow_ct_ip6 ip6	Source and destination IPv6 addresses

The value in **meta**, **l4_port**, **ip4**, and **ip6** should start from **bit0**, the least significant bit, regardless of which bits are set in mask. For example, **action_val.meta.u32[0] = DOCA_HTOBE32(0x12)**, **action_mask.meta.u32[0] = DOCA_HTOBE32(0x0000FF00)** sets bits 15-8 to **0x12**.

5.4.2.14.10 DOCA Flow Connection Tracking Samples

This section describes DOCA Flow CT samples based on the DOCA Flow CT pipe.

The samples illustrate how to use the library API to manage TCP/UDP connections.

> All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

5.4.2.14.10.1 Running the Samples

1. Refer to the following documents:
   - DOCA Installation Guide for Linux for details on how to install BlueField-related software.NVIDIA BlueField Platform Software Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

   ```
   cd /opt/mellanox/doca/samples/doca_flow/flow_ct_udp
   meson /tmp/build
   ninja -C /tmp/build
   ```

   > The binary **doca_flow_ct_udp** is created under **/tmp/build/samples/** .

3. Sample (e.g., **doca_flow_ct_udp**) usage:

   ```
   Usage: doca_<sample_name> [DOCA Flags] [Program Flags]

   DOCA Flags:
     -h, --help                            Print a help synopsis
     -v, --version                         Print program version information
     -l, --log-level                       Set the (numeric) log level for the program <10=DISABLE,
   20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
     --sdk-log-level                       Set the SDK (numeric) log level for the program <10=DISABLE,
   20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
     -j, --json <path>                     Parse command line flags from an input json file

   Program Flags:
     -p, --pci_addr <PCI-ADDRESS>          PCIe device address
   ```

4. For additional information per sample, use the **-h** option:

   ```
   /tmp/build/samples/<sample_name> -h
   ```

The following is a CLI example for running the samples when port **08:00.0** is configured (multi-port e-switch) as manager port:

```
/tmp/build/samples/doca_<sample_name> -- -r pci/08:00.0,pf0vf0 -l 60
```

The following is a CLI example for running the samples when port **08:00.0** is configured (multi-port e-switch) as manager port and **08:00.1** is configured as the representor of the second uplink:

```
/tmp/build/samples/doca_<sample_name> -- -r pci/08:00.1 -l 60
```

To avoid the test being impacted by unexpected packets, it only accepts packets like the following examples:

- IPv4 destination address is `1.1.1.1` IPv6 destination address is `0101:0101:0101:0101:0101:0101:0101:0101`

5.4.2.14.10.2 Samples List

Sample Name	Description
`doca_flow_ct_2_ports`	Deploys two independent e-switches, each maintaining its own distinct CT state and pipeline.
`doca_flow_ct_aging`	Demonstrates CT aging using a pipe with entries that feature variable aging times and custom user data.
`doca_flow_ct_iterator`	Iterates through the CT pipe across two standalone e-switches.
`doca_flow_ct_tcp`	Utilizes CT in conjunction with TCP flags for robust session handling.
`doca_flow_ct_tcp_actions`	Attaches both shared and non-shared actions to a TCP CT implementation.
`doca_flow_ct_tcp_entry_finalize`	Leverages the CT entry finalize callback when sessions terminate or are manually removed.
`doca_flow_ct_tcp_ipv4_ipv6`	Handles complex flows where each packet direction utilizes a different IP version.
`doca_flow_ct_tcp_wire_to_wire`	Executes a mark action on the CT for a strictly wire-to-wire TCP path.
`doca_flow_ct_udp`	Deploys a basic UDP pipeline that natively incorporates a CT pipe.
`doca_flow_ct_udp_query`	Queries the Flow CT UDP session state based on the origin or reply direction.
`doca_flow_ct_udp_single_match`	Creates a hardware CT entry applying a single-direction match within `doca_flow_ct_add_entry()`.
`doca_flow_ct_udp_tunnel_asymmetric`	Implements an asymmetric tunnel mode for Flow CT (an extension of the core UDP query sample).
`doca_flow_ct_udp_update`	Dynamically updates CT entries post-creation, allowing inactive UDP sessions to receive updated aging timeouts.

5.4.2.15 DOCA Flow Tune Server

This guide provides an overview and configuration instructions for DOCA Flow Tune Server API.

5.4.2.15.1 Introduction

DOCA Flow Tune Server (TS) is a DOCA Flow subcomponent that collects predefined internal key performance indicators (KPIs) and pipeline information from a running DOCA Flow application. All information is transferred via an inter-process communication channel (Unix domain socket) to the DOCA Flow Tune Tool for further analysis and monitoring.

TS can run in either two modes:

1. Basic: TS uses an internal thread to handle the communication channel.
2. Expert: Server socket polling and thread management are the user's responsibility. The TS API must be invoked whenever the communication channel socket is ready to process a message.

5.4.2.15.2 Prerequisites

DOCA Flow Tune Server API is only available when using the DOCA Flow and DOCA Flow Tune Server trace libraries.

> For more detailed information, refer to section "Debug and Trace Features" under DOCA Flow.

5.4.2.15.3 Configuration

DOCA Flow Tune Server has a configuration file that allows customizing various settings. The configuration file is divided into different sections so to ease its use.

5.4.2.15.3.1 Config File Default Values

If a configuration file was not provided, DOCA Flow Tune Server uses default values for its mandatory fields. List of all default values can be seen in section "Configuration File Example".

5.4.2.15.3.2 Custom Config File

Instead of using the default configuration values, users may create a file of their own and provide a file path using the **doca_flow_tune_server_cfg_set_cfg_file_path()** API call.

Once used, DOCA Flow Tune Server loads all provided values directly from the file, while the rest of the fields (if any) use their respective default values.

5.4.2.15.3.3 Configuration File Example

flow_tune_cfg.json

```
{
    "network": {
        "uds_path": "/tmp/tune_server.sock"
    }
}
```

- **network**

- **uds_path** – Unix Domain Socket (**AF_UNIX**) path for the tune server to bind to. This socket is used for the inter-process-communication (IPC) channel between DOCA Flow Tune Server and DOCA Flow Tune Tool. Default value is **/tmp/tune_server.sock** .

5.4.2.15.4 API

For more detailed information on DOCA Flow API, refer to *NVIDIA DOCA Library APIs*.

The following subsections provide additional details about the library API.

5.4.2.15.4.1 enum doca_flow_tune_server_mode

Enum to be used to set the running mode.

5.4.2.15.4.2 struct doca_flow_tune_server_cfg

Opaque configuration struct to use on configuration API calls.

5.4.2.15.4.3 doca_flow_tune_server_cfg_create

Allocates and creates DOCA Flow Tune Server configuration structure.

```
doca_error_t doca_flow_tune_server_cfg_create(struct doca_flow_tune_server_cfg **cfg);
```

5.4.2.15.4.4 doca_flow_tune_server_cfg_set_cfg_file_path

Sets the local configuration file path in the opaque configuration struct, for DOCA Flow Tune Server to use when searching for the JSON configuration file.

Providing a JSON configuration file is optional. If a file is not provided, DOCA Flow Tune Server uses internal defaults.

```
doca_error_t doca_flow_tune_server_cfg_set_bind_path(struct doca_flow_tune_server_cfg *cfg, const char *path);
```

5.4.2.15.4.5 doca_flow_tune_server_cfg_set_mode

Sets the running mode in the opaque configuration struct.

Using this API call is optional. Without the call, DOCA Flow Tune Server runs in Basic mode.

```
doca_error_t doca_flow_tune_server_cfg_set_mode(struct doca_flow_tune_server_cfg *cfg, enum
doca_flow_tune_server_mode mode);
```

5.4.2.15.4.6 doca_flow_tune_server_cfg_set_fd

Sets the user Unix Domain Socket in the opaque configuration struct. This socket is used by DOCA Flow Tune Server to receive messages on the communication channel.

Using this API call is mandatory on Expert mode.

```
doca_error_t doca_flow_tune_server_cfg_set_fd(struct doca_flow_tune_server_cfg *cfg, int fd);
```

5.4.2.15.4.7 doca_flow_tune_server_cfg_destroy

Destroys and deallocates DOCA Flow Tune Server opaque configuration structure.

Should be called after calling **doca_flow_tune_server_init()**.

```
doca_error_t doca_flow_tune_server_cfg_destroy(struct doca_flow_tune_server_cfg *cfg);
```

5.4.2.15.4.8 doca_flow_tune_server_init

Starts DOCA Flow Tune Server main thread.

```
doca_error_t doca_flow_tune_server_init(struct doca_flow_tune_server_cfg *cfg);
```

5.4.2.15.4.9 doca_flow_tune_server_destroy

Stops DOCA Flow Tune Server main thread.

```
void doca_flow_tune_server_destroy(void);
```

5.4.2.15.4.10 doca_flow_tune_server_process_msg

Process communication channel message from the provided Unix Domain Socket. This API call should be used with Expert mode only.

```
doca_error_t doca_flow_tune_server_process_msg(void);
```

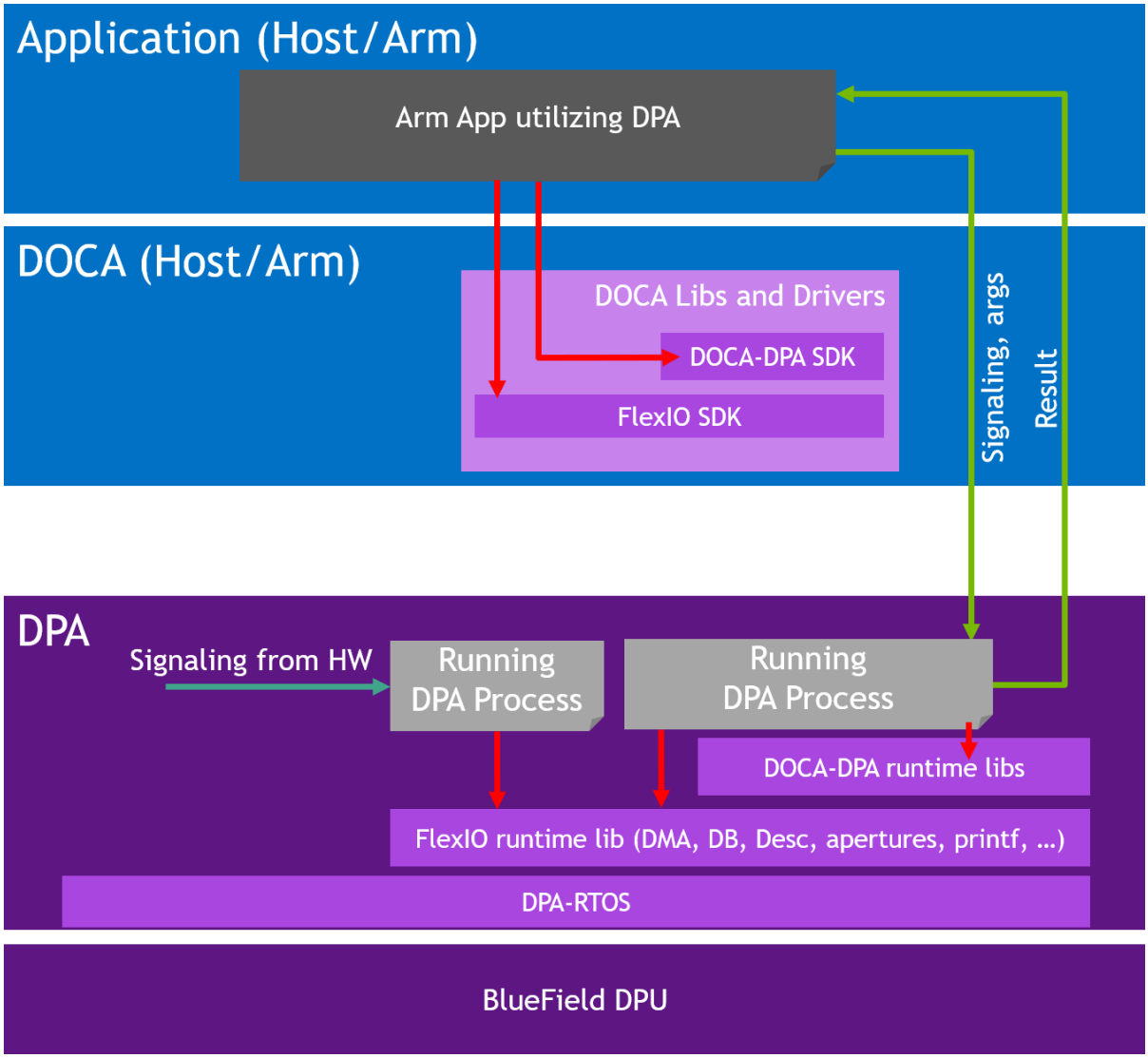
5.4.3 DPA Subsystem

The NVIDIA® BlueField®-3 data-path accelerator (DPA) is an embedded subsystem designed to accelerate workloads that require high-performance access to the NIC engines in certain packet and I/O processing workloads. Applications leveraging DPA capabilities run faster on the DPA than on host. Unlike other programmable embedded technologies, such as FPGAs, the DPA enables a high degree of programmability using the C programming model, multi-process support, tools chains like compilers and debuggers, SDKs, dynamic application loading, and management.

The DPA architecture is optimized for executing packet and I/O processing workloads. As such, the DPA subsystem is characterized by having many execution units that can work in parallel to overcome latency issues (such as access to host memory) and provide an overall higher throughput.

The following diagram illustrates the DPA subsystem. The application accesses the DPA through the DOCA library (DOCA DPA) or the DOCA driver layer (FlexIO SDK). On the host or DPU side, the application loads its code into the DPA (shown as "Running DPA Process") as well as allocates memory, NIC queues, and more resources for the DPA process to access. The DPA process can

use device side libraries to access the resources. The provided APIs support signaling of the DPA process from the host or DPU to explicitly pass control or to obtain results from the DPA.



The threads on the DPA can react independently to incoming messages via interrupts from the hardware, thereby providing full bypass of DPU or Arm CPU for datapath operations.

The following sections provide an overview of the DPA platform design.

5.4.3.1 Multiple Processes on Multiple Execution Units

The DPA platform supports multiple processes with each process having multiple threads. Each thread can be mapped to a different execution unit to achieve parallel execution. The processes operate within their own address spaces and their execution contexts are isolated. Processes are loaded and unloaded dynamically per the user's request. This is achieved by the platform's hardware design (i.e., privilege layers, memory translation units, DMA engines) and a light-weight

real-time operating system (RTOS). The RTOS enforces the privileges and isolation among the different processes.

5.4.3.2 DPA RTOS

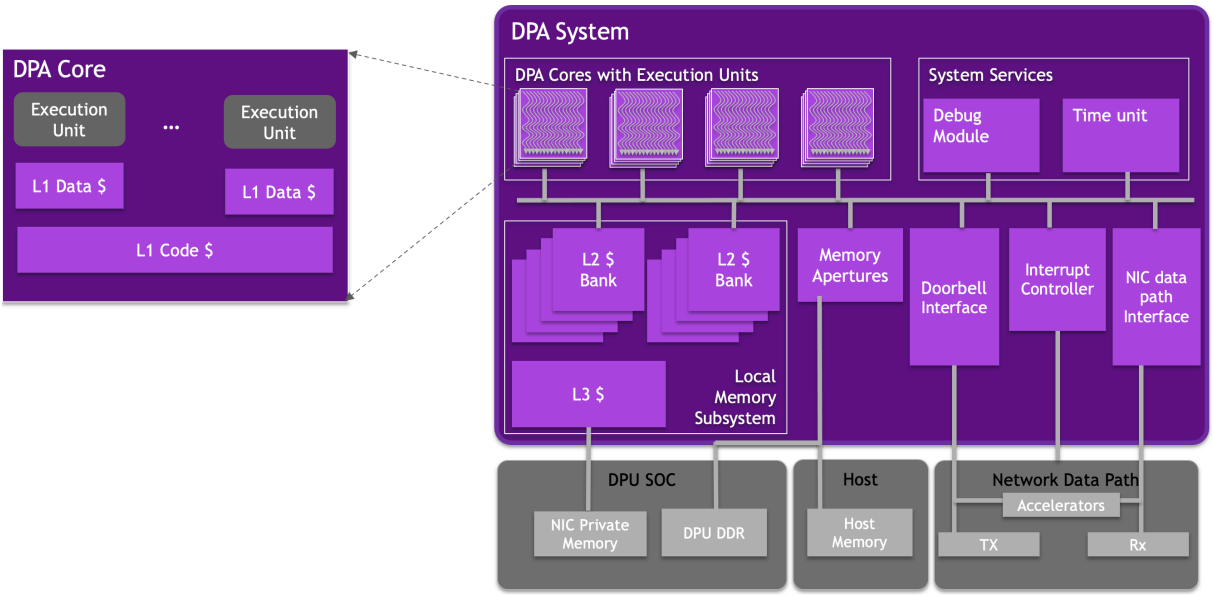
The RTOS is designed to rely on hardware-based scheduling to enable low activation latency for the execution handlers. The RTOS works in a cooperative run-to-completion scheduling model.

Under cooperative scheduling, an execution handler can use the execution unit without interrupts until it relinquishes it. Once relinquished, the execution unit is handed back to the RTOS to schedule the next handler. The RTOS sets a watchdog for the handlers to prevent any handler from unduly monopolizing the execution units.

5.4.3.3 DPA Memory and Caches

The following diagram illustrates the DPA memory hierarchy. Memory accessed by the DPA can be cached at three levels (L1, L2, and L3). Each execution unit has a private L1 data cache. The L1 code cache is shared among all the execution units in a DPA core. The L2 cache is shared among all the DPA cores. The DPA execution units can access external memory via load/store operations through the Memory Apertures.

The external memory that is fetched can be cached directly in L1. The DPA caches are backed by NIC private memory, which is located in the DPU's DDR memory banks. Therefore, the address spaces are scalable and bound only by the size of the NIC's private memory, which in turn is limited only by the DPU's DDR capacity.



See "Memory Model" for more details.

5.4.3.4 DPA Access to NIC Accelerators

The DPA can send and receive any kind of packet toward the NIC and utilize all the accelerators that reside on the BlueField DPU (e.g., encryption/decryption, hash computation, compression/decompression).

The DPA platform has efficient DMA accelerators that enable the different execution units to access any memory location accessible by the NIC in parallel and without contention. This includes both synchronous and asynchronous DMA operations triggered by the execution units. In addition, the NIC can DMA data to the DPA caches to enable low-latency access and fast processing. For example, a packet received from the wire may be "DMA-gathered" directly to the DPA's last level caches.

5.4.3.5 DPA Development

5.4.3.5.1 Overview

5.4.3.5.1.1 DOCA Libs and Drivers

The NVIDIA DOCA framework is the key for unlocking the potential of NVIDIA® BlueField®-3 platforms.

DOCA's software environment allows developers to program the DPA to accelerate workloads. Specifically, DOCA includes:

- DOCA DPA SDK – a high-level SDK for application-level protocol acceleration
- DOCA Flex IO SDK – a low-level SDK to load DPA programs into the DPA, manage the DPA memory, create the execution handlers and the needed hardware rings and contexts
- DPACC – DPA toolchain for compiling and ELF file manipulation of the DPA code

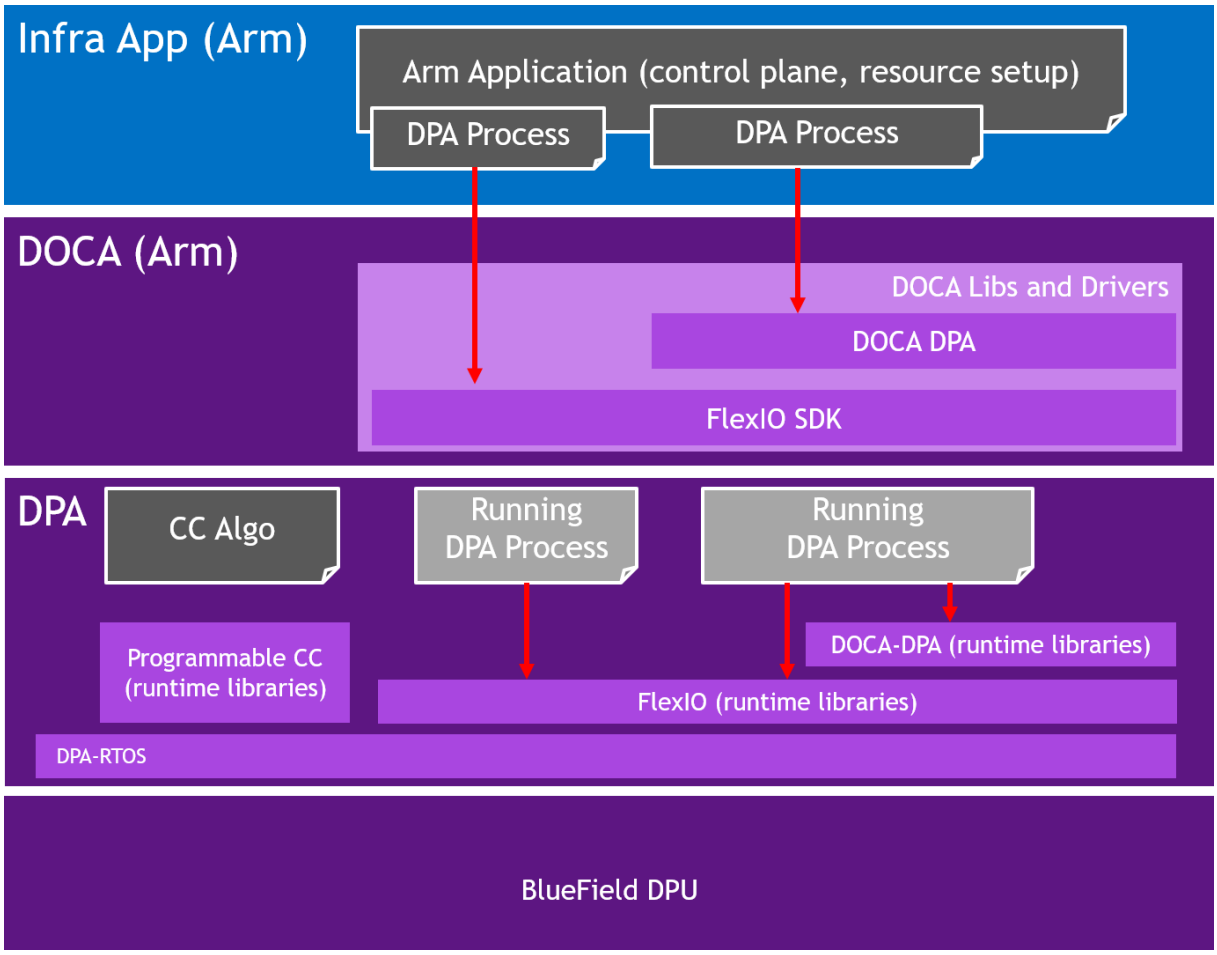
5.4.3.5.1.2 Programming Model

The DPA is intended to accelerate datapath operations for the DPU and host CPU. The accelerated portion of the application using DPA is presented as a library for the host application. The code within the library is invoked in an event-driven manner in the context of a process that is running on the DPA. One or many DPA execution units may work to handle the work associated with network events. The programmer specifies different conditions when each function should be called using the appropriate SDK APIs on the host or DPU.

The DPA cannot be used as a standalone CPU.

Management of the DPA, such as loading processes and allocating memory, is performed from a host or DPU process. The host process discovers the DPA capabilities on the device and drives the control plane to set up the different DPA objects. The DPA objects exist as long as the host process exists. When the host process is destroyed, the DPA objects are freed. The host process decides which functions it wants to accelerate using the DPA: Either its entire data plane or only a part of it.

The following diagram illustrates the different processes that exist in the system:



Compiler

DPACC is a compiler for the DPA processor. It compiles code targeted for the DPA processor into an executable and generates a DPA program. A DPA program is a host library with interfaces encapsulating the DPA executable.

This DPA program is linked with the host application to generate a host executable. The host executable can invoke the DPA code through the DPA SDK's runtime.

Compiler Keywords

DPACC implements the following keywords:

Keyword	Application Usage	Comment
`__dpa_global__`	Annotate all event handlers that execute on the DPA and all common user-defined datatypes (including user-defined structures) which are passed from the host to the DPA as arguments.	Used by the compiler to generate entry points in the DPA executable and automatically replicate user-defined datatypes between the host and DPA.
`__dpa_rpc__`	Annotate all RPC calls which are invoked by the host and execute on the DPA. RPC calls return a value of `uint64_t`.	Used by the compiler to generate RPC specific entry points.

Security Attributes

Security attributes enhance the DPA security model by preventing unauthorized access to hardware DPA resources.

If a DPA application requires non-default security attributes, a security attributes blob must be provided to the DPACC compiler using the **`--dpa-proc-attr`** option.

For more details, refer to the following documentation:

- DPA Application Security Attributes to Blob – Describes how to create the attributes blob and details the **`dpa_attributes`** file format.
- DOCA DPACC Compiler – Provides general information about the compiler.

Flex IO SDK

Flex IO SDK is a low-level event-driven library to program and accelerate functions on the DPA.

Flex IO SDK Execution Model

To load an application onto the DPA, the user must create a process on the DPA, called a Flex IO SDK process. Flex IO SDK processes are isolated from each other like standard host OS processes.

Flex IO SDK supports the following options for executing a user-defined function on the DPA:

1. Flex IO SDK event hander – the event handler executes its function each time an event occurs. An event on this context is a completion event (CQE) received on the NIC completion queue (CQ) when the CQ was in the armed state. The event triggers an internal DPA interrupt that activates the event handler. When the event handler is activated, it is provided with a user-defined argument. The argument in most cases is a pointer to the software execution context of the event handler.
   The following pseudo-code example describes how to create an event handler and attach it to a CQ:

```
// Device code
__dpa_global__ void myFunc(flexio_uintptr_t myArg){
      struct my_db *db = (struct my_db *)myArg;
      get_completion(db->myCq)
      work();
      arm_cq(myCq);
      // reschedule the thread
      flexio_dev_thread_reschedule();
}

// Host code
main() {

      /* Load the application code into the DPA */
      flexio_process_create(device, application, &myProcess);

      /* Create event handler to run my_func with my_arg */
      flexio_event_handler_create(myProcess, myFunc, myArg, &myEventHandler);

      /* Associate the event hanlder with a specific CQ */
      create_cq(&myCQ,… , myEventHandler)

      /* Start the event handler */
      flexio_event_handler_run(myEventHandler)
      …
}
```

2. RPC – remote, synchronous, one-time call of a specific function. RPC is mainly used for the control path to update DPA memory contexts of a process. The RPC's return value is reported back to the host application.
   The following pseudo-code example describes how to use the RPC:

```
// Device code
__dpa_rpc__ uint64_t myFunc(myArg) {
      struct my_db *db = (struct my_db *)myArg;
      if (db->flag) return 1;
      db->flag = 1;
```

```
        return 0;
}

// Host code
main() {
        …

        /* Load the application code into the DPA */
        flexio_process_create(device, application, &myProcess);

        /* run the function */
        flexio_process_call(myProcess, myFunc, myArg, &returnValue);
        …
}
```

Flex IO SDK Multi-hardware Model

The Flex IO SDK enables building a single DPA application which supports multiple DPA hardware models. DPA hardware models refer to different DPA versions tailored to specific NVIDIA devices, such as NVIDIA® BlueField®-3 and NVIDIA® ConnectX®-8.

When an application is built for multiple hardware models, it generates a copy for each specified hardware model. Users can then select the appropriate Flex IO program (the Flex IO representation of the executable) to load into the Flex IO process by searching for the program by its name and hardware model.

Example implementation:

Flex IO program selection

```
/* Initialize the Flex IO application selection attributes for the device. */

/* Set the application name to match. */
flexio_app_sel_attr.app_name = DEV_APP_NAME_XSTR(DEV_APP_NAME);

/* Set the hardware platform to default - this will automatically select the appropriate program. */
flexio_app_sel_attr.hw_model_id = FLEXIO_HW_MODEL_DEF;

/* Specify the IBV device context to use for the hardware model query. */
flexio_app_sel_attr.ibv_ctx = ctx.ibv_ctx;

/* Retrieve the Flex IO application.
 * The Flex IO application is created per hardware model by DPACC.
 * This function matches the selection attributes (application name and HW model)
 * with the available applications and returns the best match. If no exact match is
 * found, a program built for an older hardware model will be selected.
 */
flexio_app_get(&flexio_app_sel_attr, &ctx.flexio_app);
```

This example demonstrates how to search for the appropriate Flex IO program for a device's hardware model or an older compatible version (as new hardware models maintain backward compatibility with older code). Users can also request a program for a specific hardware model, in which case only a Flex IO program explicitly built for that model will be selected.

Flex IO SDK Memory Management

The DPA process can access several memory locations:

- Global variables defined in the DPA process.
- Stack memory – local to the DPA execution unit. Stack memory is not guaranteed to be preserved between different execution of the same handler. For limitations regarding stack memory, refer to the "DPA Development#Memory Model" section.
- Heap memory – this is the process' main memory. The heap memory contents are preserved as long as the DPA process is active.
- External registered memory – remote to the DPA but local to the server. The DPA can access any memory location that can be registered to the local NIC using the provided API. This includes BlueField DRAM, external host DRAM, GPU memory, and more.

The heap and external registered memory locations are managed from the host process. The DPA execution units can load/store from stack/heap and external memory locations. Note that for external memory locations, the window should be configured appropriately using Flex IO Window APIs.

Flex IO SDK allows the user to allocate and populate heap memory on the DPA. The memory can later be used by in the DPA application as an argument to the execution context (RPC and event handler):

```
/* Load the application code into the DPA */
flexio_process_create(device, application, &myProcess);

/* allocate some memory */
flexio_buf_dev_alloc(process, size, ptr)

/* populate it with user defined data */
flexio_host2dev_memcpy(process, src, size, ptr)

/* run the function */
flexio_process_call(myProcess, function, ptr, &return value);
```

Flex IO SDK allows accessing external registered memory from the DPA execution units using Flex IO Window. Flex IO Window maps a memory region from the DPA process address space to an external registered memory. A memory key for the external memory region is required to be associated with the window. The memory key is used for address translation and protection. Flex IO window is created by the host process and is configured and used by the DPA handler during execution. Once configured, LD/ST from the DPA execution units access the external memory directly.

The access for external memory is not coherent. As such, an explicit memory fencing is required to flush the cached data to maintain consistency. See section "Memory Fences" for more.

The following example code demonstrates the window management:

```
// Device code
__dpa_rpc__ uint64_t myFunc(arg1, arg2, arg3)
{
    struct flexio_dev_thread_ctx *dtctx;
    flexio_dev_get_thread_ctx(&dtctx);
    uint32_t windowId = arg1;
    uint32_t mkey = arg2;
    uint64_t *dev_ptr;
    flexio_dev_window_config(dtctx, windowId, mkey );
    /* get ptr to the external memory (arg3) from the DPA process address space */
    flexio_dev_status status = flexio_dev_window_ptr_acquire (dtctx, arg3, dev_ptr);
    /* will set the external memory */
    *dev_ptr = 0xff;
    /* flush the data out */
    __dpa_thread_window_writeback();
    return 0;
}

// Host code
main() {
      /* Load the application code into the DPA */
      flexio_process_create(device, application, &myProcess);
      /* define an array on host */
      uint64_t var= {0};
      /* register host buffer  */
      mkey =ibv_reg_mr(&var, …)
      /* create the window */
      flexio_window_create(process, doca_device->pd, mkey, &window_ctx);
      /* run the function */
      flexio_process_call(myProcess, myFunc, flexio_window_get_id(window_ctx), mkey, &var, &returnValue);
}
```

Send and Receive Operation

A DPA process can initiate send and receive operations using the Flex IO outbox object. The Flex IO outbox contains memory-mapped IO registers that enable the DPA application to issue device doorbells to manage the send and receive planes. The DPA outbox can be configured during run time to perform send and receive from a specific NIC function exposed by the DPU. This capability is not available for Host CPUs that can only access their assigned NIC function.

Each DPA execution engine has its own outbox. As such, each handler can efficiently use the outbox without needing to lock to protect against accesses from other handlers. To enforce the required security and isolation, the DPA outbox enables the DPA application to send and receive only for queues created by the DPA host process and only for NIC functions the process is allowed to access.

Like the Flex IO window, the Flex IO outbox is created by the host process and configured and used at run time by the DPA process.

```
// Device code
__dpa_rpc__ uint64_t myFunc(arg1,arg2,arg3) {

    struct flexio_dev_thread_ctx *dtctx;

    flexio_dev_get_thread_ctx(&dtctx);

    uint32_t outbox = arg1;
    flexio_dev_outbox_config (dtctx, outbox);

    /* Create some wqe and post it on sq */

    /* Send DB on sq*/

    flexio_dev_qp_sq_ring_db(dtctx, sq_pi,arg3);

    /* Poll CQ (cq number is in arg2) */
    return 0;
}

// Host code
main() {

      /* Load the application code into the DPA */
      flexio_process_create(device, application, &myProcess);

      /* Allocate uar */
      uar = ibv_alloc_uar(ibv_ctx);

      /* Create queues*/
      flexio_cq_create(myProcess, ibv_ctx, uar, cq_attr, &myCQ);
      my_hwcq = flexio_cq_get_hw_cq (myCQ);

      flexio_sq_create(myProcess, ibv_ctx, myCQ, uar, sq_attr, &mySQ);
      my_hwsq = flexio_sq_get_hw_sq(mySQ);

      /* Outbox will allow access only for queues created with the same UAR*/
      flexio_outbox_create(process, ibv_ctx, uar, &myOutbox);

      /* Run the function */
      flexio_process_call(myProcess, myFunc, myOutbox, my_hwcq->cq_num, my_hwsq->sq_num,  &return_value);
}
```

Synchronization Primitives

The DPA execution units support atomic instructions to protect from concurrent access to the DPA process heap memory. Using those instructions, multiple synchronization primitives can be designed.

Flex IO currently supports basic spin lock primitives. More advanced thread pipelining can be achieved using DOCA DPA events.

DOCA DPA

The DOCA DPA SDK eases DPA code management by providing high-level primitives for DPA work offloading, synchronization, and communication. This leads to simpler code but lacks the low-level control that Flex IO SDK provides.

User-level applications and libraries wishing to utilize the DPA to offload their code may choose DOCA DPA. Use-cases closer to the driver level and requiring access to low-level NIC features would be better served using Flex IO.

The implementation of DOCA DPA is based on the Flex IO API. The higher level of abstraction enables the user to focus on their program logic and not the low-level mechanics.

Refer to DOCA DPA documentation for more details.

Memory Model

The DPA offers a coherent but weakly ordered memory model. The application is required to use fences to impose the desired memory ordering. Additionally, where applicable, the application is required to write back data for the data to be visible to NIC engines (see the coherency table).

The memory model offers "same address ordering" within a thread. This means that, if a thread writes to a memory location and subsequently reads that memory location, the read returns the contents that have previously been written.

The memory model offers 8-byte atomicity for aligned accesses to atomic datatypes. This means that all eight bytes of read and write are performed in one indivisible transaction.

The DPA does not support unaligned accesses, such as accessing `N` bytes of data from an address not evenly divisible by `N`.

The DPA processes memory can be divided into the following memory spaces:

Memory Space	Definition
Heap	Memory locations within the DPA process heap. Referenced as `__DPA_HEAP` in the code.
Memory	Memory locations belonging to the DPA process (including stack, heap, BSS and data segment) except the memory-mapped IO. Referenced as `__DPA_MEMORY` in the code.
MMIO (memory-mapped I/O)	External memory outside the DPA process accessed via memory-mapped IO. Window and Outbox accesses are considered MMIO. Referenced as `__DPA_MMIO` in the code.
System	All memory locations accessible to the thread within Memory and MMIO spaces as described above. Referenced as `__DPA_SYSTEM` in the code.

Stack Limits

As with many other real-time operating systems (RTOS), the userspace stack is limited in size, and it is user responsibility to ensure the program does not allocate memory beyond the stack's limits. The overall memory range for the userspace stack is slightly less than 8 KB (8184 bytes), meaning that the user's program and the ABI used by the compiler for the function calls themselves should not cross this limit. DPA programs consuming more memory than the amount allocated for the stack would cause anomalous behavior.

Coherency

The coherency between the DPA threads and NIC engines is described in the following table:

Producer	Observer	Coherency	Comments
DPA thread	NIC engine	Not coherent	Data to be read by the NIC must be written back using the appropriate intrinsic (see section "DPA Development#Memory Fence and Cache Control Usage Examples").
NIC engine	DPA Thread	Coherent	Data written by the NIC is eventually visible to the DPA threads. The order in which the writes are visible to the DPA threads is influenced by the ordering configuration of the memory region (see **IBV_ACCESS_RELAXED_ORDERING**). In a typical example of the NIC writing data and generating a completion entry (CQE), it is guaranteed that when the write to the CQE is visible, the DPA thread can read the data without additional fences.
DPA thread	DPA thread	Coherent	Data written by a DPA thread is eventually visible to the other DPA threads without additional fences. The order in which writes made by a thread are visible to other threads is undefined when fences are not used. Programmers can enforce ordering of updates using fences (see section "DPA Development#Memory Fences").

Memory Fences

Fence APIs are intended to impose memory access ordering. The fence operations are defined on the different memory spaces. See information on memory spaces under section "DPA Development#Memory Model".

The fence APIs apply ordering between the operations issued by the calling thread. As a performance note, the fence APIs also have a side effect of writing back data to the memory space used in the fence operation. However, programmers should not rely on this side effect. See section "DPA Development#Cache Control" for explicit cache control operations. The fence APIs have an effect of a compiler-barrier which means that memory accesses are not reordered around the fence API invocation by the compiler.

A fence applies between the "predecessor" and the "successor" operations. The predecessor and successor ops can be refenced using **__DPA_R**, **__DPA_W**, and **__DPA_RW** in the code.

The generic memory fence operation can operate on any memory space and any set of predecessor and successor operations. The other fence operations are provided as convenient shortcuts that are specific to the use case. It is preferable for programmers to use the shortcuts when possible.

Fence operations can be included using the **dpaintrin.h** header file.

Generic Fence

```
void __dpa_thread_fence(memory_space, pred_op, succ_op);
```

This fence can apply to any DPA thread memory space. Memory spaces are defined under section "DPA Development#Memory Model". The fence ensures that all operations (**pred_op**) performed by the calling thread, before the call to **__dpa_thread_fence()**, are performed and made visible to all threads in the DPA, host, NIC engines, and peer devices as occurring before all operations (**succ_op**) to the memory space after the call to **__dpa_thread_fence()**.

System Fence

```
void __dpa_thread_system_fence();
```

This is equivalent to calling **__dpa_thread_fence(__DPA_SYSTEM, __DPA_RW, __DPA_RW)**.

Outbox Fence

```
void __dpa_thread_outbox_fence(pred_op, succ_op);
```

This is equivalent to calling **__dpa_thread_fence(__DPA_MMIO, pred_op, succ_op)**.

Window Fence

```
void __dpa_thread_window_fence(pred_op, succ_op);
```

This is equivalent to calling **__dpa_thread_fence(__DPA_MMIO, pred_op, succ_op)**.

Memory Fence

```
void __dpa_thread_memory_fence(pred_op, succ_op);
```

This is equivalent to calling **__dpa_thread_fence(__DPA_MEMORY, pred_op, succ_op)**.

Cache Control

Cache control operations allow the programmer to exercise fine-grained control over data resident in the DPA's caches. They have an effect of a compiler-barrier. The operations can be included using the **dpaintrin.h** header file.

Window Read Contents Invalidation

```
void __dpa_thread_window_read_inv();
```

The DPA can cache data that was fetched from external memory using a window. Subsequent memory accesses to the window memory location may return the data that is already cached. In some cases, it is required by the programmer to force a read of external memory (see example under "DPA Development#Polling Externally Set Flag"). In such a situation, the window read contents cached must be dropped.

This function ensures that contents in the window memory space of the thread before the call to **__dpa_thread_window_read_inv()** are invalidated before read operations made by the calling thread after the call to **__dpa_thread_window_read_inv()**.

Window Writeback

```
void __dpa_thread_window_writeback();
```

Writes to external memory must be explicitly written back to be visible to external entities.

This function ensures that contents in the window space of the thread before the call to **__dpa_thread_window_writeback()** are performed and made visible to all threads in the DPA,

host, NIC engines, and peer devices as occurring before any write operation after the call to **__dpa_thread_window_writeback()**.

Memory Writeback

```
void __dpa_thread_memory_writeback();
```

Writes to DPA memory space may need to be written back. For example, the data must be written back before the NIC engines can read it. Refer to the coherency table for more.

This function ensures that the contents in the memory space of the thread before the call to **__dpa_thread_writeback_memory()** are performed and made visible to all threads in the DPA, host, NIC engines, and peer devices as occurring before any write operation after the call to **__dpa_thread_writeback_memory()**.

Memory Fence and Cache Control Usage Examples

These examples illustrate situations in which programmers must use fences and cache control operations.

In most situations, such direct usage of fences is not required by the application using Flex IO or DOCA DPA SDKs as fences are used within the APIs.

Issuing Send Operation

In this example, a thread on the DPA prepares a work queue element (WQE) that is read by the NIC to perform the desired operation.

The ordering requirement is to ensure the WQE data contents are visible to the NIC engines read it. The NIC only reads the WQE after the doorbell (MMIO operation) is performed. Refer to coherency table.

	User Code – WQE Present in DPA Memory	Comment
1	Write WQE	Write to memory locations in the DPA (memory space = **__DPA_MEMORY**)
2	**__dpa_thread_memory_writeback();**	Cache control operation
3	Write doorbell	MMIO operation via Outbox

In some cases, the WQE may be present in external memory. See the description of **flexio_qmem** below. The table of operations in such a case is below.

	User Code – WQE Present in External Memory	Comment
1	Write WQE	Write to memory locations in the DPA (memory space = **__DPA_MMIO**)
2	**__dpa_thread_window_writeback();**	Cache control operation
3	Write doorbell	MMIO operation via Outbox

Posting Receive Operation

In this example, a thread on the DPA is writing a WQE for a receive queue and advancing the queue's producer index. The DPA thread will have to order its writes and writeback the doorbell record contents so that the NIC engine can read the contents.

	User Code – WQE Present in DPA Memory	Comment
1	Write WQE	Write to memory locations in the DPA (memory space = `__DPA_MEMORY`)
2	`__dpa_thread_memory_fence(__DPA_W, __DPA_W);`	Order the write to the doorbell record with respect to WQE
3	Write doorbell record	Write to memory locations in the DPA (memory space = `__DPA_MEMORY`)
4	`__dpa_thread_memory_writeback();`	Ensure that contents of doorbell record are visible to the NIC engine

Polling Externally Set Flag

In this example, a thread on the DPA is polling on a flag that will be updated by the host or other peer device. The memory is accessed by the DPA thread via a window. The DPA thread must invalidate the contents so that the underlying hardware performs a read.

User Code – Flag Present in External Memory	Comment
`while (!flag) { __dpa_thread_window_read_inv(); }`	`flag` is a memory location read using a window

Thread-to-thread Communication

In this example, a thread on the DPA is writing a data value and communicating that the data is written to another thread via a flag write. The data and flag are both in DPA memory.

User Code – Thread 1	User Code – Thread 2	Comment
		Initial condition, `flag = 0`
`var1 = x;`	`while(*((volatile int *)&flag) !=1);`	• Thread 1 - write to var1 • Thread 2 - `flag` is accessed as a volatile variable, so the compiler preserves the intended program order of reads
`__dpa_thread_memory_fence(__DPA_W, __DPA_W);`		Thread 1 – write to flag cannot bypass write to `var1`
	`var_t2 = var1;`	
`flag = 1;`	`assert(var_t2 == x);`	`var_t2` must be equal to `x`

Setting Flag to be Read Externally

In this example, a thread on the DPA sets a flag that is observed by a peer device. The flag is written using a window.

User Code – Flag Present in External Memory	Comment
`flag = data;`	`flag` is updated in local DPA memory
`__dpa_thread_window_writeback();`	Contents from DPA memory for the window are written to external memory

Polling Completion Queue

In this example, a thread on the DPA reads a NIC completion queue and updates its consumer index.

First, the DPA thread polls the memory location for the next expected CQE. When the CQE is visible, the DPA thread processes it. After processing is complete, the DPA thread updates the CQ's consumer index. The consumer index is read by the NIC to determine whether a completion queue entry has been read by the DPA thread. The consumer index is used by the NIC to monitor a potential completion queue overflow situation.

User Code – CQE in DPA Memory	Comment
`while(*((volatile uint8_t *)&cq→op_own) & 0x1 == hw_owner);`	Poll CQ owner bit in DPA memory until the value indicates the CQE is in software ownership. Coherency model ensures update to the CQ is visible to the DPA execution unit without additional fences or cache control operations. Coherency model ensures that data in the CQE or referenced by it are visible when the CQE changes ownership to software.
`process_cqe();`	User processes the CQE according to the application's logic.
`cq→cq_index++; // next CQ index. Handle wraparound if necessary`	Calculate the next CQ index taking into account any wraparound of the CQ depth.
`update_cq_dbr(cq, cq_index); // writes cq_index to DPA memory`	Memory operation to write the new consumer index.
`__dpa_thread_memory_writeback();`	Ensures that write to CQ's consumer index is visible to the NIC. Depending on the application's logic, the `__dpa_thread_memory_writeback()` may be coalesced or eliminated if the CQ is configured in overrun ignore mode.
`arm_cq();`	Arm the CQ to generate an event if this handler is going to call `flexio_dev_thread_reschedule()`. Arming the CQ is not required if the handler calls `flexio_dev_thread_finish()`.

DPA-specific Operations

The DPA supports some platform-specific operations. These can be accessed using the functions described in the following subsections. The operations can be included using the `dpaintrin.h` header file.

Clock Cycles

```
uint64_t __dpa_thread_cycles();
```

Returns a counter containing the number of cycles from an arbitrary start point in the past on the execution unit the thread is currently scheduled on.

Note that the value returned by this function in the thread is meaningful only for the duration of when the thread remains associated with this execution unit.

This function also acts as a compiler barrier, preventing the compiler from moving instructions around the location where it is used.

Timer Ticks

```
uint64_t __dpa_thread_time();
```

Returns the number of timer ticks from an arbitrary start point in the past on the execution unit the thread is currently scheduled on.

Note that the value returned by this function in the thread is meaningful only for the duration of when the thread remains associated with this execution unit.

This intrinsic also acts as a compiler barrier, preventing the compiler from moving instructions around the location where the intrinsic is used.

Instructions Retired

```
uint64_t __dpa_thread_inst_ret();
```

Returns a counter containing the number of instructions retired from an arbitrary start point in the past by the execution unit the thread is currently scheduled on.

Note that the value returned by this function in the software thread is meaningful only for the duration of when the thread remains associated with this execution unit.

This intrinsic also acts as a compiler barrier, preventing the compiler from moving instructions around the location where the intrinsic is used.

Fixed Point Log2

```
int __dpa_fxp_log2(unsigned int);
```

This function evaluates the fixed point Q16.16 base 2 logarithm. The input is an unsigned integer.

Fixed Point Reciprocal

```
int __dpa_fxp_rcp(int);
```

This function evaluates the fixed point Q16.16 reciprocal (1/x) of the value provided.

Fixed Point Pow2

```
int __dpa_fxp_pow2(int);
```

This function evaluates the fixed point Q16.16 power of 2 of the provided value.

DPA Attributes

DPA programs should make sure to enable the necessary DPA features for their application:

Feature	Description
Programmable_Congestion_Control	Enables application to serve Programmable Congestion Control events
Device_Emulation	Enables application to emulate a PCIe device
Generating_Empty_CQE	Enables application to generate Empty CQE on a given CQ (e.g., for triggering other software entries).
Congestion_Control_Flow_Context_Access	Enables application to access Congestion Control Flow Context

Enabling these features is done using a dedicated tool. See the DPA Application Security Attributes to Blob guide for details.

5.4.3.5.2 Flex IO SDK

⚠ The quality status of DOCA libraries is listed here.

This chapter provides an overview and configuration instructions for DOCA Flex IO SDK API.

The DPA processor is an auxiliary processor designed to accelerate packet processing and other data-path operations. The Flex IO SDK exposes an API for managing the DPA device and executing native code over it.

The DPA processor is supported on NVIDIA® BlueField®-3 DPUs and later generations.

After DOCA installation, Flex IO SDK headers may be found under **/opt/mellanox/flexio/include** and libraries may be found under **/opt/mellanox/flexio/lib/**.

5.4.3.5.2.1 Prerequisites

DOCA Flex IO applications can run either on the host machine or on the target DPU.

Developing programs over Flex IO SDK requires knowledge of DPU networking queue usage and management.

5.4.3.5.2.2 Architecture

Flex IO SDK library exposes a few layers of functionality:

- **libflexio** – library for Host-side operations. It is used for resource management.
- **libflexio-dev** – library for DPA-side operations. It is used for data path implementation.
- **libflexio-libc** – a lightweight C library for DPA device code. **libflexio_libc** may expose very partial functionality compared to a standard **libc**.

A typical application is composed of two parts: One running on the host machine or the DPU target and another running directly over the DPA.

5.4.3.5.2.3 API

Please refer to the *DOCA API References*.

5.4.3.5.2.4 Resource Management

DPA programs cannot create resources. The responsibility of creating resources, such as Flex IO process, thread, outbox and window, as well as queues for packet processing (completion, receive and send), lies on the DPU program. The relevant information should be communicated (copied) to the DPA side and the address of the copied information should be passed as an argument to the running thread.

Example

Host side:

1. Declare a variable to hold the DPA buffer address.

```
flexio_uintptr_t app_data_dpa_daddr;
```

2. Allocate a buffer on the DPA side.

```
flexio_buf_dev_alloc(flexio_process, sizeof(struct my_app_data), &app_data_dpa_daddr);
```

3. Copy application data to the DPA buffer.

```
flexio_host2dev_memcpy(flexio_process, (uintptr_t)app_data, sizeof(struct my_app_data),
app_data_dpa_daddr);
```

struct my_app_data should be common between the DPU and DPA applications so the DPA application can access the struct fields.
The event handler should get the address to the DPA buffer with the copied data:

```
flexio_event_handler_create(flexio_process, net_entry_point, app_data_dpa_daddr, NULL, flexio_outbox,
&app_ctx.net_event_handler)
```

DPA side:

```
__dpa_rpc__ uint64_t event_handler_init(uint64_t thread_arg)
{
      struct my_app_data *app_data;
      app_data = (my_app_data *)thread_arg;
      ...
}
```

5.4.3.5.2.5 DPA Memory Management

As mentioned previously, the DPU program is responsible for allocating buffers on the DPA side (same as resources). The DPU program should allocate device memory in advance for the DPA program needs (e.g., queues data buffer and rings, buffers for the program functionality, etc).

The DPU program is also responsible for releasing the allocated memory. For this purpose, the Flex IO SDK API exposes the following memory management functions:

```
flexio_status flexio_buf_dev_alloc(struct flexio_process *process, size_t buff_bsize, flexio_uintptr_t
*dest_daddr_p);
```

```
flexio_status flexio_buf_dev_free(flexio_uintptr_t daddr_p);
flexio_status flexio_host2dev_memcpy(struct flexio_process *process, void *src_haddr, size_t buff_bsize,
flexio_uintptr_t dest_daddr);
flexio_status flexio_buf_dev_memset(struct flexio_process *process, int value, size_t buff_bsize, flexio_uintptr_t
dest_daddr);
```

Allocating NIC Queues for Use by DPA

The Flex IO SDK exposes an API for allocating work queues and completion queues for the DPA. This means that the DPA may have direct access and control over these queues, allowing it to create doorbells and access their memory.

When creating a Flex IO SDK queue, the user must pre-allocate and provide memory buffers for the queue's work queue elements (WQEs). This buffer may be allocated on the DPU or the DPA memory.

To this end, the Flex IO SDK exposes the **flexio_qmem** struct, which allows the user to provide the buffer address and type (DPA or DPU).

Memory Allocation Best Practices

To optimize process device memory allocation, it is recommended to use the following allocation sizes (or closest to it):

- Up to 1 page (4KB)
- 2^6 pages (256KB)
- 2^{11} pages (8MB)
- 2^{16} pages (256MB)

Using these sizes minimizes memory fragmentation over the process device memory heap. If other buffer sizes are required, it is recommended to round the allocation up to one of the listed sizes and use it for multiple buffers.

5.4.3.5.2.6 DPA Window

DPA windows are used to access external memory, such as on the DPU's DDR or host's memory. DPA windows are the software mechanism to use the Memory Apertures mentioned in section "DPA Memory and Caches". To use the window functionality, DPU or host memory must be registered for the device using the **ibv_reg_mr()** call.

Both the address and size provided to this call must be 64 bytes aligned for the window to operate. This alignment may be obtained using the **posix_memalign()** allocation call.

5.4.3.5.2.7 DPA Event Handler

Default Window/Outbox

The DPA event handler expects a DPA window and DPA outbox structs upon creation. These are used as the default for the event handler thread. Users may choose to set one or both to NULL, in which case there would be no valid default value for one/both of them.

Upon thread invocation on the DPA side, the thread context is set for the provided default IDs. If, at any point, the outbox/window IDs are changed, then the thread context on the next invocation is restored to the default IDs. This means that the DPA Window MKey must be configured each time the thread is invoked, as it has no default value.

Execution Unit Management

DPA execution units (EUs) are the equivalent to logical cores. For a DPA program to execute, it must be assigned an EU.

It is possible to set EU affinity for an event handler upon creation. This causes the event handler to execute its DPA program over specific EUs (or a group of EUs).

DPA supports three types of affinity: `none`, `strict`, `group`.

The affinity type and ID, if applicable, are passed to the event handler upon creation using the `affinity` field of the `flexio_event_handler_attr` struct.

For more information, please refer to *Single Point of Resource Distribution*.

Execution Unit Partitions

To work over DPA, an EU partition must be created for the used device. A partition is a selection of EUs marked as available for a device. For the DPU ECPF, a default partition is created upon boot with all EUs available in it. For any other device (i.e., function), the user must create a partition. This means that running an application on a non-ECPF function without creating a partition would result in failure.

Flex IO SDK uses `strict` and `none` affinity for internal threads, which require a partition with at least one EU for the participating devices. Failing to comply with this assumption may cause failures.

Virtual Execution Units

Users should be aware that beside the default EU partition, which is exposed to the real EU numbers, all other partitions created use virtual EUs.

For example, if a user creates a partition with the range of EUs 20-40, querying the partition info from one of its virtual HCAs (vHCAs) it would display EUs from 0-20. So, the real EU number, 39 in this example, would correspond to the virtual EU number 19.

5.4.3.5.2.8 Version API and Backward Compatibility

Flex IO SDK supports partial backward compatibility. This may follow one of the following options:

1. Work only with the latest version. The user must align their entire code according to the changes in the Flex IO SDK API listed in the document accompanying each version.
2. Ensure partial backward compatibility for the working code. The user must inform the SDK which version they intend to work with. The SDK provides a set of tools that ensure backward compatibility. The set consists of compile-time and runtime tools.

Version API Toolkit

To support backward compatibility, the Flex IO SDK uses the macros `FLEXIO_VER` for the host and `FLEXIO_DEV_VER` for the DPA device. The macros have 3 parameters, where the first is the major version (year), the second is the minor version (month), and the third is the sub-minor version (not used, always 0).

Compile-time

This toolkit is available for both the host and DPA device. The header files **flexio_ver.h** and **flexio_dev_ver.h** contain the macros **FLEXIO_VER** and **FLEXIO_VER_LATEST** for the host and **FLEXIO_DEV_VER** and **FLEXIO_DEV_VER_LATEST** for the DPA device. For example, to set backward compatibility for version 24.04, the user must declare the following construct for the host:

```
#include <libflexio/flexio_ver.h>
#define FLEXIO_VER_USED FLEXIO_VER(24, 4, 0)
#include <libflexio/flexio.h>
```

And the user must declare the following construct for the DPA device:

```
#include <libflexio-dev/flexio_dev_ver.h>
#define FLEXIO_DEV_VER_USED FLEXIO_DEV_VER(24, 4, 0)
#include <libflexio-dev/flexio_dev.h>
```

Where **24** is the major version, and **4** is the minor version.

The files **flexio.h** and **flexio_dev.h** have the macros **FLEXIO_CURRENT_VERSION** and **FLEXIO_LAST_SUPPORTED_VERSION** for the host **FLEXIO_DEV_CURRENT_VERSION** and **FLEXIO_DEV_LAST_SUPPORTED_VERSION** for the DPA device. These versions are provided for internal use and user information. The user should not use these macros.

Runtime

This toolkit is only present for the host. For backward compatibility in runtime, the user can call the function **flexio_status flexio_version_set(uint64_t version);** in **flexio.h** once before calling any other function from the API, with the version parameter they wish to work with. The function returns an error in the following cases:

- If the specified version is less than **FLEXIO_LAST_SUPPORTED_VERSION**
- If it exceeds **FLEXIO_CURRENT_VERSION**
- If the function is called again with a version value different from the previous one

```
status = flexio_version_set(FLEXIO_VER(24, 4, 0));
if (status == FLEXIO_STATUS_FAILED)
{
    return ERROR;
}
```

It is recommended to use the **FLEXIO_VER_USED** macro as a parameter:

```
flexio_version_set(FLEXIO_VER_USED);
```

End of Backward Compatibility

The backward compatibility tools are designed to have an endpoint. With each new version, it is possible to gradually raise the value of **FLEXIO_LAST_SUPPORTED_VERSION** for the host and **FLEXIO_DEV_LAST_SUPPORTED_VERSION** for the DPA device. If **FLEXIO_VER_USED** equals **FLEXIO_LAST_SUPPORTED_VERSION**, then the compiler will issue a warning. This is a sign for the user to start transitioning to a newer version. This way the user has time at least until the next version to modify their code to comply with the older version. If **FLEXIO_VER_USED** is lower than

FLEXIO_LAST_SUPPORTED_VERSION, then the compiler will issue errors. This is a sign for the user to immediately transition to a newer version. The same behavior for the DPA device.

DPA Device-to-Host Capabilities

A common challenge in DPA-based SDKs is binary compatibility, as the host-side components are often provided as shared objects (**.so**) while the device-side components are statically linked into the DPA application.

The DPA "CAPS" (capabilities) mechanism is designed to solve this. It allows the DPA application to embed information (such as version or feature support) that the host application can query at runtime and adjust its behavior accordingly.

This is implemented in two parts:

1. On the DPA (device side): A variable of any type is defined and placed into a custom ELF section using the standard C library **__attribute__** annotation. This is processed at build time.

```
// Example DPA-side capability
__attribute__((section("<section_name>"), used))
my_cap_t my_dpa_capabilities = { .version = 2, .feature_x = true };
```

2. On the host side: At runtime, the host application uses the FlexIO SDK **flexio_app_extract_symbol** API to retrieve the contents of that variable from the custom section by name.

> ⚠ This process relies on a build-time/runtime distinction. The DPA variable is embedded into the DPA binary at build time, and the host-side FlexIO API reads this static information at runtime.

5.4.3.5.2.9 Application Debugging

Because application execution is divided between the host side and the DPA processor services, debugging may be somewhat challenging, especially since the DPA side does not have a terminal allowing the use of the C stdio library printf services.

Using DPA GDB

See DOCA DPA GDB Server Tool for information.

Using Device Messaging Stream API

Another logging (messaging) option is to use Flex IO SDK infrastructure to send strings or formatted text in general, from the DPA side to the host side console or file. The host side's **flexio.h** file provides the **flexio_msg_stream_create** API function for initializing the required infrastructures to support this. Once initialized, the DPA side must have the thread context, which can be obtained by calling **flexio_dev_get_thread_ctx**. **flexio_dev_msg** can then be called to write a string generated on the DPA side to the stream created (using its ID) on the host side, where it is directed to the console or a file, according to user configuration in the creation stage.

It is important to call **flexio_msg_stream_destroy** when exiting the DPU application to ensure proper clean-up of the print mechanism resources.

Device messages use an internal QP for communication between the DPA and the DPU. When running over an InfiniBand fabric, the user must ensure that the subnet is well-configured, and that the relevant device's port is in **active** state.

Message Stream Functionality

The user can create as many streams as they see fit, up to a maximum of **FLEXIO_MSG_DEV_MAX_STREAMS_AMOUNT** as defined in **flexio.h**.

Every stream has its own messaging level which serves as a filter where messages with a level below that of the stream are filtered out.

The first stream created is the **default_stream** gets stream ID 0, and it is created with messaging level **FLEXIO_MSG_DEV_INFO** by default.

The stream ID defined by **FLEXIO_MSG_DEV_BROADCAST_STREAM** serves as a broadcast stream which means it messagaes all open streams (with the proper messaging level).

A stream can be configured with a synchronization mode attribute according to the following options:

- **sync** – displays the messages as soon as they are sent from the device to the host side using the verb SEND.
- **async** – uses the verb RDMA write. When the programmer calls the stream's flush functionality, all the messages in the buffer are displayed (unless there was a wraparound due to the size of messages being bigger than the size allocated for them). In this synchronization mode, the flush should be called at the end of the run.
- **batch** – uses RDMA write and RDMA write with immediate. It works similarly to the async mode, except the fact each batch size of messages is being flushed and therefore displayed automatically in every batch. The purpose is to allow the host to use fewer resources for device messaging.

Device Messaging Assumptions

Device messaging uses RPC calls to create, modify, and destroy streams. By default, these RPC calls run with affinity **none**, which requires at least one available EU on the default group. If the user wants to set the management affinity of a stream to a different option (any affinity option is supported, including forcing **none**, which is the default behavior) they should specify this in the stream attributes using the **mgmt_affinity** field.

Printf Support

FlexIO SDK provides limited **printf** functionality, which is not a full implementation of the standard libc **printf**. The **printf** functions are operational only after the message stream mechanism is enabled and will cease to function after it is destroyed.

The following format specifiers and modifiers are supported:

- Format specifiers: **%c**, **%s**, **%d**, **%u**, **%i**, **%x**, **%X**, **%o**, **%p**, **%%**
- Flags: **0**, **-**, **+**, **#**, space
- Width: *****, numeric values
- Precision: **.***, **.number**

- Length modifiers: `hh` , `h` , `l` , `ll`

The following modifiers are not supported:

- Floating-point modifiers: `%a` , `%A` , `%e` , `%E` , `%f` , `%lf` , `%LF`
- Writing the number of characters written so far: `%n`

For I/O purposes, it is recommended to use the internal device messaging stream API instead of `printf` .

Core Dump

If the DPA process encounters a fatal error, the user can create a core dump file to review the application's status at that point using a GDB app.

Creating a core dump file can be done after the process has crashed (as indicated by the `flexio_err_status` API) and before the process is destroyed by calling the `flexio_coredump_create` API.

Recommendations for opening DPA core dump file using GDB:

- Use the `gdb-multiarch` application
- The `Program` parameter for GDB should be the device-side ELF file
    - Use the `dpacc-extract` tool (provided with the DPACC package) to extract the device-side ELF file from the application's ELF file

5.4.3.5.2.10 Application Flow Analysis (dTrace)

FlexIO provides a tracing mechanism designed to capture code flow information with minimal impact on performance. This feature is known as dTrace.

dTrace Concepts

Concept	Explanation
Trace points (TPs)	Developers place TPs within the DEV code. When dTrace is active, passing these points triggers a report.
Identification	Every TP is identified by a `u32` number.
Arguments	Every TP can optionally include up to five `u64` arguments.
Sequencing	Every TP event is assigned a unique, sequential number.
Report Structure	A report contains the TP number, arguments (if provided), sequence number, reporting thread ID, and a timestamp.
Lossy Collection	• Collisions: Data collection is lossy. In the case of a collision (simultaneous TP reporting from different threads), the conflicting event is dropped. A gap in the sequence numbers indicates dropped reports. • Transport: Transportation of reports from DEV to HOST is also lossy; unreliable connections are used to minimize overhead. • Host Processing: Reports may be dropped on the HOST side if the receiver is stuck or overwhelmed.

5.4.3.5.2.11 Tracing vs. Printing

It is crucial to understand the differences between Tracing (dTrace) and Printing.

- Printing (e.g., using **flexio_dev_print()**) guarantees that data is delivered to the viewer. However, the cost is execution time. If a print statement is placed in a critical path, it will significantly impact performance.
- Tracing (TP) does not guarantee the delivery of every report. However, because the overhead is minimal, TPs can be left in released code (used judiciously). This allows users to analyze real code flow under real traffic conditions.

> **Guideline**
>
> If you need to report an error case, use Print. The speed penalty is acceptable for errors. If you need to analyze high-frequency code flow, use Trace. Be aware that some reports may be lost.

Tool Comparison

Feature	Print	dTrace
Ease of Use	Very easy. Results are displayed in a human-readable format.	Challenging. Requires reading abstract data or using additional visualization tools.
Delivery Guarantee	Lossless. Delivery is guaranteed.	Lossy. Data may be dropped during collisions or transport.
Long-term Execution	Impossible/Useless. Not suitable for continuous reporting to the console.	Supported. Can run indefinitely without stopping the application.
Output Format	Human-readable text to screen or file.	Binary file, CSV file, or user callback. Post-processing tools are required for visualization.
On/Off Mode	generally supported, but problematic for long durations.	Supported. Preferred mode. Collect data only when analysis is required.
Management	Not supported via system console.	Supported. Tracing can be activated/deactivated from the host console without modifying user code.
Severity Levels	Supported (ERR / WARN / INFO / DBG).	Not supported. Errors and debug messages should be reported using **print()**.

dTrace Usage (HOST)

The following minimal code is required to initialize and start dTrace with the default configuration:

C code. Host.

```
/* Initialize dTrace */
flexio_dtrace_init(flexio_process);

/* Optional: Change dTrace parameters here if non-default config is needed */
flexio_dtrace_create(flexio_process);

/* Start dTrace - trace points are now active */
flexio_dtrace_start(flexio_process);

/* ... Application Runtime ... */

/* Destroy dTrace to release allocated resources */
```

```
flexio_dtrace_destroy(ctx.flexio_process);
```

Refer to **libflexio/flexio.h** for extended dTrace API information and defaults.

dTrace Usage (DEV)

Place Trace Points (TPs) at desired locations in your code. During development, TPs can be used as a substitute for standard prints.

> **Performance Warning**
>
> Even if dTrace is OFF, every TP in the code adds at least one assembly instruction and requires a read from common memory. On some platforms, this memory read may introduce latency. Remove unnecessary TPs before committing production code.

Example of TP placement:

C code. DEV

```
#define START_TP_NUMBER 1981

/* Basic timestamp/start marker */
flexio_dev_ts(START_TP_NUMBER);
...
/* Trace point with function name and return address */
flexio_dev_tp(__func__, (uint64_t)__builtin_return_address(0));
...
/* Trace point with string and line number */
flexio_dev_tp("Point 2", __LINE__);
...
/* Trace point with string, line number, and variable */
flexio_dev_tp("Wrong argument", __LINE__, count);
...
/* Trace point with formatted arguments */
flexio_dev_tp("Arguments %lu %lu %lu %lu", count, arg1, arg2, arg3);
...
/* Trace point with raw numeric ID and 5 arguments */
flexio_dev_tp(5, count, arg1, arg2, arg3, i);
...
/* Termination trace point */
flexio_dev_tp("Finish");
```

These examples demonstrate various aspects of TP usage. Combining these variations allows for efficient debugging.

dTrace Output Analysis

dTrace outputs information as binary data (via file or user callback) or as a CSV file. Both contain identical information. The CSV format is shown below for readability:

CSV output file

```
# line, seq_number,     timestamp,  tid,        TP#,         a1,  a2,  a3,  a4,  a5
      0,          0, 6543090767984, 8010, 0x400033e1, 0x40002458,    ,    ,    ,
      1,          1, 6543090767986, 8010, 0x4000347b,       0x12,    ,    ,    ,
      2,        252, 6543090768005, 8006, 0x40003388,          0, 0x1, 0x2, 0x3,
      3,        653, 6543090768006, 800d, 0x40003328,          0, 0x1, 0x2, 0x3,
      4,        654, 6543090768007, 800d, 0x4000347b,       0x27,    ,    ,    ,
      5,       7586, 6543090768009, 8010,       0x2a,       0x28,    ,    ,    ,
      6,      37654, 6543090768023, 8013, 0x40003351,          ,    ,    ,    ,
```

> While the CSV contains all necessary raw data, it is recommended to use external utilities to convert this file into a more user-friendly text format. Currently, a specific converter is not provided as part of the FlexIO package.

5.4.3.5.2.12 Experimental Flex IO APIs

Flex IO APIs are now marked as experimental which requires adjustment of users' build definitions as follows:

- For host-side projects, users must add the compilation flag `-DFLEXIO_ALLOW_EXPERIMENTAL_API` to their projects. For example:
  - If using meson, add the command `add_project_arguments('-DFLEXIO_ALLOW_EXPERIMENTAL_API', language: 'c', native: true)`
  - If using a makefile, add `-DFLEXIO_ALLOW_EXPERIMENTAL_API` to the compiler flags
- For device-side projects, add the following when launching the DPACC application:
  - Add `-DFLEXIO_ALLOW_EXPERIMENTAL_API` to the `--hostcc_args` flag
  - Add `-DFLEXIO_DEV_ALLOW_EXPERIMENTAL_API` to the `--devicecc_options` flag

The absence of these macros would result in compilation warnings or errors if the `-Werror` flag is used.

> ⚠ If the host-side project uses the pkg-config mechanism for working with libflexio, such as `dependency('libflexio', required: true)` for `meson.build` or `CFLAGS := $(shell pkg-config --cflags 'libflexio')` for a makefile, there is no need to add the compilation flag as this is done automatically.

> ⚠ An example of adding these flags to launch DPACC can be found in the `doca_build_dpacc.sh` script in the `flexio-samples` package.

5.4.3.5.2.13 Flex IO Samples

This section describes samples based on the Flex IO SDK. These samples illustrate how to use the Flex IO API to configure and execute code on the DPA.

Running Flex IO Sample

The Flex IO SDK samples serve as a reference for building and running Flex IO-based DPA applications. They provide a collection of out-of-the-box working DPA applications that encompass the basic functionality of the Flex IO SDK.

Documentation

- Refer to DOCA Installation Guide for Linux for details on how to install BlueField-related software
- Refer to DOCA Troubleshooting for any issue you may encounter with the installation, compilation, or execution of DOCA samples

Minimal Requirements

The user must have the following installed:

- DOCA DPACC package
- DOCA RDMA package

- pkg-config package
- Python3 package
- Gcc with version 7.0 or higher
- Meson package with version 0.53.0 or higher
- Ninja package
- DOCA Flex IO SDK

Sample Structure

Each sample is situated in its own directory and is accompanied by a corresponding description in README files. Every sample comprises two applications:

- The first, located in the **dev** directory, is designed for DPA
- The second, found in the **host** directory, is intended for execution on the DPU or host in a Linux OS environment

Additionally, there is a **common** directory housing libraries for the examples. These libraries are further categorized into **dev** and **host** directories to facilitate linking with similar applications. Beyond containing functions and macros, these libraries also serve as illustrative examples for how to use them.

The list of the samples:

- **flexio_rpc** – sample demonstrating how to run RPC functions from DPA
- **packet_processor** – sample demonstrating how to process packets over DPA

Building the Samples

```
cd /opt/mellanox/flexio/samples/
./build.sh --check-compatibility --rebuild
```

Flex IO samples support building for multiple hardware models. Users can specify one or more hardware models to include in the build using the **--cpu** flag.

To build Flex IO samples for specific hardware models, use the following example commands:

```
cd /opt/mellanox/flexio/samples/
./build.sh --check-compatibility --rebuild --cpu bf3,cx8
```

This process generates builds optimized for the requested hardware models.

Samples

flexio_rpc

This sample application executes Flex IO with a remote process call.

The device program calculates the sum of 2 input parameters, prints the result, and copies the result back to the host application.

This sample demonstrates how applications are built (DPA and host), how to create processes and message streams, how to open the IBV device, and how to use RPC from the host to DPA function.

Compilation

```
cd /opt/mellanox/flexio/samples/
./build.sh --check-compatibility --rebuild
```

The output path:

```
/opt/mellanox/flexio/samples/build/flexio_rpc/host/flexio_rpc
```

Usage

```
<sample_root>/build/flexio_rpc/host/flexio_rpc <mlx5_device> <arg1> <arg2>
```

Where:

- **mlx5_device** – IBV device with DPA
- **arg1** – first numeric argument
- **arg2** – second numeric argument

Example:

```
$/opt/mellanox/flexio/samples/build/flexio_rpc/host/flexio_rpc mlx5_0 44 55
Welcome to 'Flex IO RPC' sample
Registered on device mlx5_0
/  2/Calculate: 44 + 55 = 99
Result: 99
Flex IO RPC sample is done
```

flexio_packet_process

This example demonstrates packet processing handling.

The device application implements a handler for **flexio_pp_dev** that receives packets from the network, swaps MAC addresses, inserts some text into the packet, and sends it back.

This allows the user to send UDP packets (with a packet length of 65 bytes) and check the content of returned packets. Additionally, the console displays the execution of packet processing, printing each new packet index. Device messaging operates in synchronous mode (i.e., each message from the device received by the host is output immediately).

This sample illustrates how applications work with libraries (DPA and host), how to create SQ, RQ, CQ, memory keys, and doorbell rings, how to create and use DPA memory buffers, how to use UAR, and how to create and run event handlers.

Compilation

```
cd /opt/mellanox/flexio/samples/
./build.sh --check-compatibility --rebuild
```

The output path:

```
/opt/mellanox/flexio/samples/build/packet_processor/host/flexio_packet_processor
```

Usage

```
<sample_root>/build/packet_processor/host/flexio_packet_processor <mlx5_device>
```

Where:

- **mlx5_device** – name of IB device with DPA
- **--nic-mode** – optional parameter indicating that the application is run from the host. If the application is run from DPU, then the parameter should not be used.

For example

```
$sudo /build/packet_processor/host/flexio_packet_processor mlx5_0
```

The application must run with root privileges.

Debug

When running the application, the output will display a debug token. This token can be used to connect to the DPA GDB server for debugging purposes.

```
$sudo build/packet_processor/host/flexio_packet_processor mlx5_0
Welcome to 'Flex IO SDK packet processing' sample app.
Use the token >>> 0x790478344927513b <<< for debugging
```

See DOCA DPA GDB Server Tool for instructions on using DPA GDB.

Running with Traffic

Run host-side sample:

```
$ cd <sample_root>
$ sudo ./build/packet_processor/host/flexio_packet_processor mlx5_0
```

Use another machine connected to the setup running the application. Bring the interface used as packet generator up:

```
$ sudo ifconfig my_interface up
```

Use **scapy** to run traffic to the device the application is running on:

```
$ python

>>> from scapy.all import *
>>> from scapy.layers.inet import IP, UDP, Ether

>>> sendp(Ether(src="02:42:7e:7f:eb:02", dst='52:54:00:79:db:d3')/IP()/UDP()/Raw(load="================12345678"),
iface="my_interface")
```

⚠ Source MAC must be same as above as the application defines a steering rule for it. Destination MAC can be anything.

⚠ The load should be kept the same as above, as the application looks for this pattern and changes it during processing.

⚠ Interface name should be changed to the interfaced used for traffic generation.

The packets can be viewed using **tcpdump**:

```
$ sudo tcpdump -i my_interface -en host 127.0.0.1 -X
```

Example output

```
Example output:
11:53:51.422075 02:42:7e:7f:eb:02 > 52:54:00:12:34:56, ethertype IPv4 (0x0800), length 65: 127.0.0.1.domain >
127.0.0.1.domain: 15677 op7+% [b2&3=0x3d3d] [15677a] [15677q] [15677n] [15677au][|domain]
        0x0000:  4500 0033 0001 0000 4011 7cb7 7f00 0001  E..3....@.|.....
        0x0010:  7f00 0001 0035 0035 001f 42c6 3d3d 3d3d  .....5.5..B.==== <-- Original data
        0x0020:  3d3d 3d3d 3d3d 3d3d 3d3d 3d31 3233 3435  ==========12345
        0x0030:  3637 38                                  678
11:53:51.700038 52:54:00:12:34:56 > 02:42:7e:7f:eb:02, ethertype IPv4 (0x0800), length 65: 127.0.0.1.domain >
127.0.0.1.domain: 26144 op8+% [b2&3=0x4576] [29728a] [25966q] [25701n] [28015au][|domain]
        0x0000:  4500 0033 0001 0000 4011 7cb7 7f00 0001  E..3....@.|.....
        0x0010:  7f00 0001 0035 0035 001f 42c6 6620 4576  .....5.5..B.f.Ev <-- Modified data
        0x0020:  656e 7420 6465 6d6f 2a2a 2a2a 2a2a 2a2a  ent.demo********
        0x0030:  2a2a 2a                                  ***
```

5.4.3.5.2.14 Flex IO SDK Packages

The Flex IO SDK contains the following packages:

- **flexio-sdk** – this package contains Flex IO SDK header and library files for both the host and device sides. The library files for the device side are divided by chip type (i.e., some are for NVIDIA® BlueField®-3, while others are for NVIDIA® ConnectX®-8).
- **flexio-samples** – this package contains Flex IO samples for both the host and device sides
- **dpa-gdbserver** – this package contains Flex IO binary files for both the host and device sides and is used as a gdbserver to debug DPA applications

5.4.3.5.3 DPA Application Authentication

DPA Application Authentication is currently only supported with statically linked libraries. Dynamically linked libraries are currently not supported.

This section provides instructions for developing, signing, and using authenticated BlueField-3 data-path accelerator (DPA) applications. It includes information on:

- Principles of root of trust and structures supporting it
- Device ownership transfer/claiming flow (i.e., how the user should configure the device so that it will authenticate the DPA applications coming from the user)
- Crypto signing flow and ELF file structure and tools supporting it

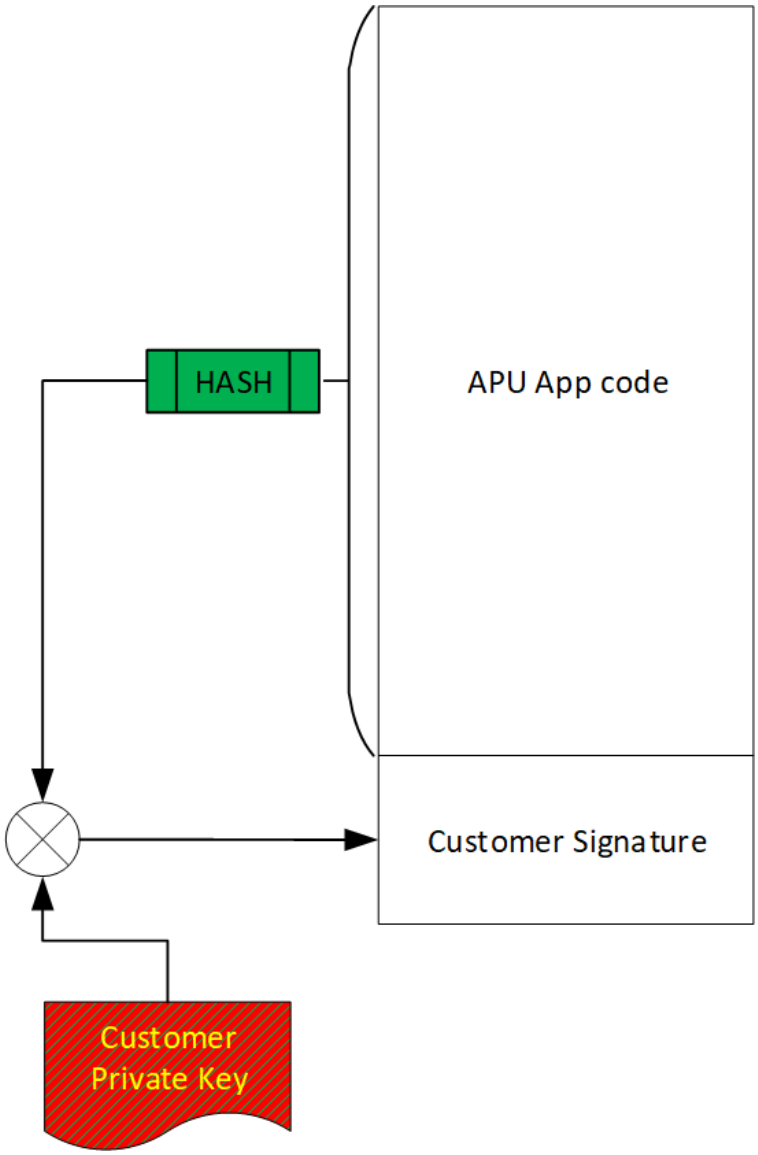
5.4.3.5.3.1 Root of Trust Principles

Signing of 3rd Party DPA App Code

NVIDIA® BlueField®-3 introduces the ability for customers/device owners to sign applications running on the DPA with their private key and have it authenticated by a device-embedded certificate chain. This provides the benefit of ensuring that only code permitted by the customer can run on the DPA. The customer can be any party writing code intended to run on the DPA (e.g., a cloud service provider, OEM, etc).

The following figure illustrates the signature of customer code. This signature will allow NVIDIA firmware to authenticate the source of the application's code.

Example of Customer DPA Code Signed by Customer for Authentication



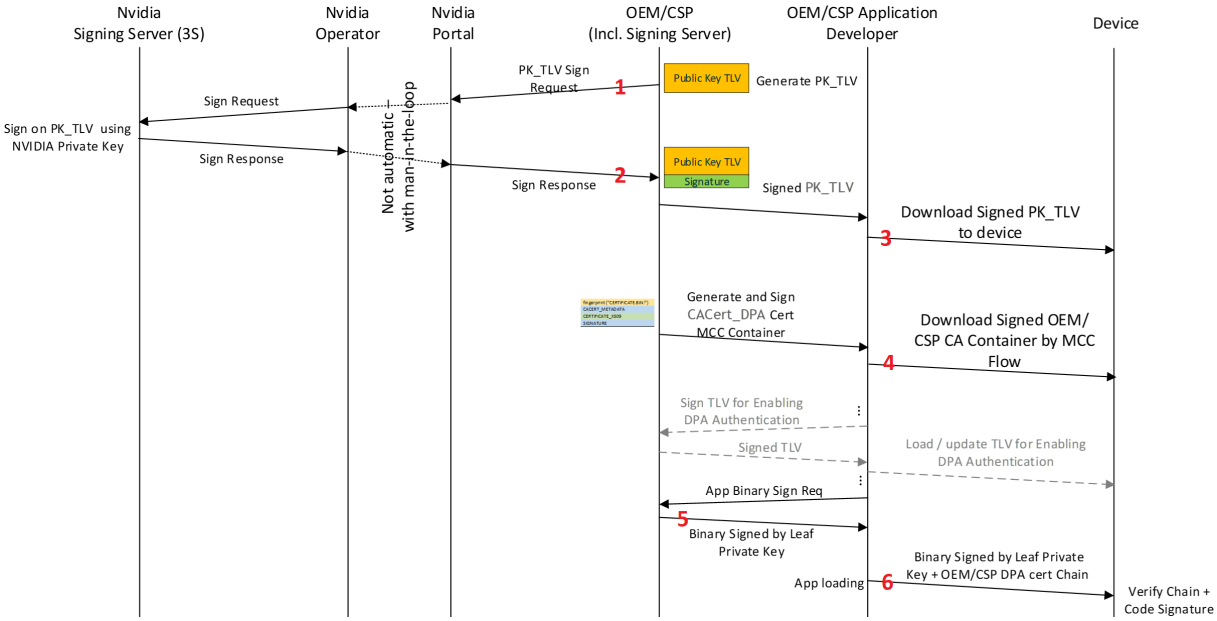
The high-level scheme is as follows (see figure "Loading of Customer Keys and CA Certificates and Provision of DPA Firmware to BlueField-3 Device"):

The numbers of these steps correspond to the numbers indicated in the figure below.

1. Customer provides NVIDIA Enterprise Support the public key for device ownership.
2. NVIDIA signs the customer's public key and sends it back to the customer.
3. Customer uploads the NVIDIA-signed public key to the device, enabling "Transfer of Ownership" to the customer (from NVIDIA).
4. Using the private key corresponding to the public key uploaded to the device, the customer can now enable DPA authentication and load the root certificate used for authentication of DPA App code.

5. DPA app code crypto-signed by the customer serves to authenticate the source of the app code.
   The public key used to authenticate the DPA app is provided as part of the certificate chain (leaf certificate), together with the DPA firmware image.
6. App code and the owner signature serves to authorize the app execution by the NVIDIA firmware (similar to NVIDIA own signature).

Loading of Customer Keys and CA Certificates and Provision of DPA Firmware to BlueField-3 Device



The following sections provide more details about this high-level process.

Verification of Authenticity of DPA App Code

Authentication of application firmware code before authorization to execute shall consist of validation of the customer certificate chain and customer signature using the customer's public key.

Public Keys (Infrastructure, Delivery, and Verification)

For the purposes of the authentication verification of the application firmware, the public key must be securely provided to the hardware. To do so, a secure Management Component Control (MCC) Flow shall be used. Using this, the content of the downloaded certificate is enveloped in an MCC Download Container and signed by NVIDIA Private Key.

The following is an example of how to use the MCC flow describes in detail the procedures, tools and structures supporting this (Section "DPA Development#Loading of CSP CA Certificates and Keys and Provisioning of DPA Firmware to Device" describes the high-level flow for this).

The following command burns the certificate container:

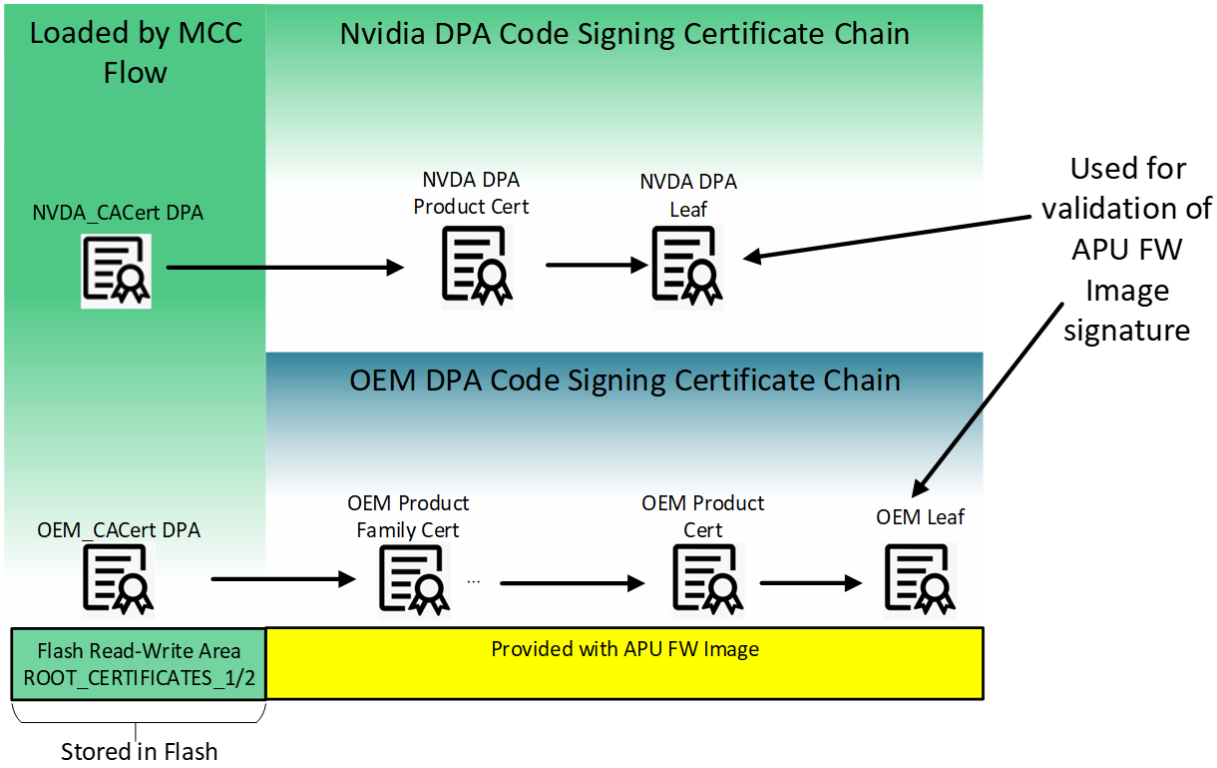
```
flint -d <mst device> -i <signed-certificate-container> burn
```

Two use cases are possible:

- The DPA application is developed internally in NVIDIA, and the authentication is based on internal NVIDIA keys and signing infrastructure
- The DPA application is developed by a customer, and the authentication is based on the customer certificate chain

In either case, the customer must download the relevant CA certificate to the device.

ROT Certificate Chain



This figure illustrates the build of the certificate chain used for validation of DPA app images. The leaf certificate of these chains is used to validate the DPA application supplied by the customer (with ROT from customer CA). The NVIDIA certificate chain for validation of DPA applications (built internally in NVIDIA) is structured in a very similar way. OEMDpaCert CA is the root CA which can be used by the customer to span their certificate chain up to the customer leaf certificate which is used for validating the signature of the application's image. Similarly, NVDADpaCert CA is the root CA used internally in NVIDIA to build the DPA certificate chain for validation of NVIDIA DPA apps.

Customer private keys must be kept secure and are the sole responsibility of the customer to maintain. It is recommended to have a set of keys ready and usable by customer for redundancy purposes. The whole customer certificate chain, including root CA and leaf, must not exceed 4 certificates.

The **NVDA_CACert_DPA** and **OEM_CACert_DPA** certificates are self-signed and trusted because they are loaded by the secure MCC flow and authenticated by the firmware.

The customer certificate chain beyond **OEM_CACert_DPA** is delivered with the DPA image, including the leaf certificate that is used for validating the cryptographic signature of the DPA firmware (see table "DPA Development#ELF Crypto Data Section Fields Description").

For more details on the certificates and their location in the flash, contact NVIDIA Enterprise Support to obtain the *Flash Application Note*. The rest of the certificate chain used for the DPA firmware authentication includes:

- For NVIDIA-signed images (e.g., figure "DPA Development#ROT Certificate Chain"): NVDA DPA root certificate (**NVDA_CACert_DPA** can be downloaded here)
- For customer-signed images (e.g., figure "DPA Development#ROT Certificate Chain"): Customer CA certificate, customer product, and customer leaf certificates

In both cases (NVIDIA internal and customer-signed) these parts of the certificate chain are attached to the DPA firmware image.

Loading of CSP CA Certificates and Keys and Provisioning of DPA Firmware to Device

The figure "DPA Development#Loading of Customer Keys and CA Certificates and Provision of DPA Firmware to BlueField-3 Device" shows, at high-level, the procedures for loading user public keys to the device, signing and loading of customer certificates MCC container, and downloading the DPA firmware images.

For clarity, the hierarchy of ROT validation is as follows:

1. Customer public key to be used for customer TLVs and **CACert_DPA** certificate validation, **PK_TLV** (i.e., **NV_LC_NV_PUBLIC_KEY**):
    a. For a device whose DPA authentication ability the customer wishes to enable for the first time, the customer must get it signed and authenticated by NVIDIA keys by reaching out to NVIDIA Enterprise Support. The complete flow is described in "DPA Development#Device Ownership Claiming Flow".
    b. After **PK_TLV** is loaded, it can be updated by authenticating the update using either the same **PK_TLV** . The complete flow is described in "DPA Development#Device Ownership Claiming Flow".
    c. Authentication of TLV for enabling/disabling DPA authentication is also validated by the **PK_TLV** . The complete flow is described in section "DPA Development#DPA Authentication Enablement".
2. Loading of CA certificate (**CACert_DPA**) to be used for DPA code validation. It is authenticated using the same **PK_TLV** .
   The complete flow is described in "DPA Development#Uploading DPA Root CA Certificate".
3. The public key in the leaf of the certificate chain anchored by **CACert_DPA** is used for authentication of the DPA firmware Image.
   The structure of the ELF file containing the DPA app and the certificate chain is described in "DPA Development#ELF File Structure".

A scalable and reliable infrastructure is required to support many users. The customer must also have an infrastructure to support their own code signing process according to their organization's security policy. Both matters are out of the scope of this document.

Trying to utilize the DPA signing flow in a firmware version prior to DOCA 2.2.0 is not supported.

Device Ownership Claiming Flow

NVIDIA networking devices allow the user of the device to customize the configurations, and in some cases change the behavior of the device. This set of available customizations is controlled by higher level NVIDIA configurations that come either as part of the device firmware or as a separate update file. To allow customers/device owners to change the set of available configurations and allowed behaviors, each device can have a device owner who is allowed to change the default behaviors and configurations of the device, and to change what configurations are exposed to the user.

The items controlled by the customer/device owner are:

- Device configurations: The customer/device owner can change the default value of any configuration available to users. They can also prevent users from changing the value.
- Trusted root certificates: The customer/device owner can control what root certificates the device trusts. These certificates control various behaviors (e.g., what 3rd party code the BlueField DPA accepts).

After the device has the public key of the owner, whenever an NVconfig file is signed with this key, one of two things must be true:

- The **nv_file_id** field in the NVconfig file must have the parameter **keep_same_priority** as **True**; or
- The NVconfig file must contain the public key itself (so the public key is rewritten to the device)

Otherwise, the public key is removed from the device, and as such will not accept files signed by the matching private key.

Detailed Ownership Claiming Flow

1. Customer generates a private-public key pair, and a UUID for the key pair.
    a. Generating UUID for the key pair:

    ```
    uuidgen -t
    ```

    Example output:

    ```
    77dd4ef0-c633-11ed-9e20-001dd8b744ff
    ```

    b. Generating an RSA key pair:

    ```
    openssl genrsa -out OEM.77dd4ef0-c633-11ed-9e20-001dd8b744ff.pem 4096
    ```

    Example output:

    ```
    Generating RSA private key, 2048 bit long modulus
    ...........+++
    ..............+++
    e is 65537 (0x10001)
    ```

c. Extracting the public key file from the RSA key pair:

```
openssl rsa -in OEM.77dd4ef0-c633-11ed-9e20-001dd8b744ff.pem -out OEM.77dd4ef0-
c633-11ed-9e20-001dd8b744ff.public -pubout -outform PEM
```

Output:

```
writing RSA key
```

The public key should look similar to the following:

```
-----BEGIN PUBLIC KEY-----
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAxfijde+27A3pQ7MoZnlm
mtpyuHO1JY9AUeKaHUXkWRiopL9Puswx1KcGfWJSNzlEPZRevTHraYlLQCru4ofr
W9NBE/qIwS2n7kiFwCCvZK6FKUUqZAuMJTpfuNtv9o4C4v0ZiX4TQqWDND8hy+1L
hPf3QLRiJ/ux4G6uHIFwENSwagershuKD0RI6BaZ1g9S9IxdXcD0vTdEuDPqQ0m4
CwEs/3xnksNRLUM+TiPEZoc5MoEoKyJv4GFbGttabhDCt5sr9RqAqTNUSDI9B0jr
XoQBQQpqRgYd31Q31Fhh3G9GjtoAcUQ6l0Gct3DXKFTAADV3Lyo1vjFNrOKUhdhT
pjDKzNmZAsxyIZI0buc24TCgj1yPyFboJtpnHmltyxfm9e+EJsdSIpRiX8YTWwkN
aIzNj08VswULwbKow5Gu5FFpE/uXDE3cXjLOUNnKihszFv4qkqsQjKaK4GszXge+
jfiEwsDKwS+cuWd9ihnyLrIWF23+OX0S5xjFXDJE8UthOD+3j3gGmP3kze1Iz2YP
Qvh3ITPRsqQltaiYh+CivqaCHC0voIMOP1ilAEZ/rW85pi6LA8EsudNMG2ELrUyl
SznBzZI/OxMk4qKx9nGgjaP2YjmcPw2Ffc9zZcwl57ThEOhlyS6w3E9xwBvZINLe
gMuOIWsu1FK3lIGxMSCUZQsCAwEAAQ==
-----END PUBLIC KEY-----
```

2. Customer provides NVIDIA Enterprise Support the public key for device ownership with its UUID.
3. NVIDIA generates a signed NVconfig file with this public key and sends it to the customer. This key may only be applied to devices that do not have a device ownership key installed yet.
4. Customer uses **mlxconfig** to install the OEM key on the needed devices.

```
mlxconfig -d /dev/mst/<dev> apply oem_public_key_nvconfig.bin
```

To check if the upload process has been successful, the customer can use **mlxconfig** to query the device and check if the new public key has been applied. The relevant parameters to query are **LC_NV_PUB_KEY_EXP**, **LC_NV_PUB_KEY_UUID**, and **LC_NV_PUB_KEY_0_255**.

Example of query command and expected response:

```
mlxconfig -d <dev>-e q LC_NV_PUB_KEY_0_255
```

Uploading DPA Root CA Certificate

After uploading a device ownership public key to the device, the owner can upload DPA root CA certificates to the device. There can be multiple DPA root CA certificates on the device at the same time.

If the owner wants to upload authenticated DPA apps developed by NVIDIA, they must upload the NVIDIA DPA root CA certificate found here.

If the owner wants to sign their own DPA apps, they must create another public-private key pair (in addition to the device ownership key pair), create a certificate containing the DPA root CA public key, and create a container with this certificate using **mlxdpa**.

To upload a signed container with a DPA root CA certificate to the device, **mlxdpa** must be used. This can be done both for either NVIDIA or customer-created certificates.

Generating DPA Root CA Certificate

1. Create a DER encoded certificate containing the public key used to validate DPA apps.
    a. Generating a certificate and a new key pair:

```
openssl req -x509 -newkey rsa:4096 -keyout OEM-DPA-root-CA-key.pem -outform der -out OEM-DPA-root-
CA-cert.crt -sha256 -nodes -subj "/C=XX/ST=OEMStateName/L=OEMCityName/O=OEMCompanyName/
OU=OEMCompanySectionName/CN=OEMCommonName" -days 3650
```

> ⚠ Both SHA256 and SHA512 are supported in cert. Only a RSA 4096 key is supported. The size of each certificate in DER format must be less than 1792 bytes.

    Output:

```
Generating a 4096 bit RSA private key
......++
.....................++
writing new private key to 'OEM-DPA-root-CA-key.pem'
-----
```

2. Create a container for the certificate and sign it with the device ownership private key.
    a. To create and add a container:

```
mlxdpa --cert_container_type add -c <cert.der> -o <path to output> --life_cycle_priority <Nvidia/
OEM/User> create_cert_container
```

    Output example:

```
Certificate container created successfully!
```

    b. To sign a container:

```
mlxdpa --cert_container <path to container> -p <key file> --keypair_uuid <uuid> --cert_uuid <uuid>
--life_cycle_priority <Nvidia/OEM/User> -o <path-to-output> sign_cert_container

Certificate container signed successfully!
```

Manually Signing Container

If the server holding the private key cannot run **mlxdpa**, it is possible to manually sign the certificate container and add the signature to the container. In that case, the following process should be followed:

1. Generate unsigned cert container:

```
mlxdpa --cert_container_type add -c <.DER-formatted-certificate> -o <unsigned-container-path> --
keypair_uuid <uuid> --cert_uuid <uuid> --life_cycle_priority OEM create_cert_container
```

2. Generate signature field header:

```
echo "90 01 02 0C 10 00 00 00 00 00 00 00" | xxd -r -p - <signature-header-path>
```

3. Generate signature of container (in whatever way, this is an example only):

```
openssl dgst -sha512 -sign <private-key-pem-file> -out <container-signature-path> <unsigned-container-path>
```

4. Concatenate unsigned container, signature header, and signature into one file:

```
cat <unsigned-container-path> <signature-header-path> <container-signature-path> > <signed-container-path>
```

Uploading Certificates

Upload each signed container containing the desired certificates for the device.

```
flint -d <dev> -i  <signed-container> -y b
```

Output example:

```
-I- Downloading FW ...
FSMST_INITIALIZE -   OK
Writing DIGITAL_CACERT_REMOVAL component -   OK
-I- Component FW burn finished successfully.
```

Removing Certificates

To remove root CA certificates from the device, the user must apply a certificate removal container signed by the device ownership private key.

There are two ways to remove certificates, either removing all certificates, or removing all installed certificates:

- Removing all root CA certificates from the device:
  a. Generate a signed container to remove all certificates.
     i. Created certificate container:

     ```
     mlxdpa --cert_container_type remove --remove_all_certs -o <path-to-container> --
     life_cycle_priority <Nvidia/OEM/User> create_cert_container
     ```

     Output example:

     ```
     Certificate container created successfully!
     ```

     ii. Sign certificate container:

     ```
     mlxdpa --cert_container <path-to-container> -p <key-file> --keypair_uuid <uuid> --
     life_cycle_priority <Nvidia/OEM/User> -o <path-to-signed-container> sign_cert_container
     ```

     Output example:

     ```
     Certificate container signed successfully!
     ```

  b. Apply the container to the device.

     ```
     flint -d <dev> -i <signed-container> -y b
     ```

     Output example:

     ```
     -I- Downloading FW ...
     FSMST_INITIALIZE -   OK
     Writing DIGITAL_CACERT_REMOVAL component -   OK
     -I- Component FW burn finished successfully.
     ```

- Removing specific root CA certificates according to their UUID:
  a. Generate a signed container to remove certificate based on UUID.
     i. Create the container.

```
mlxdpa --cert_container_type remove--cert_uuid <uuid> -o <path-to-container> --
life_cycle_priority <Nvidia/OEM/User> create_cert_container
```

Output example:

```
Certificate container created successfully!
```

ii. Sign the container:

```
mlxdpa --cert_container <path-to-container> -p <key-file> --keypair_uuid <uuid> --cert_uuid
<uuid> --life_cycle_priority <Nvidia/OEM/User> -o <path to output> sign_cert_container
```

Output example:

```
Certificate container signed successfully!
```

b. Apply the container to the device:

```
flint -d <dev> -i <signed container> -y b
```

Output:

```
-I- Downloading FW ...
FSMST_INITIALIZE -   OK
Writing DIGITAL_CACERT_REMOVAL component -   OK
-I- Component FW burn finished successfully.
```

DPA Authentication Enablement

After the device has a device ownership key and DPA root CA certificates installed, the owner of the device can enable DPA authentication. To do this, they must create an NVconfig file, sign it with the device ownership private key, and upload the NVconfig to the device.

Generating NVconfig Enabling DPA Authentication

1. Create XML with TLVs to enable DPA authentication.
   a. Get list of available TLVs for this device:

```
mlxconfig -d /dev/mst/<dev> gen_tlvs_file enable_dpa_auth.txt
```

Output:

```
Saving output...
Done!
```

Example part of the generated text file:

```
file_applicable_to                              0
file_comment                                    0
file_signature                                  0
file_dbg_fw_token_id                            0
file_cs_token_id                                0
file_btc_token_id                               0
file_mac_addr_list                              0
file_public_key                                 0
file_signature_4096_a                           0
file_signature_4096_b                           0
…
```

   b. Edit the text file to contain the following TLVs:

```
file_applicable_to                              1
nv_file_id_vendor                               1
nv_dpa_auth                                     1
```

c. Convert the `.txt` file to XML format with another mlxconfig command:

```
mlxconfig -a gen_xml_template enable_dpa_auth.txt enable_dpa_auth.xml
```

Output:

```
Saving output...
Done!
```

The generated `.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://www.mellanox.com/config">
<file_applicable_to ovr_en='1' rd_en='1' writer_id='0'>
     <psid></psid>
     <psid_branch></psid_branch>
 </file_applicable_to>

<nv_file_id_vendor ovr_en='1' rd_en='1' writer_id='0'>

    <!-- Legal Values: False/True -->
    <disable_override></disable_override>

    <!-- Legal Values: False/True -->
    <keep_same_priority></keep_same_priority>

    <!-- Legal Values: False/True -->
    <per_tlv_priority></per_tlv_priority>

    <!-- Legal Values: False/True -->
    <erase_lower_priority></erase_lower_priority>
    <file_version></file_version>
    <day></day>
    <month></month>
    <year></year>
    <seconds></seconds>
    <minutes></minutes>
    <hour></hour>

</nv_file_id_vendor>

<nv_dpa_auth ovr_en='1' rd_en='1' writer_id='0'>
    <!-- Legal Values: False/True -->
    <dpa_auth_en></dpa_auth_en>

</nv_dpa_auth>
</config>
```

d. Edit the XML file and add the information for each of the TLVs, as seen in the following example XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://www.mellanox.com/config">

<file_applicable_to ovr_en='0' rd_en='1' writer_id='0'>
    <psid>TODO</psid>
    <psid_branch>TODO</psid_branch>
</file_applicable_to>

<nv_file_id_vendor ovr_en='0' rd_en='1' writer_id='0'>
    <disable_override>False</disable_override>
    <keep_same_priority>True</keep_same_priority>
    <per_tlv_priority>False</per_tlv_priority>
    <erase_lower_priority>False</erase_lower_priority>
    <file_version>TODO</file_version>
    <day>TODO</day>
    <month>TODO</month>
    <year>TODO</year>
    <seconds>TODO</seconds>
    <minutes>TODO</minutes>
    <hour>TODO</hour>
</nv_file_id_vendor>

<nv_dpa_auth ovr_en='0' rd_en='1' writer_id='0'>
    <dpa_auth_en>True</dpa_auth_en>
</nv_dpa_auth>
</config>
```

> ⚠ In **nv_file_id_vendor**, **keep_same_priority** must be **True** to avoid removing the ownership public key from the device. More information they can be found in section "Device Ownership Claiming Flow".

> ⚠ The **ovr_en** should be set to 0. This can ignore user priority changing **nv_dpa_auth**.

2. Convert XML file to binary NVconfig file and sign it using **mlxconfig**:

```
mlxconfig -p OEM.77dd4ef0-c633-11ed-9e20-001dd8b744ff.pem -u 77dd4ef0-c633-11ed-9e20-001dd8b744ff
create_conf enable_dpa_auth.xml enable_dpa_auth.bin
```

Output of **create_conf** command:

```
Saving output...
Done!
```

3. Upload NVconfig file to device by writing the file to the device:

```
mlxconfig -d /dev/mst/<dev> apply enable_dpa_auth.bin
```

Output:

```
Saving output...
Done!
```

4. Verify that the device has DPA authentication enabled by reading the status of DPA authentication from the device:

```
mlxconfig -d /dev/mst/<dev> -e q DPA_AUTHENTICATION
```

Output:

```
Device #1:
----------

Device type:    BlueField3
…
…
Configurations:                                 Default         Current         Next Boot
 RO     DPA_AUTHENTICATION                      True(1)         True(1)         True(1)
```

The DPU's factory default setting is configured with **dpa_auth_en=0** (i.e., DPA applications can run without authentication). To prevent configuration change by any user, it is strongly recommended for the customer to generate and install NVconfig with **dpa_auth_en=0/1**, according to their preferences, with **ovr_en=0**.

Manually Signing NVconfig File

If the server holding the private key cannot run mlxconfig, it is possible to manually sign the binary NVconfig file and add the signature to the file. In this case, the following process should be followed instead of step 2:

1. Generate unsigned NVconfig bin file from the XML file:

```
mlxconfig create_conf <xml-nvconfig-path> <unsigned-nvconfig-path>
```

2. Generate random UUID for signature:

```
uuidgen -r | xxd -r -p - <signature-uuid-path>
```

3. Generate signature of NVconfig bin file (in whatever way, this is an example only):

```
openssl dgst -sha512 -sign <private-key-pem-file> -out <nvconfig-signature-path> <unsigned-nvconfig-path>
```

4. Split the signature into two parts:

```
head -c 256 <nvconfig-signature-path> > <signature-part-1-path> && tail -c 256 <nvconfig-signature-path> >
<signature-part-2-path>
```

5. Add signing key UUID:

```
echo "<signing-key-UUID>" | xxd -r -p - <signing-key-uuid-path>
```

   Use the signing key UUID, which must have a length of exactly 16 bytes, in a format like **aa9c8c2f-8b29-4e92-9b76-2429447620e0**.

6. Generate headers for signature struct:

```
echo "03 00 01 20 06 00 00 0B 00 00 00 00" | xxd -r -p - <signature-1-header-path>
echo "03 00 01 20 06 00 00 0C 00 00 00 00" | xxd -r -p - <signature-2-header-path>
```

7. Concatenate everything:

```
cat <unsigned-nvconfig-path> <signature-1-header-path> <signature-uuid-path> <signing-key-uuid-path>
<signature-part-1-path> <signature-2-header-path> <signature-uuid-path> <signing-key-uuid-path> <signature-
part-2-path> > <signed-nvconfig-path>
```

Device Ownership Transfer

The device owner may change the device ownership key to change the owner of the device or to remove the owner altogether.

First Installation

To install the first **OEM_PUBLIC_KEY** on the device, the user must upload an NVCONFIG file signed by NVIDIA. This file would contain the 3 **FILE_OEM_PUBLIC_KEY** TLVs of the current user.

Removing Device Ownership Key

Before completely removing the device ownership key, it is recommended that the device owner reverts any changes made to the device, as these changes cannot be undone afterward. Specifically, the root CA certificates installed by the owner should be removed.

1. To remove device ownership key completely, follow the steps in section "DPA Development#Generating NVconfig Enabling DPA Authentication" to create an XML file with TLVs.
2. Edit the XML file to contain the following TLVs:

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://www.mellanox.com/config">

<file_applicable_to ovr_en='0' rd_en='1' writer_id='0'>
```

```
    <psid> MT_0000000911</psid>
    <psid_branch> </psid_branch>
</file_applicable_to>

<nv_file_id_vendor ovr_en='0' rd_en='1' writer_id='0'>
    <disable_override>False</disable_override>
    <keep_same_priority>False</keep_same_priority>
    <per_tlv_priority>False</per_tlv_priority>
    <erase_lower_priority>False</erase_lower_priority>
    <file_version>0</file_version>
    <day>17</day>
    <month>7</month>
    <year>7e7</year>
    <seconds>1</seconds>
    <minutes>e</minutes>
    <hour>15</hour>
</nv_file_id_vendor>
</config>
```

The TLVs in this file are the only TLVs that will have OEM priority after this file is applied, and as the device ownership key will no longer be on the device, the OEM will no longer be able to change the TLVs. To have OEM priority TLVs on the device after removing the device ownership key, add to this XML any TLV that must stay as default on the device.

3. Convert the XML file to a binary NVconfig TLV file signed by the device ownership key as described in section "DPA Development#Generating NVconfig Enabling DPA Authentication".
4. Apply the NVconfig file to the device as described in section "DPA Development#Generating NVconfig Enabling DPA Authentication".

Changing Device Ownership Key

To transfer ownership of the device to another entity, the previous owner can change the device ownership public key to the public key of the new owner.

To do this, they can use an NVconfig file, and include in it the following TLVs:

```
<nv_ls_nv_public_key_0 ovr_en='0' rd_en='1' writer_id='0'>
    <public_key_exp>65537</public_key_exp>
    <keypair_uuid>77dd4ef0-c633-11ed-9e20-001dd8b744ff</keypair_uuid>
</nv_ls_nv_public_key_0>

<nv_ls_nv_public_key_1 ovr_en='0' rd_en='1' writer_id='0'>
    <key>
    c5:f8:a3:75:ef:b6:ec:0d:e9:43:b3:28:66:79:
    66:9a:da:72:b8:73:b5:25:8f:40:51:e2:9a:1d:45:
    e4:59:18:a8:a4:bf:4f:ba:cc:31:d4:a7:06:7d:62:
    52:37:39:44:3d:94:5e:bd:31:eb:69:89:4b:40:2a:
    ee:e2:87:eb:5b:d3:41:13:fa:88:c1:2d:a7:ee:48:
    85:c0:20:af:64:ae:85:29:45:2a:64:0b:8c:25:3a:
    5f:b8:db:6f:f6:8e:02:e2:fd:19:89:7e:13:42:a5:
    83:34:3f:21:cb:ed:4b:84:f7:f7:40:b4:62:27:fb:
    b1:e0:6e:ae:1c:81:70:10:d4:b0:6a:07:ab:b2:1b:
    8a:0f:44:48:e8:16:99:d6:0f:52:f4:8c:5d:5d:c0:
    f4:bd:37:44:b8:33:ea:43:49:b8:0b:01:2c:ff:7c:
    67:92:c3:51:2d:43:3e:4e:23:c4:66:87:39:32:81:
    28:2b:22:6f:e0:61:5b:1a:db:5a:6e:10:c2:b7:9b:
    2b:f5:1a:80:a9:33:54:48:32:3d:07:48:eb:5e:84:
    01:41:0a:6a:46:06:1d:de:54:37:d4:58:61:dc:6f:
    46:8e:da:00:71:44:3a:97:41:9c:b7:70:d7:28:54:
    c0:00:35:77:2f:2a:35:be:31:4d:ac:e2:94:85:d8:
    53:a6:
    </key>
</nv_ls_nv_public_key_1>

<nv_ls_nv_public_key_2 ovr_en='0' rd_en='1' writer_id='0'>
    <key>
    30:ca:cc:d9:99:02:cc:72:21:92:34:6e:e7:
    36:e1:30:a0:8f:5c:8f:c8:56:e8:26:da:67:1e:69:
    6d:cb:17:e6:f5:ef:84:26:c7:52:22:94:62:5f:c6:
    13:5b:09:0d:68:8c:cd:8f:4f:15:b3:05:0b:c1:b2:
    a8:c3:91:ae:e4:51:69:13:fb:97:0c:4d:dc:5e:32:
    ce:50:d9:ca:8a:1b:33:16:fe:2a:92:ab:10:8c:a6:
    8a:e0:6b:33:5e:07:be:8d:f8:84:c2:c0:ca:c1:2f:
    9c:b9:67:7d:8a:19:f2:2e:b2:16:17:6d:fe:39:7d:
    12:e7:18:c5:5c:32:44:f1:4b:61:38:3f:b7:8f:78:
    06:98:fd:e4:cd:ed:48:cf:66:0f:42:f8:77:21:33:
    d1:b2:a4:25:b5:a8:98:87:e0:a2:be:a6:82:1c:2d:
    2f:a0:83:0e:3f:58:a5:00:46:7f:ad:6f:39:a6:2e:
    8b:03:c1:2c:b9:d3:4c:1b:61:0b:ad:4c:a5:4b:39:
    c1:cd:92:3f:3b:13:24:e2:a2:b1:f6:71:a0:8d:a3:
    f6:62:39:9c:3f:0d:85:7d:cf:73:65:cc:25:e7:b4:
    e1:10:e8:65:c9:2e:b0:dc:4f:71:c0:1b:d9:20:d2:
    de:80:cb:8e:21:6b:2e:d4:52:b7:94:81:b1:31:20:
    94:65:0b
    </key>
```

```
</nv_ls_nv_public_key_2>
```

If the transfer is internal, the owner should set **keep_same_priority=True in nv_file_id_vendor** TLV and only include the 3 **nv_ls_nv_public_key_*** TLVs, **file_applicable_to** and **nv_file_id_vendor** TLVs in the NVconfig file.

If the transfer is to another OEM/CSP, the owner should clean the device (similarly to removing the device ownership key) and set **keep_same_priority=False** in **nv_file_id_vendor** TLV.

5.4.3.5.3.2 ELF File Structure

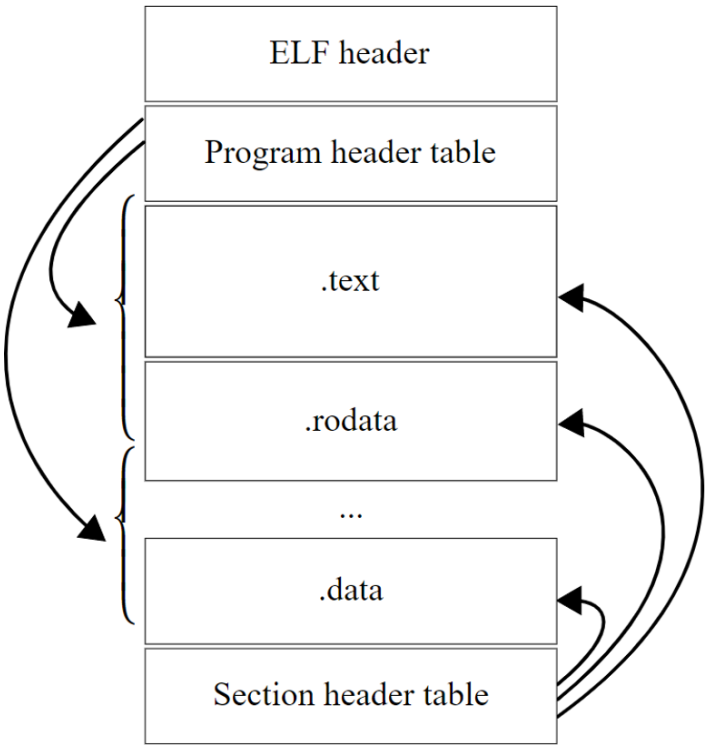
For maximal firmware code reuse, the format of the DPA image loaded from driver should be the same as for the file loaded from flash. As for files loaded from the host, ELF is the default file format. This is chosen as the format for the DPA image, both for flash and for files loaded from the host.

The following figure shows, schematically, a generic ELF file structure.

To support DPA Code authentication additional information needs to be presented to firmware. This info must include:

- Cryptographic signature of the DPA code
- Customer certificate chain including a Leaf Certificate with the public key to be used for signature validation (as described in section "DPA Development#Public Keys (Infrastructure, Delivery, and Verification)")

ELF File Structure Schematic



Crypto Signing Flow

The host ELF includes parts which run on the host, and those that run on DPA. DPA code files are incorporated in the "big" host ELF as binaries. Each host file may include several DPA applications.

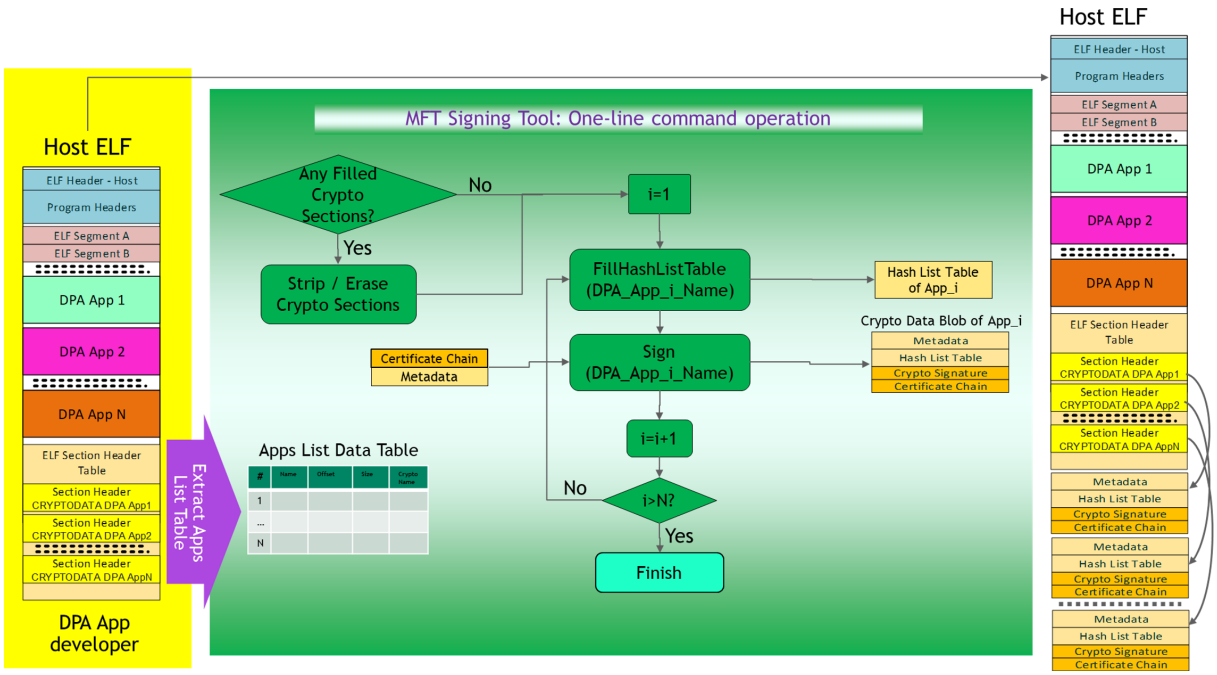
When it is required to sign the DPA applications, the following steps need to be performed by the MFT Signing Tool (also see figure "Crypto Signing Flow"):

1. Using ELF manipulation library APIs of DPACC, extract Apps List Table
    a. Input – host ELF
    b. Output – apps list data table to include:
        i. DPA app index
        ii. DPA app name
        iii. Offset in host ELF
        iv. Size of app
        v. Name of corresponding crypto data section
        For each DPA application (from i=1 to i=N, N- number of DPA apps in the host ELF) run steps 2 and 3.
2. Fill hash list table:
    - Input: **`Dpa_App_i`**
    - Output: Hash list table
3. Sign the crypto data:
    - Input: {Metadata, Hash List Table}, key handle (e.g., UUID from leaf of the Certificate Chain)
    - Output: **`Crypto_Data`** "Blob", including: Metadata, Hash List Table, Crypto Signature, Certificate Chain
4. Add crypto data section to host ELF:
    - Inputs: Host ELF, crypto data section name to use
    - Output: File name of host ELF with signature added

The structures used in the flow (hash list table, metadata, etc.) are described in sections "DPA Development#ELF Crypto Data Section Content" and "DPA Development#Hash List Table Layout".

Signing the crypto data may be done using a signing server or a locally stored key.

Crypto Signing Flow



ELF Cryptographic Data Section

This figure shows, schematically, the layout of the cryptographic data section, and the following subsections provide details about the ELF section header and the rest of the structures.

ELF Cryptographic Data Section Layout

Crypto Data ELF Section Header

Defined according to the ELF section header format.

ELF Section Header

Name	Offset	Range	Description
sh_name	0x0	4B	&("Cryptographic Data Section DPA App X") An offset to a string (in the .shstrtab section of ELF) which represents the name of this section
sh_type	0x4	4B	0x70000666 SHT_CRYPTODATA – the section is proprietary and holds crypto information defined in this document
sh_flags	0X8	8B	0 – no flags
sh_addr	0x10	8B	Virtual address of the section in memory, for sections that are loaded
sh_offset	0x18	8B	Offset of the section in the file image
sh_size	0x20	8B	Size in bytes of the section in the file image. Depends on the content (e.g., presence and type of public key certificate chain and signature).
sh_link	0x28	4B	0 – =SHN_UNDEF, no link information
sh_info	0x2C	4B	0 – no extra information about the section
sh_addralign	0x30	8B	Contains the required alignment of the section. This field must be a power of two.
sh_entsize	0x38	8B	0
	0x40		End of section header (size)

ELF Crypto Data Section Content

ELF Crypto Data Section Fields Description

Name	Offset	Range	Description
metadata_version	0x0	15:0	Version metadata structure format. Initial version is 0.
Reserved (DPA_fw_type)	0x4	15:8	Reserved
Reserved	0x8	31:0	Reserved
Reserved	0xC	31:0	Reserved. Shall be set to all zeros.
Reserved	0x10	16B	Reserved. Shall be set to all zeros.
Reserved	0x20	4 bytes	Reserved. Shall be set to all zeros.
Reserved	0x24	24B	Reserved. Shall be set to all zeros.
signature_type	0x3c	15:0	Signature Type. Only relevant for signed firmware: • 0, 1 – Reserved • 2 – RSA_ SHA_512 • >3 – Reserved
Hash List Table	0x40	HashTable Length	

Name	Offset	Range	Description
Crypto Signature	0x40 + HashTableLength	Signature_Length	Signature_Length depends on the signature_type.
Certificate_Chain	0x40 + HashTableLength + Signature_Length	CrtChain_Length	Structure given the table under section "DPA Development#Certificate Chain Layout".
Padding			FF-padding to align the full size of the data to multiples of DWords (DWs)

The full length of the ELF crypto data section shall be a multiple of DWs (due to firmware legacy implementation). Thus, the MFT (as part of the flow described in figure "Crypto Signing Flow") shall add FF-padding for this structure to align to multiple of DW.

Hash List Table Layout

This table specifies the hash table layout (proposal).

The table contains two parts:

- The 1st part corresponds to the segments of the ELF file, as referenced by the Program Header Table of the EFL file
- The 2nd part corresponds to the sections of the ELF file, as referenced by the Section Header Table

The hash algorithm to be used is SHA-256.

Hash List Table Layout (Proposal)

Name	Offset	Range	Description
Hash Table Magic Pattern	0x0	8 bytes	ASCII "HASHLIST' string: 0x0: 31:24 – "H", 23:16 – "A", 15:8 – "S", 7:0 – "H" 0x4: 31:24 – "L", 23:16 – "I", 15:8 – "S", 7:0 – "T"
Number of Entries – Segments	0x8	7:0	Number of entries in Hashes Segments part, N_Segments.
Reserved	0x8	31:8	Reserved
Number of Entries – Sections	0xc	7:0	Number of entries in Hashes Sections part, N_Sections. Minimum – 0
Reserved	0xc	31:8	Reserved
Reserved	0x10	16 bytes	Reserved
DPA Application ELF Hash	0x20	32 bytes	Hash of the full ELF App file
ELF Header Hash	0x40	32 bytes	Hash of the ELF Header

Name	Offset	Range	Description
Program Header Hash	0x60	32 bytes	Hash of the program header
Hash of 1st Segment referenced in the Program Header Table	0x80	32 bytes	Hash of 1st segment referenced in the Program Header Table
Hash of 2nd Segment referenced in the Program Header Table	0xA0	32 bytes	Hash of 2nd Segment referenced in the Program Header Table
.....	.....	.....	.....
Hash of N_Segments (last) Segment referenced in the Program Header Table	0x60 + N_Segments*0x20	32 bytes	Hash of 2nd segment referenced in the Program Header Table
Section Header Table Hash	0x80 + N_Segments*0x20	32 bytes	Hash of the Section Header Table
Hash of 1st Section referenced in the Section Header Table	+ 0x20	32 bytes	Hash of 1st section referenced in the Section Header Table
Hash of 2nd Section referenced in the Section Header Table	+ 0x20	32 bytes	Hash of 2nd section referenced in the Section Header Table
.....	.....	.....	.....
Hash of N_Sections (last) Section referenced in the Section Header Table	+ 0x20	32 bytes	Hash of N_Sections (last) section referenced in the Section Header Table

The 32-bytes hash fields of different sections/segments in the previous table shall follow Big-Endian convention, as illustrated here:

Hash Fields (Big Endian) Bytes Alignment

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	offset
hash[0]								hash[1]								hash[2]								hash[3]								0h
hash[4]								hash[5]								hash[6]								hash[7]								4h
...								...								...								...								...
hash[28]								hash[29]								hash[30]								hash[31]								1Ch

Certificate Chain Layout

The following table specifies the certificate chain layout. The leaf (the last certificate) of the chain is used as the public key for authentication of the DPA code. This structure is aligned with the certificate chain layout as defined in the *Flash Application Note*.

Certificate Chain Layout

Name	Offset	Range	Description

Type	0x0	3:0	Chain type. Shall be set to 1. 3rd party code authentication certificate chain.
Count	0x0	7:4	Number of certificates in this chain
Length	0x0	23:8	Total length of the certificate chain, in bytes, including all fields in this table
Reserved	0x4	31:0	31:0 – Reserved
CRC	0x8	15:0	The CRC of the header, for header integrity check, covering DWs in 0x0, 0x4
Certificates	0xC-0x1000		One or more ASN.1 DER-encoded X509v3 certificates. The ASN.1 DER encoding of each individual certificate can be analyzed to determine its length. The certificates shall be listed in hierarchical order, with the leaf certificate being the last on the list.

5.4.3.5.4 Distribution of Execution Units Across Multiple Applications

Managing DPA execution units across multiple applications is challenging due to the variability in available units and running applications. To address this, administrators can define a subset of execution units (EUs) for each application to run on, as described in the Single Point of Resource Distribution page.

This approach is supported by the following applications:

- DOCA PCC application
- DPA basic initiator sample

To implement this, create a YAML file describing the DPA resource allocation before running the application.

Applications receive their EU allocation through the following command-line arguments:

- **`--dpa-resources`** – Path to the YAML file containing DPA resource definitions
- **`--dpa-app-key`** – Name of the application as defined in the resources YAML file

When properly configured, execution units are assigned to the application according to the YAML file's resource specifications.

⚠ Execution Unit Groups (EUGs) are currently unsupported.

⚠ The PCC application still accepts the **`--threads`** command line argument for backward compatibility, but it cannot be used alongside the new DPA resources flags. This argument will be removed in future versions.

5.4.3.5.5 Known Limitations

5.4.3.5.5.1 Supported Devices

- BlueField-3 based DPUs
- ConnectX-8 based NICs
- Only ECPF/PF may be used for loading DPA applications

5.4.3.5.5.2 Supported Host OS

- Windows is not supported

5.4.3.5.5.3 Supported SDKs

- DOCA Flex IO at GA level

5.4.3.5.5.4 Toolchain

- DPA image-signing and signature-verification are not currently supported

5.4.3.5.5.5 Flex IO

- When **flexio_dev_outbox_config_uar_extension** API is called with a **device_id** parameter different than PF/ECPF ID (i.e., move to SF/VF outbox) and the APIs **flexio_dev_yield()**, **flexio_dev_print()**, or **flexio_dev_msg()** are called, then when either of those 3 APIs return, the user cannot work with the SF/VF queues.
- Flex IO SDK tracer might suffer from data loss and ordering issues when working under high load (multithread/high rate).
- Functions that copy data from the host to the DPA use WQEs and may be blocked for an extended period under heavy high-priority traffic. To mitigate this, it is recommended to configure a 99/1 weight between high-priority traffic and TC7.
    - The following Flex IO SDK API functions are directly affected:
        - **flexio_memcpy()**
        - **flexio_host2dev_memcpy()**
        - **flexio_buf_dev_memset()**
        - **flexio_copy_from_host()**
    - Additionally, the following API functions use copy operations and are indirectly affected:
        - **flexio_cq_create()**
        - **flexio_event_handler_create()**
        - **flexio_event_handler_run()**
        - **flexio_process_call()**
        - **flexio_cmdq_create()**
        - **flexio_log_dev_init()**

5.4.3.6 DOCA DPA

5.4.3.6.1 Introduction

> ⚠ The quality status of DOCA libraries is listed here.

This chapter provides an overview and configuration instructions for DOCA DPA API.

The DOCA DPA libraries offer a programming model for offloading communication-centric user code to run on the DPA processor on NVIDIA® BlueField®-3 networking platform. DOCA DPA provides a high-level programming interface to the DPA processor.

DOCA DPA offers:

- Full control over DPA threads –
  - The user can control the thread function (kernel) that runs on DPA and their placement on DPA EUs
  - The user can associate a DPA thread with a DPA Completion Context. When the completion context receives a notification, the DPA thread is scheduled.
- Abstraction to allow a DPA thread to issue asynchronous operations
- Abstraction to execute a blocking one-time call from host application to execute the kernel on the DPA from the host application (RPC)
- Abstraction for memory services
- Abstraction for remote communication primitives (integrated with remote event signaling)
- Full control on execution-ordering and notifications/synchronization of the DPA and host/ Target BlueField
- A set of debugging APIs that allow diagnosing and troubleshooting any issue on the device, as well as accessing real-time information from the running application
- C API for application developers

DPACC is used to compile and link kernels with DOCA DPA device libraries to get DPA applications that can be loaded from the host program to execute on the DPA (similar to CUDA usage with NVCC). For more information on DPACC, refer to the *DOCA DPACC Compiler*.

5.4.3.6.2 Prerequisites

DOCA DPA applications can run either on the host or on the Target BlueField. Running on the host machine requires EU pre-configuration using the `dpaeumgmt` tool. For more information, please refer to *NVIDIA DOCA DPA EU Management Tool*.

5.4.3.6.3 Library Changes From Previous Releases

5.4.3.6.3.1 Changes in 2.9.0

The following subsection(s) detail the `doca_dpa` library updates in version 2.9.0.

Added

- `doca_dpa_dev.h`

- Added **enum doca_dpa_dev_submit_flag** to be used in the new parameter **uint32_t flags** in DPA RDMA device and DPA memcpy APIs

Removed

- **doca_dpa_dev_rdma.h**
  - **void doca_dpa_dev_rdma_synchronize(doca_dpa_dev_rdma_t rdma);**
  - **void doca_dpa_dev_rdma_srq_post_receive(doca_dpa_dev_rdma_srq_t rdma_srq, uint32_t next_index);**
  - **void doca_dpa_dev_rdma_srq_receive_ack(doca_dpa_dev_rdma_srq_t rdma_srq, uint32_t num_acked);**

Changed

- **doca_dpa_dev_rdma.h**
  - Added **uint32_t connection_id** parameter to all DPA RDMA device APIs
  - Changed **uint32_t completion_requested** parameter to **uint32_t flags** in all DPA RDMA APIs
- **doca_dpa_dev_buf.h**
  - Changed **uint32_t completion_requested** parameter to **uint32_t flags** in all DPA memcpy APIs

5.4.3.6.3.2 Changes in 3.5.0

The following subsection(s) detail the **doca_dpa** library updates in version 3.5.0.

Added

doca_dpa.h

- DPA EU group affinity APIs: **doca_dpa_eu_group_affinity_create()**, **doca_dpa_eu_group_affinity_destroy()**, **doca_dpa_eu_group_affinity_set()**, **doca_dpa_eu_group_affinity_clear()**, **doca_dpa_eu_group_affinity_get()**
- DPA thread group-affinity APIs: **doca_dpa_thread_set_group_affinity()**, **doca_dpa_thread_get_group_affinity()**
- DPA resources EU group query APIs: **doca_dpa_resources_get_num_eu_groups()**, **doca_dpa_resources_get_eu_groups()**
- Validation-resources APIs: **doca_dpa_set_validation_resources()**, **doca_dpa_get_validation_resources()**

Changed

doca_dpa.h

- DPA thread affinity now supports relaxed, fixed (single EU), and group (EU group) modes
- DPA Resources SPRD parsing now supports EU group allocation (**number_of_groups**, **groups**)

5.4.3.6.4 Development Flow

DOCA enables developers to program the DPA processor using both DOCA DPA libraries and a suite of other tools (mainly DPACC).

The following are the main steps to start DPA offload programming:

1. Write DPA device code, or kernels, (**.c** files) with:
   - The **__dpa_global__** keyword before DPA thread function (see "DOCA DPA#Examples" section)
   - The **__dpa_rpc__** keyword before RPC function (see "DOCA DPA#Examples" section)
2. Use DPACC to build a DPA program (i.e., a host library which contains an embedded device executable). Inputs for DPACC are:
   - Kernels from the previous step
   - DOCA DPA device libraries
3. Build host executable using a host compiler. Inputs for the host compiler are:
   - DPA program from the previous step
   - User host application source files
   - DOCA DPA host library

DPACC is provided by the DOCA SDK installation. For more information, please refer to the *DOCA DPACC Compiler*.

5.4.3.6.4.1 DPA Queries

- Before invoking the DPA API, make sure that DPA is indeed supported on the relevant device. The API which checks whether a device supports DPA is:

```
doca_error_t doca_devinfo_get_is_dpa_supported(const struct doca_devinfo *devinfo)
```

> Only if this call returns **DOCA_SUCCESS** can the user invoke DOCA DPA API on the device.

- To use a valid EU ID for the DPA EU Affinity of a DPA thread, use the following APIs to query EU ID and core valid values:

```
doca_error_t doca_dpa_get_core_num(struct doca_dpa *dpa, unsigned int *num_cores)
doca_error_t doca_dpa_get_num_eus_per_core(struct doca_dpa *dpa, unsigned int *eus_per_core)
doca_error_t doca_dpa_get_total_num_eus_available(struct doca_dpa *dpa, unsigned int *total_num_eus)
```

- There is a limitation on the maximum number of DPA threads that can run a single kernel. This can be retrieved by calling the host API:

```
doca_error_t doca_dpa_get_max_threads_per_kernel(struct doca_dpa *dpa, unsigned int *value)
```

- Each kernel launched into the DPA has a maximum DPA kernel time alive supported by the DOCA device. During the DPA process creation, the host application should read the cap and adjust the parameters of the DPA threads, such as batch size and loop iterations, to ensure they remain within the specified limits.

```
doca_error_t doca_dpa_cap_get_max_kernel_time_alive_supported(const struct doca_devinfo *devinfo, uint64_t
*value)
```

> ⚠ If the kernel execution time on the DPA exceeds this maximum runtime limit, it may be terminated and cause a fatal error. To recover, the application must destroy the DPA context and create a new one.

> ⚠ DEPRECATED API, will be removed in future release.

- Each kernel launched into the DPA has a maximum runtime limit. This can be retrieved by calling the host API:

```
doca_error_t doca_dpa_get_kernel_max_run_time(struct doca_dpa *dpa, unsigned long long *value)
```

5.4.3.6.4.2 Initialization

The DPA context encapsulates the DPA device and a DPA process (program). Within this context, the application creates various DPA SDK objects and controls them. After verifying DPA is supported for the chosen device, the DPA context is created.

Use the following host-side APIs to create/set the DPA context and it is expected to be the first programming step:

- To create/destroy DPA context:

```
doca_error_t doca_dpa_create(struct doca_dev *dev, struct doca_dpa **dpa)
doca_error_t doca_dpa_destroy(struct doca_dpa *dpa)
```

> ⚠ Please note that the **dev** used to create the "base" DPA context must be a PF DOCA device.

- To start/stop DPA context:

```
doca_error_t doca_dpa_start(struct doca_dpa *dpa)
doca_error_t doca_dpa_stop(struct doca_dpa *dpa)
```

5.4.3.6.4.3 Interface to DPACC

DPA Application

To associate a DPA program (app) with a DPA context, use the following host-side APIs:

```
doca_error_t doca_dpa_set_app(struct doca_dpa *dpa, struct doca_dpa_app *app)
doca_error_t doca_dpa_get_app(struct doca_dpa *dpa, struct doca_dpa_app **app)
doca_error_t doca_dpa_app_get_name(struct doca_dpa_app *app, char *app_name, size_t *app_name_len)
```

The **app** variable name used in **doca_dpa_set_app()** API must be the token passed to DPACC **--app-name** parameter.

Example (Pseudo Code)

For example, when using the following **dpacc** command line:

```
dpacc \
    kernels.c \
    -o dpa_program.a \
    -hostcc=gcc \
    -hostcc-options="..." \
    --devicecc-options="..." \
    -device-libs="-L/opt/mellanox/doca/include -ldoca_dpa_dev -ldoca_dpa_dev_comm" \
    --app-name="dpa_example_app"
```

The user must use the following commands to set the **app** of a DPA context:

```
extern struct doca_dpa_app *dpa_example_app;
doca_dpa_create(&dpa);
doca_dpa_set_app(dpa, dpa_example_app);
doca_dpa_start(dpa);
```

5.4.3.6.5 Software Architecture

5.4.3.6.5.1 DOCA DPA Software Objects

Term	Definition
DPA context	Software construct for the host process that encapsulates the state associated with a DPA process (on a specific device). DPA context must be associated with a PF device.
Extended DPA context	A DPA context associated with a VF/SF device to be used for RDMA utilities. This to allow creation of DPA resources such as RDMA/DPA completion/DPA Async ops... contexts on the VF/SF device.
DPA Application	Interface with the DPACC compiler to produce a DPA program (app) which is obtained by the DPA context to begin working on DPA.
Kernel	User function (and its arguments) to be executed on DPA. A kernel may be executed by one or more DPA threads.
DPA EU Affinity	An object used to control which EU to use for DPA thread.
DPA EU Group Affinity	An object used to bind a DPA thread to an EU group. The thread may run on any EU within the group.
DPA Thread	DOCA DPA provides APIs to create/manage DPA thread which runs a given kernel.
DPA Completion Context	An object used to receive/handle a completion notification. The user can associate a DPA thread with a completion context. When the completion context receives a notification, DPA thread is scheduled.
DPA Thread Notification	A mechanism for one DPA thread to notify another DPA thread.
DPA Async Ops	An object used to allow a DPA thread to issue asynchronous operations, like memcpy or post_wait operations.
DPA RPC	A blocking one-time call from host application to execute a kernel on DPA. RPC is mainly used for control path. The RPC's return value is reported back to the host application.
DPA Memory	DOCA DPA provides an API to allocate/manage DPA memory, as well as handling host/Target BlueField memory that has been exported to DPA.
Sync Event	Data structure in either CPU, Target BlueField, GPU, or DPA-heap. An event contains a counter that can be updated and waited on.

Term	Definition
RDMA	Abstraction around a network transport object. Allows executing various RDMA operations.
DPA Hash Table	DOCA DPA provides an API to create a Hash Table on DPA. This data structure is managed on DPA using relevant device APIs.
DPA Logger/Tracer	DOCA DPA provides a set of debugging APIs to allow the user to diagnose and troubleshoot any issue on the device, as well as accessing real-time information from the running application.

5.4.3.6.5.2 Deployment View

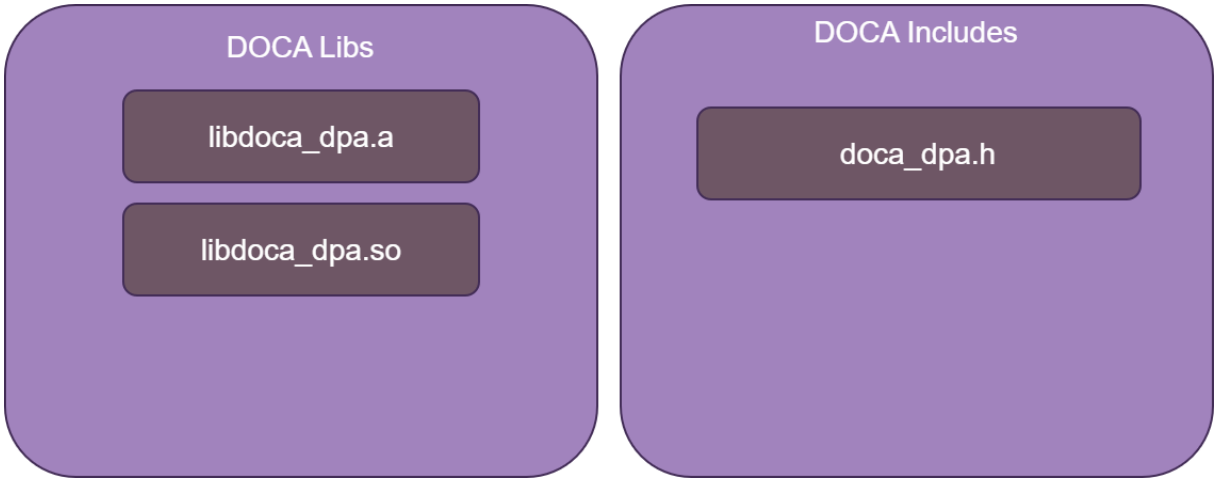
DOCA DPA comprises two main components which are part of the DOCA SDK installation package:

- DOCA DPA host library
- DOCA DPA device library

> ⚠ The DOCA DPA SDK does not use any means of multi-thread synchronization primitives. All DOCA DPA objects are non-thread-safe. Developers should make sure the user program and kernels are written to avoid race conditions.

DOCA DPA Host Library

Host library and header files:



The host library offers an interface for managing the following components:

Affinity

Users can control which EU a DPA thread runs on using DPA affinity objects. There are two affinity object types:

Type	Binding level	Description
DPA EU affinity	Fixed (single EU)	Thread runs only on the specified EU ID

Type	Binding level	Description
DPA EU group affinity	Group	Thread may run on any EU within the specified EU group

A DPA thread supports three affinity modes:

Mode	How to set	Behavior
Relaxed (default)	No affinity set	Thread may run on any available EU when rescheduled
Fixed	`doca_dpa_thread_set_affinity()`	Thread runs only on the EU specified by a DPA EU affinity object
Group	`doca_dpa_thread_set_group_affinity()`	Thread may run on any EU within the EU group specified by a DPA EU group affinity object

Setting one affinity type clears the other:

- Setting group affinity clears any previously set EU affinity.
- Setting EU affinity clears any previously set group affinity.

Affinity must be configured before the thread is started (`doca_dpa_thread_start()`).

DPA EU Affinity

A DPA EU affinity object targets a single EU ID at a time.

Use the following host-side APIs to manage it:

Create/destroy DPA EU affinity object:

```
doca_error_t doca_dpa_eu_affinity_create(struct doca_dpa *dpa, struct doca_dpa_eu_affinity **affinity)
doca_error_t doca_dpa_eu_affinity_destroy(struct doca_dpa_eu_affinity *affinity)
```

Set/clear EU ID in DPA EU affinity object:

```
doca_error_t doca_dpa_eu_affinity_set(struct doca_dpa_eu_affinity *affinity, unsigned int eu_id)
doca_error_t doca_dpa_eu_affinity_clear(struct doca_dpa_eu_affinity *affinity)
```

Get EU ID of a DPA EU affinity object:

```
doca_error_t doca_dpa_eu_affinity_get(struct doca_dpa_eu_affinity *affinity, unsigned int *eu_id)
```

DPA EU Group Affinity

A DPA EU group affinity object targets a single EU group ID at a time. A thread bound to a group affinity may run on any EU within that group.

Valid EU group IDs are discovered from the application's SPRD resources (see DPA Resources below) using `doca_dpa_resources_get_num_eu_groups()` / `doca_dpa_resources_get_eu_groups()` .

Use the following host-side APIs to manage it:

Create/destroy DPA EU group affinity object:

```
doca_error_t doca_dpa_eu_group_affinity_create(struct doca_dpa *dpa, struct doca_dpa_eu_group_affinity **affinity)
doca_error_t doca_dpa_eu_group_affinity_destroy(struct doca_dpa_eu_group_affinity *affinity)
```

Set/clear EU group ID in DPA EU group affinity object:

```
doca_error_t doca_dpa_eu_group_affinity_set(struct doca_dpa_eu_group_affinity *affinity, uint32_t eu_group_id)
doca_error_t doca_dpa_eu_group_affinity_clear(struct doca_dpa_eu_group_affinity *affinity)
```

Get EU group ID of a DPA EU group affinity object:

```
doca_error_t doca_dpa_eu_group_affinity_get(const struct doca_dpa_eu_group_affinity *affinity, uint32_t
*eu_group_id)
```

DPA Resources

The DPA Resources API provides a mechanism to parse and query pre-defined DPA resource configurations. This enables applications to discover and utilize the Execution Units (EUs) and EU groups allocated to them in the Single Point of Resource Distribution (SPRD) file.

The SPRD is a YAML-formatted configuration file that defines DPA application resources. It maps application names (keys) to specific resources, specifying:

- The number of allocated EUs per application.
- The specific EU IDs assigned to each application.
- The number of allocated EU groups per application.
- The specific EU group IDs assigned to each application.

Create/destroy DPA resources object:

```
doca_error_t doca_dpa_resources_create(const char *app_key, const void *sprd_buf, size_t sprd_buf_size, struct
doca_dpa_resources **resources)
doca_error_t doca_dpa_resources_destroy(struct doca_dpa_resources *resources)
```

Query allocated EUs:

```
doca_error_t doca_dpa_resources_get_num_eus(const struct doca_dpa_resources *resources, uint32_t *num_eus)
doca_error_t doca_dpa_resources_get_eus(const struct doca_dpa_resources *resources, uint32_t num_eus, uint32_t
*eus)
```

Query allocated EU groups:

```
doca_error_t doca_dpa_resources_get_num_eu_groups(const struct doca_dpa_resources *resources, uint32_t
*num_eu_groups)
doca_error_t doca_dpa_resources_get_eu_groups(const struct doca_dpa_resources *resources, uint32_t num_eu_groups,
uint32_t *eu_groups)
```

Validation resources

A DPA resources object can also be attached to a DPA context for runtime validation via **doca_dpa_set_validation_resources()**. When validation resources are set, the runtime verifies at thread start that:

- A thread with fixed affinity is bound to an EU ID listed in the resources object.
- A thread with group affinity is bound to an EU group ID listed in the resources object.

Threads with relaxed affinity (no explicit binding) are not validated against the resources object.

Validation resources must be set before **doca_dpa_start()**.

```
doca_error_t doca_dpa_set_validation_resources(struct doca_dpa *dpa, struct doca_dpa_resources *resources)
doca_error_t doca_dpa_get_validation_resources(const struct doca_dpa *dpa, struct doca_dpa_resources **resources)
```

For file syntax and examples, refer to Single Point of Resource Distribution documentation.

DPA Thread

DOCA DPA thread used to run a user function “DPA kernel” on DPA.

You can control where the DPA kernel runs by attaching either a DPA EU affinity object (fixed binding to a single EU) or a DPA EU group affinity object (binding to any EU within a group).

The thread can be triggered on DPA using two methods:

1. DPA thread notification – Notifying one DPA thread from another DPA thread.
2. DPA completion context – A completion is arrived at a DPA completion context which is attached to the thread.

DPA thread operations:

- To create/destroy DPA thread:

```
doca_error_t doca_dpa_thread_create(struct doca_dpa *dpa, struct doca_dpa_thread **dpa_thread)
doca_error_t doca_dpa_thread_destroy(struct doca_dpa_thread *dpa_thread)
```

- To set/get thread user function and its argument:

```
doca_error_t doca_dpa_thread_set_func_arg(struct doca_dpa_thread *thread, doca_dpa_func_t *func, uint64_t
arg)
doca_error_t doca_dpa_thread_get_func_arg(struct doca_dpa_thread *dpa_thread, doca_dpa_func_t **func,
uint64_t *arg)
```

- To set/get DPA EU Affinity (fixed binding):

```
doca_error_t doca_dpa_thread_set_affinity(struct doca_dpa_thread *thread, const struct doca_dpa_eu_affinity
*affinity)
doca_error_t doca_dpa_thread_get_affinity(struct doca_dpa_thread *dpa_thread, const struct
doca_dpa_eu_affinity **affinity)
```

- To set/get DPA EU group affinity (group binding):

```
doca_error_t doca_dpa_thread_set_group_affinity(struct doca_dpa_thread *dpa_thread, const struct
doca_dpa_eu_group_affinity *affinity)
doca_error_t doca_dpa_thread_get_group_affinity(const struct doca_dpa_thread *dpa_thread, const struct
doca_dpa_eu_group_affinity **affinity)
```

- Thread Local Storage (TLS)
  You can ask to store an opaque for a DPA thread in host side application using the following API:

```
doca_error_t doca_dpa_thread_set_local_storage(struct doca_dpa_thread *dpa_thread, doca_dpa_dev_uintptr_t
dev_ptr)
doca_error_t doca_dpa_thread_get_local_storage(struct doca_dpa_thread *dpa_thread, doca_dpa_dev_uintptr_t
*dev_ptr)
```

dev_ptr is a pre-allocated DPA memory.
In kernel, user can retrieve the stored opaque using the relevant device API (see below API). This opaque is stored/retrieved using the Thread Local Storage (TLS) mechanism.

- To start/stop DPA thread:

```
doca_error_t doca_dpa_thread_start(struct doca_dpa_thread *thread)
doca_error_t doca_dpa_thread_stop(struct doca_dpa_thread *dpa_thread)
```

- To run DPA thread:

```
doca_error_t doca_dpa_thread_run(struct doca_dpa_thread *dpa_thread)
```

This API sets the thread to run state. This function must be called after DPA thread is:

a. Created, set, and started.
b. If a DPA thread is attached to a DPA completion context, the completion context must be started first.

Example (host-side pseudo code):

```
extern doca_dpa_func_t hello_kernel;

// create DPA thread
doca_dpa_thread_create(&dpa_thread);

// set thread kernel
doca_dpa_thread_set_func_arg(dpa_thread, &hello_kernel, func_arg);

// set thread affinity
doca_dpa_eu_affinity_create(&eu_affinity);
doca_dpa_eu_affinity_set(eu_affinity, 10 /* EU ID */);
doca_dpa_thread_set_affinity(dpa_thread, eu_affinity);

// set thread local storage
doca_dpa_mem_alloc(&tls_dev_ptr);
doca_dpa_thread_set_local_storage(dpa_thread, tls_dev_ptr);

// start thread
doca_dpa_thread_start(dpa_thread);

// create and initialize DPA Completion Context
doca_dpa_completion_create(&dpa_comp);
doca_dpa_completion_set_thread(dpa_comp, dpa_thread);
doca_dpa_completion_start(dpa_comp);

// run thread only after both thread is started and the attached completion context is started
doca_dpa_thread_run(dpa_thread);
```

Completion Context

The DPA Completion Context integrates user applications with the DPA's native event-driven scheduling model. It functions as a generic mechanism shared across Message Queues, RDMA, Ethernet, and DPA Async Operations.

Operational modes:

- Event-driven: A DPA Thread is associated with a completion context. When the context receives a notification, the DPA Thread is automatically triggered.
- Polling: The application creates a completion context without an associated DPA Thread and polls for events manually.

Additionally, the system provides control over notification delivery, allowing the application to enable or suppress incoming notifications as needed

- To create/destroy DPA Completion Context:

```
doca_error_t doca_dpa_completion_create(struct doca_dpa *dpa, unsigned int queue_size, struct
doca_dpa_completion **dpa_comp)
```

```
doca_error_t doca_dpa_completion_destroy(struct doca_dpa_completion *dpa_comp)
```

- To get queue size:

```
doca_error_t doca_dpa_completion_get_queue_size(struct doca_dpa_completion *dpa_comp, unsigned int *size)
```

- To attach to a DPA Thread:

```
doca_error_t doca_dpa_completion_set_thread(struct doca_dpa_completion *dpa_comp, struct doca_dpa_thread
*thread)

doca_error_t doca_dpa_completion_get_thread(struct doca_dpa_completion *dpa_comp, struct doca_dpa_thread
**thread)
```

  Attaching to a thread is only required if the user wants triggering of the thread when a completion is arrived at the completion context.

- To start/stop DPA Completion Context:

```
doca_error_t doca_dpa_completion_start(struct doca_dpa_completion *dpa_comp)

doca_error_t doca_dpa_completion_stop(struct doca_dpa_completion *dpa_comp)
```

- To get DPA handle:

```
doca_error_t doca_dpa_completion_get_dpa_handle(struct doca_dpa_completion *dpa_comp,
doca_dpa_dev_completion_t *handle)
```

  Use output parameter **handle** for below device APIs which can be used in thread kernel.

Thread Notification

Thread Activation is a mechanism for one DPA thread to trigger another DPA Thread.

Thread activation is done without receiving a completion on the attached thread. Therefore it is expected that user of this method of thread activation passes the message in another fashion – such as shared memory.

Thread Activation can be achieved using DPA Notification Completion object.

- To create/destroy DPA Notification Completion:

```
doca_error_t doca_dpa_notification_completion_create(struct doca_dpa *dpa, struct doca_dpa_thread
*dpa_thread, struct doca_dpa_notification_completion **notify_comp)

doca_error_t doca_dpa_notification_completion_destroy(struct doca_dpa_notification_completion *notify_comp)
```

  Attaching DPA Notification Completion to a DPA Thread is done using the given parameter **dpa_thread**.

- To get attached DPA Thread:

```
doca_error_t doca_dpa_notification_completion_get_thread(struct doca_dpa_notification_completion
*notify_comp, struct doca_dpa_thread **dpa_thread)
```

- To start/stop DPA Notification Completion:

```
doca_error_t doca_dpa_notification_completion_start(struct doca_dpa_notification_completion *notify_comp)

doca_error_t doca_dpa_notification_completion_stop(struct doca_dpa_notification_completion *notify_comp)
```

- To get DPA handle:

```
doca_error_t doca_dpa_notify_completion_get_dpa_handle(struct doca_dpa_notification_completion
*notify_comp, doca_dpa_dev_notification_completion_t *comp_handle)
```

  Use output parameter **comp_handle** for below device API which can be used in thread kernel.

Example (host-side pseudo code):

```
extern doca_dpa_func_t hello_kernel;

// create DPA thread
doca_dpa_thread_create(&dpa_thread);

// set thread kernel
doca_dpa_thread_set_func_arg(dpa_thread, &hello_kernel, func_arg);

// start thread
doca_dpa_thread_start(dpa_thread);

// create and start DPA notification completion
doca_dpa_notification_completion_create(dpa, dpa_thread, &notify_comp);

doca_dpa_notification_completion_start(notify_comp);

// get its DPA handle
doca_dpa_notification_completion_get_dpa_handle(notify_comp, &notify_comp_handle);

// run thread only after both thread is started and attached notification completion is started
doca_dpa_thread_run(dpa_thread);
```

Asynchronous Ops

DPA Async Ops allows DPA Thread to issue asynchronous operations, like memcpy or post_wait.

This feature requires the user to create an “asynchronous ops” context and attach to a completion context.

User is expected to adhere to `queue_size` limit on the device when posting operations.

The completion context can raise activation if it is attached to a DPA Thread.

User can also choose to progress the completion context via polling it manually.

User can provide DPA Async Ops `user_data`, and retrieve this metadata in device using relevant device API.

- To create/destroy DPA Async Ops:

```
doca_error_t doca_dpa_async_ops_create(struct doca_dpa *dpa, unsigned int queue_size, uint64_t user_data,
struct doca_dpa_async_ops **async_ops)

doca_error_t doca_dpa_async_ops_destroy(struct doca_dpa_async_ops *async_ops)
```

  Please use the following define for valid user_data values:

```
#define DOCA_DPA_COMPLETION_LOG_MAX_USER_DATA   (24)
```

- To get queue size/user_data:

```
doca_error_t doca_dpa_async_ops_get_queue_size(struct doca_dpa_async_ops *async_ops, unsigned int
 *queue_size)

doca_error_t doca_dpa_async_ops_get_user_data(struct doca_dpa_async_ops *async_ops, uint64_t *user_data)
```

- To attach to a DPA Completion Context:

```
doca_error_t doca_dpa_async_ops_attach(struct doca_dpa_async_ops *async_ops, struct doca_dpa_completion
*dpa_comp)
```

- To start/stop DPA Async Ops:

```
doca_error_t doca_dpa_async_ops_start(struct doca_dpa_async_ops *async_ops)

doca_error_t doca_dpa_async_ops_stop(struct doca_dpa_async_ops *async_ops)
```

- To get DPA handle:

```
doca_error_t doca_dpa_async_ops_get_dpa_handle(struct doca_dpa_async_ops *async_ops,
doca_dpa_dev_async_ops_t *handle)
```

Use output parameter **handle** for below device API which can be used in thread kernel.

Example (host-side pseudo code):

```
doca_dpa_thread_create(&dpa_thread);
doca_dpa_thread_set_func_arg(dpa_thread);
doca_dpa_thread_start(dpa_thread);

doca_dpa_completion_create(&dpa_comp);
doca_dpa_completion_set_thread(dpa_comp, dpa_thread);
doca_dpa_completion_start(dpa_comp);

doca_dpa_thread_run(dpa_thread);

doca_dpa_async_ops_create(&async_ops);
doca_dpa_async_ops_attach(async_ops, dpa_comp);
doca_dpa_async_ops_start(async_ops);

doca_dpa_async_ops_get_dpa_handle(async_ops, &handle); // use this handle in relevant Async Ops device APIs
```

Thread Group

Thread group is used to aggregate individual DPA threads to a single group.

- To create/destroy DPA Thread Group:

```
doca_error_t doca_dpa_thread_group_create(struct doca_dpa *dpa, unsigned int num_threads, struct
doca_dpa_tg **tg)

doca_error_t doca_dpa_thread_group_destroy(struct doca_dpa_tg *tg)
```

- To get number of threads:

```
doca_error_t doca_dpa_thread_group_get_num_threads(struct doca_dpa_tg *tg, unsigned int *num_threads);
```

- To set DPA Thread at 'rank' in DPA Thread Group:

```
doca_error_t doca_dpa_thread_group_set_thread(struct doca_dpa_tg *tg, struct doca_dpa_thread *thread,
unsigned int rank)
```

Thread rank is an index of the thread (between 0 and (num_threads - 1)) within the group.

- To start/stop DPA Thread Group:

```
doca_error_t doca_dpa_thread_group_start(struct doca_dpa_tg *tg)
doca_error_t doca_dpa_thread_group_stop(struct doca_dpa_tg *tg)
```

Memory Subsystem

The user can allocate (from the host API) and access (from both the host and device API) several memory locations using the relevant DOCA DPA API.

DOCA DPA supports access from the host/Target BlueField to DPA heap memory and also enables device access to host memory (e.g., kernel writes to host memory).

The normal memory usage flow would be to:

1. Allocate memory (Host/Target BlueField/DPA).
2. Register the memory.
3. Get a DPA handle for the registered memory so it can be accessed by DPA kernels.
4. Access/use the memory from the kernel (see relevant device-side APIs).

Operations:

- To allocate DPA heap memory:

```
doca_dpa_mem_alloc(doca_dpa_t dpa, size_t size, doca_dpa_dev_uintptr_t *dev_ptr)
```

- To free previously allocated DPA memory:

```
doca_dpa_mem_free(doca_dpa_dev_uintptr_t dev_ptr)
```

- To copy previously allocated memory from a host pointer to a DPA heap device pointer:

```
doca_dpa_h2d_memcpy(doca_dpa_t dpa, doca_dpa_dev_uintptr_t src_ptr, void *dst_ptr, size_t size)
```

- To copy previously allocated memory from a DOCA Buffer to a DPA heap device pointer:

```
doca_error_t doca_dpa_h2d_buf_memcpy(struct doca_dpa *dpa, doca_dpa_dev_uintptr_t dst_ptr, struct doca_buf
*buf, size_t size)
```

- To copy previously allocated memory from a DPA heap device pointer to a host pointer:

```
doca_dpa_d2h_memcpy(doca_dpa_t dpa, void *dst_ptr, doca_dpa_dev_uintptr_t src_ptr, size_t size)
```

- To copy previously allocated memory from a DPA heap device pointer to a DOCA Buffer:

```
doca_error_t doca_dpa_d2h_buf_memcpy(struct doca_dpa *dpa, struct doca_buf *buf, doca_dpa_dev_uintptr_t
src_ptr, size_t size)
```

- To set memory:

```
doca_dpa_memset(doca_dpa_t dpa, doca_dpa_dev_uintptr_t dev_ptr, int value, size_t size)
```

- To get a DPA handle to use in kernels, the user must use a DOCA Core Memory Inventory Object in the following manner (refer to "DOCA Memory Subsystem"):
  - When the user wants to use device APIs with DOCA Buffer, use the following pseudo code:

    ```
    doca_buf_arr_create(&buf_arr);
    doca_buf_arr_set_target_dpa(buf_arr, doca_dpa);
    doca_buf_arr_start(buf_arr);
    doca_buf_arr_get_dpa_handle(buf_arr, &handle);
    ```

    Use output parameter **handle** in relevant device APIs in thread kernel.
  - When the user wants to use device APIs with DOCA Mmap, use the following pseudo code:

    ```
    doca_mmap_create(&mmap);
    ```

```
doca_mmap_set_dpa_memrange(mmap, doca_dpa, dev_ptr, dev_ptr_len); // dev-ptr is a pre-allocated DPA
memory
doca_mmap_start(mmap);
doca_mmap_dev_get_dpa_handle(mmap, doca_dev, &handle);
```

Use output parameter **handle** in relevant device APIs in thread kernel.

Extended DOCA DPA Context

A base DOCA DPA context is a context created on a PF DOCA device using:

```
doca_dpa_create(pf_doca_dev, &base_dpa_ctx);
```

To enable creating DPA resources (e.g., RDMA/DPA completion/DPA Async ops contexts) on VF/SF DOCA device, an extended DOCA DPA context is required.

DOCA DPA provides the following host API to extend a base DOCA DPA context (created on a PF DOCA device) to an SF/VF DOCA device:

```
doca_error_t doca_dpa_device_extend(struct doca_dpa *dpa, struct doca_dev *other_dev, struct doca_dpa
**extended_dpa)
```

The extended DPA context can be used later on for creation of DPA resources (e.g., as RDMA/DPA completion/DPA Async ops contexts) on the other DOCA device (SF/VF).

Note that:

- The extended DPA context is already started.
- The extended DPA context can be used later for all DOCA DPA APIs (e.g., creating DPA memory, DPA completion context) or within kernel launch flow.

> ⚠ When running from the DPU, DOCA RDMA context (on DPA datapath) must be created on SF DOCA device. Therefore, it must be created using an extended DOCA DPA context (created on the same SF DOCA device).

To obtain a DPA handle for a DOCA DPA context (both base or extended):

```
doca_error_t doca_dpa_get_dpa_handle(struct doca_dpa *dpa, doca_dpa_dev_t *handle)
```

Use output parameter **handle** in the relevant device's APIs in the thread kernel.

> ⚠ When creating a DOCA RDMA context with an extended DOCA DPA context and that RDMA context is attached to a DPA completion context and a DPA thread, all DOCA RDMA, DPA completion context, and the DPA thread must be created on the same extended DOCA DPA context.

Example (host-side pseudo code):

```
// create a base DPA context
doca_dpa_create(pf_doca_dev, &base_dpa_ctx);
// call base_dpa_ctx setters
doca_dpa_start(base_dpa_ctx);

// create an extended DPA context
doca_dpa_device_extend(base_dpa_ctx, sf_doca_dev, &extended_dpa_ctx);
doca_dpa_get_dpa_handle(extended_dpa_ctx, &extended_dpa_ctx_handle);

// create DPA thread on extended DPA context
```

```
doca_dpa_thread_create(extended_dpa_ctx, &dpa_thread);
// call dpa_thread setters
doca_dpa_thread_start(dpa_thread);

// create DPA completion context on extended DPA context
doca_dpa_completion_create(extended_dpa_ctx, &dpa_completion);
doca_dpa_completion_set_thread(dpa_completion, dpa_thread);
doca_dpa_completion_start(dpa_completion);

// create DOCA RDMA context on SF DOCA device and extended DPA context
doca_rdma_create(sf_doca_dev, &rdma);
doca_ctx_set_datapath_on_dpa(rdma_as_ctx, extended_dpa_ctx);
doca_rdma_dpa_completion_attach(rdma, dpa_completion);
// other rdma setters
doca_ctx_start(rdma_as_ctx);
doca_rdma_get_dpa_handle(rdma, rdma_dpa_handle);
```

Data Structures

Hash Table

DOCA DPA provides an API to create a hash table on DPA. This data structure is managed on DPA using relevant device APIs.

- To create a hash table on DPA:

```
doca_error_t doca_dpa_hash_table_create(struct doca_dpa *dpa, unsigned int num_entries, struct
doca_dpa_hash_table **ht)
```

- To destroy a hash table:

```
doca_error_t doca_dpa_hash_table_destroy(struct doca_dpa_hash_table *ht)
```

- To obtain a DPA handle:

```
doca_error_t doca_dpa_hash_table_get_dpa_handle(struct doca_dpa_hash_table *ht, doca_dpa_dev_hash_table_t
*handle)
```

  Use output parameter **handle** in relevant device APIs in the thread kernel.

RPC

A blocking one-time call from the host application to execute a kernel on DPA.

> RPC is mainly used for control path.

The RPC's return value is reported back to the host application.

```
doca_error_t doca_dpa_rpc(struct doca_dpa *dpa, doca_dpa_func_t *func, uint64_t *retval, … /* func arguments */)
```

Example:

- Device-side – DPA device **func** must be annotated with **__dpa_rpc__** annotation, such as:

```
__dpa_rpc__ uint64_t hello_rpc(int arg)
{
    ...
}
```

- Host-side:

```
extern doca_dpa_func_t hello_rpc;

uint64_t retval;
```

```
doca_dpa_rpc(dpa, &hello_rpc, &retval, 10);
```

Kernel Launch

DOCA DPA provides an API which enables full control for launching and monitoring kernels.

Since DOCA DPA libraries are not thread-safe, it is up to the programmer to make sure the kernel is written to allow it to run in a multi-threaded environment. For example, to program a kernel that uses RDMAs with 16 concurrent threads, the user should pass an array of 16 RDMAs to the kernel so that each thread can access its RDMA using its rank (**doca_dpa_dev_thread_rank()**) as an index to the array.

```
doca_dpa_kernel_launch_update_<add|set>(struct doca_dpa *dpa, struct doca_sync_event *wait_event, uint64_t
wait_threshold, struct doca_sync_event *comp_event, uint64_t comp_count, unsigned int num_threads, doca_dpa_func_t
*func, ... /* args */)
```

- This function asks DOCA DPA to run **func** in DPA by **num_threads** and give it the supplied list of arguments (variadic list of arguments).
- This function is asynchronous so when it returns, it does not mean that **func** started/ended its execution.
- To add control or flow/ordering to these asynchronous kernels, two optional parameters for launching kernels are available:
  - **wait_event** – the kernel does not start its execution until the event is signaled (if NULL, the kernel starts once DOCA DPA has an available EU to run on it) which means that DOCA DPA would not run the kernel until the event's counter is bigger than **wait_threshold** .

    > ⚠ Please note that the valid values for **wait_threshold** and **wait_event** counter and are [0-254]. Values out of this range might cause anomalous behavior.

  - **comp_event** – once the last thread running the kernel is done, DOCA DPA updates this event (either sets or adds to its current counter value with **comp_count**).
- DOCA DPA takes care of packing (on host/Target BlueField) and unpacking (in DPA) the kernel parameters.
- **func** must be prefixed with the **__dpa_global__** macro for DPACC to compile it as a kernel (and add it to DPA executable binary) and not as part of host application binary.
- The programmer must declare **func** in their application also by adding the line **extern doca_dpa_func_t func** .

> ⚠ The following APIs are only relevant for a kernel used in **kernel_launch** APIs. These APIs are not relevant in **doca_dpa_thread** kernel.

- To retrieve the running thread's rank for a given kernel on the DPA. If, for example, a kernel is launched to run with 16 threads, each thread running this kernel is assigned a rank ranging from 0 to 15 within this kernel. This is helpful for making sure each thread in the kernel only accesses data relevant for its execution to avoid data-races:

```
unsigned int doca_dpa_dev_thread_rank()
```

- To return the number of threads running current kernel:

```
unsigned int doca_dpa_dev_num_threads()
```

- To yield the thread which runs the kernel:

```
void doca_dpa_dev_yield(void)
```

Examples

Linear Execution Example

Device-side pseudo code:

```
#include "doca_dpa_dev.h"
#include "doca_dpa_dev_sync_event.h"

__dpa_global__ void
linear_kernel(doca_dpa_dev_sync_event_t wait_ev, doca_dpa_dev_sync_event_t comp_ev)
{
    if (wait_ev)
        doca_dpa_dev_sync_event_wait_gt(wait_ev, wait_th = 0);

    doca_dpa_dev_sync_event_update_add(comp_ev, comp_count = 1);
}
```

Host-side pseudo code:

```
#include <doca_dev.h>
#include <doca_error.h>
#include <doca_sync_event.h>
#include <doca_dpa.h>

int main(int argc, char **argv)
{

    /*
        A
        |
        B
        |
        C
    */

    /* Open DOCA device */
    open_doca_dev(&doca_dev);
    /* Create doca dpa conext */
    doca_dpa_create(doca_dev, dpa_linear_app, &dpa_ctx, 0);

    /* Create event A - subscriber is DPA and publisher is CPU */
    doca_sync_event_create(&ev_a);
    doca_sync_event_add_publisher_location_cpu(ev_a, doca_dev);
    doca_sync_event_add_subscriber_location_dpa(ev_a, dpa_ctx);
    doca_sync_event_start(ev_a);

    /* Create event B - subscriber and publisher are DPA */
    doca_sync_event_create(&ev_b);
    doca_sync_event_add_publisher_location_dpa(ev_b, dpa_ctx);
    doca_sync_event_add_subscriber_location_dpa(ev_b, dpa_ctx);
    doca_sync_event_start(ev_b);

    /* Create event C - subscriber and publisher are DPA */
    doca_sync_event_create(&ev_c);
    doca_sync_event_add_publisher_location_dpa(ev_c, dpa_ctx);
    doca_sync_event_add_subscriber_location_dpa(ev_c, dpa_ctx);
    doca_sync_event_start(ev_c);

    /* Create completion event for last kernel - subscriber is CPU and publisher is DPA */
    doca_sync_event_create(&comp_ev);
    doca_sync_event_add_publisher_location_dpa(comp_ev, dpa_ctx);
    doca_sync_event_add_subscriber_location_cpu(comp_ev, doca_dev);
    doca_sync_event_start(comp_ev);

    /* Export kernel events and acquire their handles */
    doca_sync_event_get_dpa_handle(ev_b, dpa_ctx, &ev_b_handle);
    doca_sync_event_get_dpa_handle(ev_c, dpa_ctx, &ev_c_handle);
    doca_sync_event_get_dpa_handle(comp_ev, dpa_ctx, &comp_ev_handle);

    /* Launch kernels */
    doca_dpa_kernel_launch_update_add(wait_ev = ev_a, wait_threshold = 1, num_threads = 1, &linear_kernel,
kernel_args: NULL, ev_b_handle);
```

```
    doca_dpa_kernel_launch_update_add(wait_ev = NULL,  num_threads = 1, &linear_kernel, kernel_args: ev_b_handle,
ev_c_handle);
    doca_dpa_kernel_launch_update_add(wait_ev = NULL, &linear_kernel, num_threads = 1, kernel_args: ev_c_handle,
comp_ev_handle);

    /* Update host event to trigger kernels to start executing in a linear manner */
    doca_sync_event_update_set(ev_a, 1)

    /* Wait for completion of last kernel */
    doca_sync_event_wait_gt(comp_ev, 0);

    /* Tear Down... */
    teardown_resources();
}
```

Diamond Execution Example

Device-side pseudo code:

```
#include "doca_dpa_dev.h"
#include "doca_dpa_dev_sync_event.h"

__dpa_global__ void
diamond_kernel(doca_dpa_dev_sync_event_t wait_ev, uint64_t wait_th, doca_dpa_dev_sync_event_t comp_ev1,
doca_dpa_dev_sync_event_t comp_ev2)
{
    if (wait_ev)
        doca_dpa_dev_sync_event_wait_gt(wait_ev, wait_th);

    doca_dpa_dev_sync_event_update_add(comp_ev1, comp_count = 1);
    if (comp_ev2)    // can be 0 (NULL)
        doca_dpa_dev_sync_event_update_add(comp_ev2, comp_count = 1);
}
```

Host-side pseudo code:

```
#include <doca_dev.h>
#include <doca_error.h>
#include <doca_sync_event.h>
#include <doca_dpa.h>

int main(int argc, char **argv)
{
    /*
         A
        / \
       C   B
      /   /
     D   /
      \ /
       E
    */

    /* Open DOCA device */
    open_doca_dev(&doca_dev);
    /* Create doca dpa conext */
    doca_dpa_create(doca_dev, dpa_diamond_app, &dpa_ctx, 0);

    /* Create root event A that will signal from the host the rest to start */
    doca_sync_event_create(&ev_a);
    // set publisher to CPU, subscriber to DPA and start event

    /* Create events B,C,D,E */
    doca_sync_event_create(&ev_b);
    doca_sync_event_create(&ev_c);
    doca_sync_event_create(&ev_d);
    doca_sync_event_create(&ev_e);
    // for events B,C,D,E, set publisher & subscriber to DPA and start event

    /* Create completion event for last kernel */
    doca_sync_event_create(&comp_ev);
    // set publisher to DPA, subscriber to CPU and start event

    /* Export kernel events and acquire their handles */
    doca_sync_event_get_dpa_handle(&ev_b_handle, &ev_c_handle, &ev_d_handle, &ev_e_handle, &comp_ev_handle);

    /* wait threshold for each kernel is the number of parent nodes */
    constexpr uint64_t wait_threshold_one_parent {1};
    constexpr uint64_t wait_threshold_two_parent {2};

    /* launch diamond kernels */
    doca_dpa_kernel_launch_update_set(wait_ev = ev_a, wait_threshold = 1, num_threads = 1, &diamond_kernel,
kernel_args: NULL, 0, ev_b_handle, ev_c_handle);
    doca_dpa_kernel_launch_update_set(wait_ev = NULL, num_threads = 1, &diamond_kernel, kernel_args: ev_b_handle,
wait_threshold_one_parent, ev_e_handle, NULL);
    doca_dpa_kernel_launch_update_set(wait_ev = NULL, num_threads = 1, &diamond_kernel, kernel_args: ev_c_handle,
wait_threshold_one_parent, ev_d_handle, NULL);
    doca_dpa_kernel_launch_update_set(wait_ev = NULL, num_threads = 1, &diamond_kernel, kernel_args: ev_d_handle,
wait_threshold_one_parent, ev_e_handle, NULL);
    doca_dpa_kernel_launch_update_set(wait_ev = NULL, num_threads = 1, &diamond_kernel, kernel_args: ev_e_handle,
wait_threshold_two_parent, comp_ev_handle, NULL);

    /* Update host event to trigger kernels to start executing in a diamond manner */
    doca_sync_event_update_set(ev_a, 1);
    /* Wait for completion of last kernel */
```

```
    doca_sync_event_wait_gt(comp_ev, 0);

    /* Tear Down... */
    teardown_resources();
}
```

Performance Optimizations

- The time interval between a kernel launch call from the host and the start of its execution on the DPA is significantly optimized when the host application calls **doca_dpa_kernel_launch_update_<add|set>()** repeatedly to execute with the same number of DPA threads. So, if the application calls **doca_dpa_kernel_launch_update_<add|set>(..., num_threads = x)**, the next call with **num_threads = x** would have a shorter latency (as low as ~5-7 microseconds) for the start of the kernel's execution.
- Applications calling for kernel launch with a wait event (i.e., the completion event of a previous kernel) also have significantly lower latency in the time between the host launching the kernel and the start of the execution of the kernel on the DPA. So, if the application calls **doca_dpa_kernel_launch_update_<add|set>(..., completion event = m_ev, ...)** and then **doca_dpa_kernel_launch_update_<add|set>(wait event = m_ev, ...)**, the latter kernel launch call would have shorter latency (as low as ~3 microseconds) for the start of the kernel's execution.

Limitations

- The order in which kernels are launched is important. If an application launches K1 and then K2, K1 must not depend on K2's completion (e.g., wait on its wait event that K2 should update).
  Not following this guideline leads to unpredictable results (at runtime) for the application and might require restarting the DOCA DPA context (i.e., destroying, reinitializing, and rerunning the workload).
- DPA threads are an actual hardware resource and are, therefore, limited in number to 256 (including internal allocations and allocations explicitly requested by the user as part of the kernel launch API)
  - DOCA DPA does not check these limits. It is up to the application to adhere to this number and track thread allocation across different DPA contexts.
  - Each **doca_dpa_dev_rdma_t** consumes one thread.
- The DPA has an internal watchdog timer to make sure threads do not block indefinitely. Kernel execution time must be finite and not exceed the time returned.
  by **doca_dpa_get_kernel_max_run_time**.
- The **num_threads** parameter in the **doca_dpa_kernel_launch** call cannot exceed the maximum allowed number of threads to run a kernel returned.
  by **doca_dpa_get_max_threads_per_kernel**.

Logging and Tracing

DOCA DPA provides a set of debugging APIs to allow diagnosing and troubleshooting any issues on the device, as well as accessing real-time information from the running application.

Logging in the data path has significant impact on an application's performance. While the tracer provided by the library is of high-frequency and is designed to prevent significant impact on the application's performance.

Therefore, it is recommended to use:

- Logging in the control path
- Tracing in the data path

You can control the log/trace file path and device log verbosity.

- To set/get the trace file path:

```
doca_error_t doca_dpa_trace_file_set_path(struct doca_dpa *dpa, const char *file_path)
doca_error_t doca_dpa_trace_file_get_path(struct doca_dpa *dpa, char *file_path, uint32_t *file_path_len);
```

- To set/get the log file path:

```
doca_error_t doca_dpa_log_file_set_path(struct doca_dpa *dpa, const char *file_path)
doca_error_t doca_dpa_log_file_get_path(struct doca_dpa *dpa, char *file_path, uint32_t *file_path_len)
```

- To set/get device log verbosity:

```
doca_error_t doca_dpa_set_log_level(struct doca_dpa *dpa, doca_dpa_dev_log_level_t log_level)
doca_error_t doca_dpa_get_log_level(struct doca_dpa *dpa, doca_dpa_dev_log_level_t *log_level)
```

Error Handling

DPA context can enter an error state caused by the device flow. The application can check this error state by calling the following host API:

```
doca_error_t doca_dpa_peek_at_last_error(const struct doca_dpa *dpa)
```

If a fatal error core dump and crash occur, data is written to the file path **/tmp/doca_dpa_fatal** or to the file path set by the API **doca_dpa_log_file_set_path()**, with the suffixes **.PID.core** and **.PID.crash** respectively, where PID is the process ID. The data written to the file would include a memory snapshot at the time of the crash, which would contain instrumental information in pinpointing the cause of a crash (e.g., the program's state, variable values, and the call stack).

Creating core dump files can be done after the DPA application has crashed.

This call does not reset the error state.

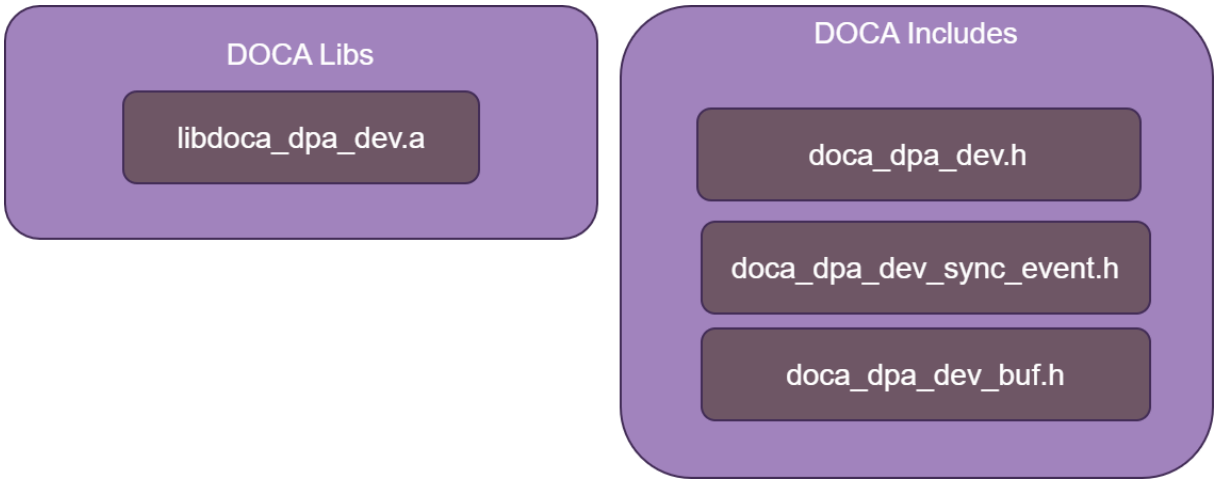
If an error occurred, DPA context enters a fatal state and must be destroyed by the user.

DOCA DPA Device Library

DOCA DPA device library offers an interface for common utilities such as:

- Managing DPA thread
- Managing DPA completion context
- Managing DPA hash table
- Log and trace
- Managing DPA sync event
- Managing DPA DOCA buf and mmap

DPA device library and header files:



DPA Thread

- Thread restart APIs
  DPA thread can ends its run using one of the following device APIs:
  - Reschedule API:

    ```
    void doca_dpa_dev_thread_reschedule(void)
    ```

    - DPA thread is still active
    - DPA thread resources are back to RTOS
    - DPA thread can be triggered again
  - Finish API:

    ```
    void doca_dpa_dev_thread_finish(void)
    ```

    - DPA thread is marked as finished
    - DPA thread resources are back to RTOS
    - DPA thread cannot be triggered again
- To get TLS:

  ```
  doca_dpa_dev_uintptr_t doca_dpa_dev_thread_get_local_storage(void)
  ```

  This function returns DPA thread local storage previously set using the host API **`doca_dpa_thread_set_local_storage()`**.

⚠ DPA thread device APIs cannot be used in the user's written kernel which is used in the DOCA DPA kernel launch API.

Completion Context

Kernels get **`doca_dpa_dev_completion_t`** handle and invoke the following API:

- To get a completion element:

```
int doca_dpa_dev_get_completion(doca_dpa_dev_completion_t dpa_comp_handle,
doca_dpa_dev_completion_element_t *comp_element)
```

Use the returned **comp_element** to retrieve completion info using the APIs below.

- To get completion element type:

```
typedef enum {
    DOCA_DPA_DEV_COMP_SEND = 0x0,                /**< Send completion */
    DOCA_DPA_DEV_COMP_RECV_RDMA_WRITE_IMM = 0x1, /**< Receive RDMA Write with Immediate completion */
    DOCA_DPA_DEV_COMP_RECV_SEND = 0x2,           /**< Receive Send completion */
    DOCA_DPA_DEV_COMP_RECV_SEND_IMM = 0x3,       /**< Receive Send with Immediate completion */
    DOCA_DPA_DEV_COMP_SEND_ERR = 0xD,            /**< Send Error completion */
    DOCA_DPA_DEV_COMP_RECV_ERR = 0xE             /**< Receive Error completion */
} doca_dpa_dev_completion_type_t;

doca_dpa_dev_completion_type_t doca_dpa_dev_get_completion_type(doca_dpa_dev_completion_element_t
comp_element)
```

- To get completion element user data:

```
uint32_t doca_dpa_dev_get_completion_user_data(doca_dpa_dev_completion_element_t comp_element)
```

This API returns user data which:

- Was set previously in host API **doca_dpa_async_ops_create(..., user_data, ...)**
  When DPA completion context is attached to the DPA async ops.
- Equivalent to **connection_id** in **doca_error_t doca_rdma_connection_get_id(const struct doca_rdma_connection *rdma_connection, uint32_t *connection_id)**
  When DPA completion context is attached to DOCA RDMA context.

- To get completion element immediate data:

```
uint32_t doca_dpa_dev_get_completion_immediate(doca_dpa_dev_completion_element_t comp_element)
```

This API returns immediate data for a completion element of type:

- **DOCA_DPA_DEV_COMP_RECV_RDMA_WRITE_IMM**
- **DOCA_DPA_DEV_COMP_RECV_SEND_IMM**

- To acknowledge that the completions have been read on the DPA completion context:

```
void doca_dpa_dev_completion_ack(doca_dpa_dev_completion_t dpa_comp_handle, uint64_t num_comp)
```

This API releases resources of the acked completion elements in the completion context. This acknowledgment enables receiving new **num_comp** completions.

- To request notification on the DPA completion context:

```
void doca_dpa_dev_completion_request_notification(doca_dpa_dev_completion_t dpa_comp_handle)
```

This API enables requesting new notifications on the DPA completion context. If this function is not called, the DPA completion context is not notified on newly arrived completion elements. Therefore, new completions are not populated in DPA completion context.

Example (device-side pseudo code):

```
__dpa_global__ void hello_kernel(uint64_t arg)
{
    // User is expected to pass in some way the attached completion context handle "dpa_comp_handle" to kernel such
as func_arg or a shared memory.
    DOCA_DPA_DEV_LOG_INFO("Hello from kernel\n");

    doca_dpa_dev_completion_element_t comp_element;
    found = doca_dpa_dev_get_completion(dpa_comp_handle, &comp_element);
```

```
    if (found) {
        comp_type = doca_dpa_dev_get_completion_type(comp_element);
        // process the completion according to completion type...

        // ack on 1 completion
        doca_dpa_dev_completion_ack(dpa_comp_handle, 1);

        // enable getting more completions and triggering the thread
        doca_dpa_dev_completion_request_notification(dpa_comp_handle);
  }

    // reschedule thread
    doca_dpa_dev_thread_reschedule();
}
```

Thread Notification

Kernels get **doca_dpa_dev_notification_completion_t** handle and invoke the following API:

```
void doca_dpa_dev_thread_notify(doca_dpa_dev_notification_completion_t comp_handle)
```

Calling this API triggers the attached DPA Thread (the one that is specified in **dpa_thread** parameter in host-side API **doca_dpa_notification_completion_create()**).

Asynchronous Ops

Kernels get the **doca_dpa_dev_async_ops_t** handle and invoke the APIs listed below.

Users may control work request submit configuration using the following enum:

```
/**
 * @brief DPA submit flag type
 */
__dpa_global__ enum doca_dpa_dev_submit_flag {
    DOCA_DPA_DEV_SUBMIT_FLAG_NONE = 0U,
    DOCA_DPA_DEV_SUBMIT_FLAG_FLUSH = (1U << 0),           /**
                                * Use flag to inform related DPA context
                                * (such as RDMA or DPA Async ops) to flush related
                                * operation and previous operations to HW,
                                * otherwise the context may aggregate the operation
                                * and not flush it immediately
                                */
    DOCA_DPA_DEV_SUBMIT_FLAG_OPTIMIZE_REPORTS = (1U << 1), /**
                                * Use flag to inform related DPA context that it may
                                * defer completion of the operation to a later time. If
                                * flag is not provided then a completion will be raised
                                * as soon as the operation is finished, and any
                                * preceding completions that were deferred will also be
                                * raised. Use this flag to optimize the amount of
                                * completion notifications it receives from HW when
                                * submitting a batch of operations, by receiving a
                                * single completion notification on the entire batch.
                                */
};
```

This enum can be used in the **flags** parameter in the following device APIs:

- To post a memcpy operation using **doca_buf**:

  ```
  void doca_dpa_dev_post_buf_memcpy(doca_dpa_dev_async_ops_t async_ops_handle, doca_dpa_dev_buf_t
  dst_buf_handle, doca_dpa_dev_buf_t src_buf_handle, uint32_t flags)
  ```

  This API copies data between two DOCA buffers. The destination buffer, specified by **dst_buf_handle**, will contain the copied data after memory copy is complete. This is a non-blocking routine.
- To post a memcpy operation using **doca_mmap** and an explicit address:

  ```
  void doca_dpa_dev_post_memcpy(doca_dpa_dev_async_ops_t async_ops_handle,
                                doca_dpa_dev_mmap_t dst_mmap_handle,
                                uint64_t dst_addr,
                                doca_dpa_dev_mmap_t src_mmap_handle,
                                uint64_t src_addr,
                                size_t length,
  ```

```
                                uint32_t flags)
```

This API copies data between two DOCA Mmaps. The destination DOCA Mmap (**dst_addr**), specified by **dst_mmap_handle** , will contain the copied data in source DOCA Mmap specified by **src_mmap_handle** , **src_addr** , and **length** after memory copy is complete. This is a non-blocking routine.

> Use this API for memcpy instead of using the **doca_buf memcpy** API to gain better performance.

- To post a wait greater operation on a DOCA Sync Event:

```
void doca_dpa_dev_sync_event_post_wait_gt(doca_dpa_dev_async_ops_t async_ops_handle,
doca_dpa_dev_sync_event_t wait_se_handle, uint64_t value)
```

This function posts a wait operation on the DOCA Sync Event using DPA async ops to obtain DPA thread activation. The attached thread is activated when the value of a DOCA Sync Event is greater than a given value. This is a non-blocking routine.

> Valid values must be in the range [0, 254] and can be called for an event with a value in the range [0, 254]. Invalid values leads to anomalous behavior.

- To post a wait not equal operation on a DOCA Sync Event:

```
void doca_dpa_dev_sync_event_post_wait_ne(doca_dpa_dev_async_ops_t async_ops_handle,
doca_dpa_dev_sync_event_t wait_se_handle, uint64_t value)
```

This function posts a wait operation on the DOCA Sync Event using the DPA async ops to obtain DPA thread activation. The attached thread is activated when the value of DOCA Sync Event is not equal to a given value. This is a non-blocking routine.

Memory Subsystem

Memory APIs supplied by the DOCA DPA SDK are all asynchronous (i.e., non-blocking).

The user can acquire either:

1. Pre-configured DOCA Buffers (previously configured with **doca_buf_arr_set_params**).
2. Non-configured DOCA Buffers and use below device setters to configure them.

Device-side API operations:

- To obtain a single buffer handle from the buf array handle:

```
doca_dpa_dev_buf_t doca_dpa_dev_buf_array_get_buf(doca_dpa_dev_buf_arr_t buf_arr, const uint64_t buf_idx)
```

- To set/get the address pointed to by the buffer handle:

```
void doca_dpa_dev_buf_set_addr(doca_dpa_dev_buf_t buf, uintptr_t addr)
uintptr_t doca_dpa_dev_buf_get_addr(doca_dpa_dev_buf_t buf)
```

- To set/get the length of the buffer:

```
void doca_dpa_dev_buf_set_len(doca_dpa_dev_buf_t buf, size_t len)
```

```
uint64_t doca_dpa_dev_buf_get_len(doca_dpa_dev_buf_t buf)
```

- To set/get the DOCA Mmap associated with the buffer:

```
void doca_dpa_dev_buf_set_mmap(doca_dpa_dev_buf_t buf, doca_dpa_dev_mmap_t mmap)
doca_dpa_dev_mmap_t doca_dpa_dev_buf_get_mmap(doca_dpa_dev_buf_t buf)
```

- To get a pointer to external memory registered on the host using DOCA Buffer:

```
doca_dpa_dev_uintptr_t doca_dpa_dev_buf_get_external_ptr(doca_dpa_dev_buf_t buf)
```

> After calling this API, users can read/write the memory indicated by the returned pointer from the DPA kernel.

- To get a pointer to external memory registered on the host using an explicit address and DOCA Mmap:

```
doca_dpa_dev_uintptr_t doca_dpa_dev_mmap_get_external_ptr(doca_dpa_dev_mmap_t mmap_handle, uint64_t addr)
```

> After calling this API, users can read/write the memory indicated by the returned pointer from the DPA kernel.

Sync Events

Sync events fulfill the following roles:

- DOCA DPA execution model is asynchronous and sync events are used to control various threads running in the system (allowing order and dependency)
- DOCA DPA supports remote sync events, so the programmer is capable of invoking remote nodes by means of DOCA sync events

> For host-side APIs, refer to "DOCA Sync Event".

- To get the current event value:

```
doca_dpa_dev_sync_event_get(doca_dpa_dev_sync_event_t event, uint64_t *value)
```

- To add/set to the current event value:

```
doca_dpa_dev_sync_event_update_<add|set>(doca_dpa_dev_sync_event_t event, uint64_t value)
```

- To wait until event is greater than threshold:

```
doca_dpa_dev_sync_event_wait_gt(doca_dpa_dev_sync_event_t event, uint64_t value, uint64_t mask)
```

Use mask to apply bitwise AND on the DOCA Sync Event value for comparison with the wait threshold.

Extended DOCA DPA Context

To work with a DPA resource created on an extended DOCA DPA context, DOCA DPA offers the following device API:

```
void doca_dpa_dev_device_set(doca_dpa_dev_t dpa_handle)
```

This function must be called before calling any related device API of a DPA resource (e.g., DPA RDMA) created on an extended DOCA DPA context.

When creating only a DPA base context without an extended DPA context in this application, there is no need to call this API.

When creating DPA resources on both base and extended DPA contexts, the proper DPA device (and DPA context) must be set before calling the relevant DPA resource API.

Example (device-side pseudo code):

```
__dpa_global__ void kernel(uint64_t thread_arg)
{
    // parse completion...

    // set extended device before executing RDMA operation
    doca_dpa_dev_device_set(thread_arg->extended_dpa_ctx_handle);
    doca_dpa_dev_rdma_post_send(rdma_dpa_handle);
    ...
}
```

Data Structures

Hash Table

This data structure is managed on DPA using the following device APIs:

- To add a new entry to the hash table:

  ```
  void doca_dpa_dev_hash_table_add(doca_dpa_dev_hash_table_t ht_handle, uint32_t key, uint64_t value)
  ```

  Adding a new key when the hash table is full causes anomalous behavior.

- To remove an entry from the hash table:

  ```
  void doca_dpa_dev_hash_table_remove(doca_dpa_dev_hash_table_t ht_handle, uint32_t key)
  ```

- To return the value to which the specified key is mapped in the hash table:

  ```
  int doca_dpa_dev_hash_table_find(doca_dpa_dev_hash_table_t ht_handle, uint32_t key, uint64_t *value)
  ```

Logging and Tracing

- Log to host:

  ```
  typedef enum doca_dpa_dev_log_level {
      DOCA_DPA_DEV_LOG_LEVEL_DISABLE = 10, /**< Disable log messages */
      DOCA_DPA_DEV_LOG_LEVEL_CRIT = 20,    /**< Critical log level */
      DOCA_DPA_DEV_LOG_LEVEL_ERROR = 30,   /**< Error log level */
      DOCA_DPA_DEV_LOG_LEVEL_WARNING = 40, /**< Warning log level */
      DOCA_DPA_DEV_LOG_LEVEL_INFO = 50,    /**< Info log level */
  ```

```
    DOCA_DPA_DEV_LOG_LEVEL_DEBUG = 60,    /**< Debug log level */
} doca_dpa_dev_log_level_t;

void doca_dpa_dev_log(doca_dpa_dev_log_level_t log_level, const char *format, ...)
```

- Log macros:

```
DOCA_DPA_DEV_LOG_CRIT(...)
DOCA_DPA_DEV_LOG_ERR(...)
DOCA_DPA_DEV_LOG_WARN(...)
DOCA_DPA_DEV_LOG_INFO(...)
DOCA_DPA_DEV_LOG_DBG(...)
```

- To create a trace message entry with arguments:

```
void doca_dpa_dev_trace(uint64_t arg1, uint64_t arg2, uint64_t arg3, uint64_t arg4, uint64_t arg5)
```

- To flush the trace message buffer to host:

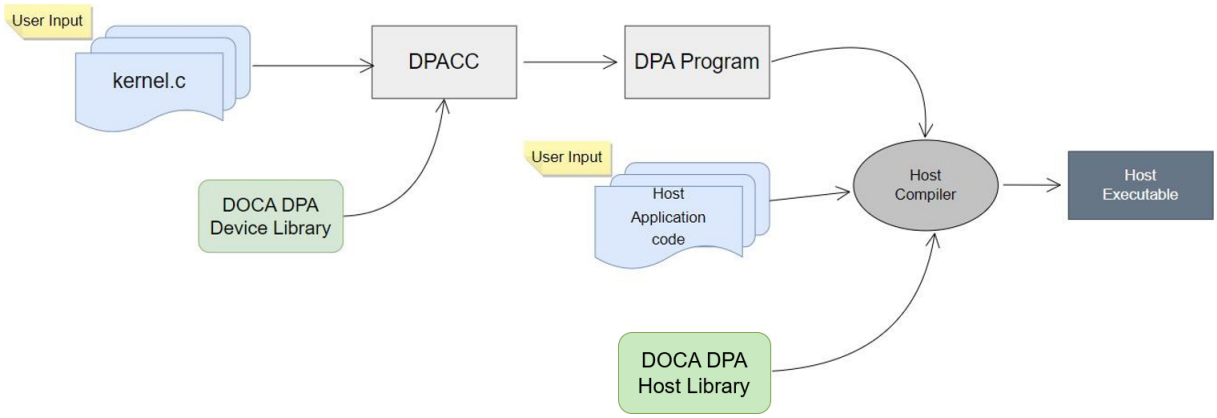
```
void doca_dpa_dev_trace_flush(void)
```

5.4.3.6.6 Hello World Example

> **Warning**
> The provided example code demonstrates the basic functionality and flow of the DPA but is not intended to be executed as-is due to missing initializations. For complete and runnable DPA examples, refer to the full samples in the DOCA DPA#Samples section.

5.4.3.6.6.1 Procedure Outline

1. Write DPA device code (i.e., kernels or `.c` files).
2. Use DPACC to build a DPA program (i.e., a host library which contains an embedded device executable). Input for DPACC:
    a. Kernels from step 1.
    b. DOCA DPA device library.
3. Build host executable using a host compiler. Input for the host compiler:
    a. DPA program.
    b. User host application `.c` / `.cpp` files.
4. Run host executable.



5.4.3.6.6.2 Procedure Steps

The following code snippets show a basic DPA code that eventually prints "Hello World" to **stdout**.

This is achieved using:

- A DPA Thread which prints the string and signals a DOCA Sync Event to indicate completion to host application
- A DPA RPC to notify DPA Thread

The steps are elaborated in the following subsections.

Prepare Kernels Code

```
#include "doca_dpa_dev.h"
#include "doca_dpa_dev_sync_event.h"

__dpa_global__ void hello_world_thread_kernel(uint64_t arg)
{
    DOCA_DPA_DEV_LOG_INFO("Hello World From DPA Thread!\n");
    doca_dpa_dev_sync_event_update_set(arg, 1);
    doca_dpa_dev_thread_finish();
}

__dpa_rpc__ uint64_t hello_world_thread_notify_rpc(doca_dpa_dev_notification_completion_t comp_handle)
{
    DOCA_DPA_DEV_LOG_INFO("Notifying DPA Thread From RPC\n");
    doca_dpa_dev_thread_notify(comp_handle);
    return 0;
}
```

Prepare Host Application Code

```
#include <stdio.h>
#include <unistd.h>
#include <doca_dev.h>
#include <doca_error.h>
#include <doca_sync_event.h>
#include <doca_dpa.h>

/**
 * A struct that includes all needed info on registered kernels and is initialized during linkage by DPACC.
 * Variable name should be the token passed to DPACC with --app-name parameter.
 */
extern struct doca_dpa_app *dpa_hello_world_app;

/**
 * kernel declaration that the user must declare for each kernel and DPACC is responsible to initialize.
 * Only then, user can use this variable in relevant host APIs
 */
doca_dpa_func_t hello_world_thread_kernel;
doca_dpa_func_t hello_world_thread_notify_rpc;

int main(int argc, char **argv)
{
    struct doca_dev *doca_dev = NULL;
    struct doca_dpa *dpa_ctx = NULL;
    struct doca_sync_event *cpu_se = NULL;
    doca_dpa_dev_sync_event_t cpu_se_handle = 0;
    struct doca_dpa_thread *dpa_thread = NULL;
    struct doca_dpa_notification_completion *notify_comp = NULL;
    doca_dpa_dev_notification_completion_t notify_comp_handle = 0;
    uint64_t retval = 0;

    printf("\n----> Open DOCA Device\n");
    /* Open appropriate DOCA device doca_dev */

    printf("\n----> Initialize DOCA DPA Context\n");
    doca_dpa_create(doca_dev, &dpa_ctx);
    doca_dpa_set_app(dpa_ctx, dpa_hello_world_app);
    doca_dpa_start(dpa_ctx);

    printf("\n----> Initialize DOCA Sync Event\n");
    doca_sync_event_create(&cpu_se);
    doca_sync_event_add_publisher_location_dpa(cpu_se, dpa_ctx);
    doca_sync_event_add_subscriber_location_cpu(cpu_se, doca_dev);
    doca_sync_event_start(cpu_se);
    doca_sync_event_get_dpa_handle(cpu_se, dpa_ctx, &cpu_se_handle);

    printf("\n----> Initialize DOCA DPA Thread\n");
    doca_dpa_thread_create(dpa_ctx, &dpa_thread);
    doca_dpa_thread_set_func_arg(dpa_thread, &hello_world_thread_kernel, cpu_se_handle);
    doca_dpa_thread_start(dpa_thread);

    printf("\n----> Initialize DOCA DPA Notification Completion\n");
```

```
    doca_dpa_notification_completion_create(dpa_ctx, dpa_thread, &notify_comp);
    doca_dpa_notification_completion_start(notify_comp);
    doca_dpa_notification_completion_get_dpa_handle(notify_comp, &notify_comp_handle);

    printf("\n----> Run DOCA DPA Thread\n");
    doca_dpa_thread_run(dpa_thread);

    printf("\n----> Trigger DPA RPC\n");
    doca_dpa_rpc(dpa_ctx, &hello_world_thread_notify_rpc, &retval, notify_comp_handle);

    printf("\n----> Waiting For hello_world_thread_kernel To Finish\n");
    doca_sync_event_wait_gt(cpu_se, 0, 0xFFFFFFFFFFFFFFFF);

    printf("\n----> Destroy DOCA DPA Notification Completion\n");
    doca_dpa_notification_completion_destroy(notify_comp);

    printf("\n----> Destroy DOCA DPA Thread\n");
    doca_dpa_thread_destroy(dpa_thread);

    printf("\n----> Destroy DOCA DPA event\n");
    doca_sync_event_destroy(cpu_se);

    printf("\n----> Destroy DOCA DPA context\n");
    doca_dpa_destroy(dpa_ctx);

    printf("\n----> Destroy DOCA device\n");
    doca_dev_close(doca_dev);

    printf("\n----> DONE!\n");
    return 0;
}
```

Build DPA Program

```
/opt/mellanox/doca/tools/dpacc \
    kernel.c \
    -o dpa_program.a \
    -hostcc=gcc \
    -hostcc-options="-Wno-deprecated-declarations -Werror -Wall -Wextra -W" \
    --devicecc-options="-D__linux__ -Wno-deprecated-declarations -Werror -Wall -Wextra -W" \
    --app-name="dpa_hello_world_app" \
    --mcpu=nv-dpa-bf3 \
    --device-libs="-L/opt/mellanox/doca/lib/x86_64-linux-gnu/ -ldoca_dpa_dev" \
    -I/opt/mellanox/doca/include/
```

Build Host Application

```
gcc hello_world.c -o hello_world \
    dpa_program.a \
    -I/opt/mellanox/doca/include/ \
    -DDOCA_ALLOW_EXPERIMENTAL_API \
    -L/opt/mellanox/doca/lib/x86_64-linux-gnu/ -ldoca_dpa -ldoca_common \
    -L/opt/mellanox/flexio/lib/ -lflexio \
    -lstdc++ -libverbs -lmlx5
```

Execution

```
$  ./hello_world

----> Open DOCA Device

----> Initialize DOCA DPA Context

----> Initialize DOCA Sync Event

----> Initialize DOCA DPA Thread

----> Initialize DOCA DPA Notification Completion

----> Run DOCA DPA Thread

----> Trigger DPA RPC
/ 10/[DOCA][DPA DEVICE][INF] Notifying DPA Thread From RPC
/  8/[DOCA][DPA DEVICE][INF] Hello World From DPA Thread!

----> Waiting For hello_world_thread_kernel To Finish

----> Destroy DOCA DPA Notification Completion

----> Destroy DOCA DPA Thread

----> Destroy DOCA DPA event

----> Destroy DOCA DPA context

----> Destroy DOCA device

----> DONE!
```

5.4.3.6.7 Samples

This section provides DPA sample implementation on top of the BlueField-3 networking platform.

> All DOCA samples in this section are provided under the BSD-3 software license and are also available on GitHub.

DPA samples can be executed from either the host or the DPU.

When running from the DPU and using RDMA utilities, the RDMA DOCA device (SF device) must be provided as a parameter. If not specified, a random RDMA device is selected.

In this case:

- The DOCA DPA context is created on the PF device.
- An extended DPA context is created on the SF device.
- The DOCA RDMA context is created on the SF device.

> All DPA resources (e.g., DPA threads, DPA completion contexts) associated with the SF RDMA instance must be created using the extended DPA context.

To run DPA samples:

1. Refer to the following documents:
   - *DOCA Installation Guide for Linux* for details on how to install BlueField-related software.
   - *NVIDIA BlueField Platform Software Troubleshooting Guide* for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

   ```
   cd /opt/mellanox/doca/samples/doca_dpa/<sample_name>
   meson /tmp/build
   ninja -C /tmp/build
   ```

   > The binary **doca_<sample_name>** is created under **/tmp/build/**.

3. Sample (e.g., **dpa_initiator_target**) usage:

   ```
   Usage: doca_dpa_initiator_target [DOCA Flags] [Program Flags]

   DOCA Flags:
     -h, --help                        Print a help synopsis
     -v, --version                     Print program version information
     -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
   30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
     --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
   20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
     -j, --json <path>                 Parse command line flags from an input json file

   Program Flags:
     -pf_dev, --pf-device <PF DOCA device name> PF device name that supports DPA (optional). If not provided
   then a random device will be chosen
   -rdma_dev, --rdma-device <RDMA DOCA device name> device name that supports RDMA (optional). If not provided
   then a random device will be chosen
     --dpa-resources <DPA resources file> Path to a DPA resources .yaml file
     --dpa-app-key <DPA application key> Application key in specified DPA resources .yaml file
     -g, --gid-index                   GID index for DOCA RDMA (optional)
   ```

4. For additional information per sample, use the `-h` option:

```
/tmp/build/doca_<sample_name> -h
```

5.4.3.6.7.1 Basic Initiator Target

This sample illustrates how to trigger DPA Thread using DPA Completion Context attached to DOCA RDMA.

This sample consists of initiator and target endpoints.

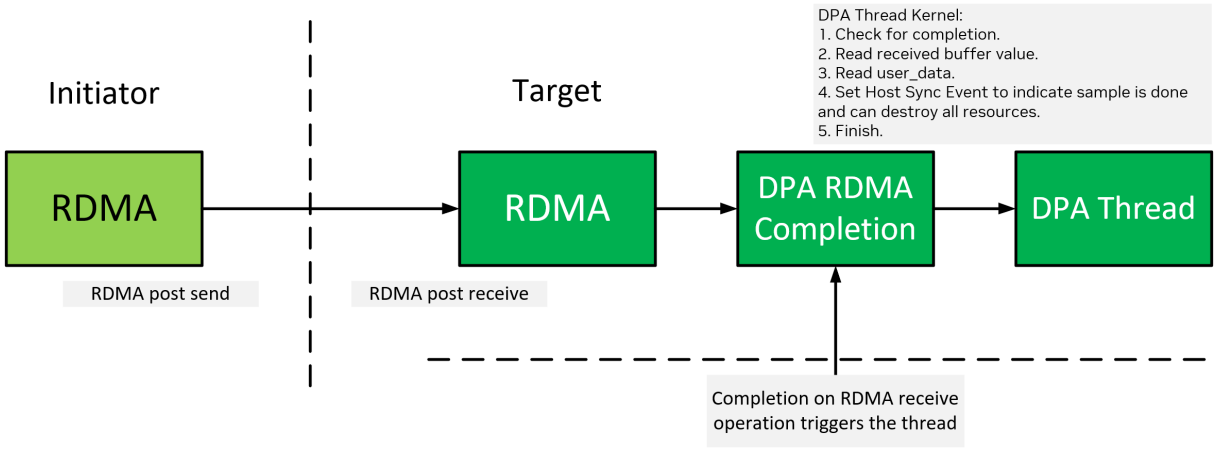
In the initiator endpoint, a DOCA RDMA executes RDMA post send operation using DPA RPC.

In the target endpoint, a DOCA RDMA, attached to DPA Completion Context, executes RDMA post receive operation using DPA RPC.

Completion on the post receive operation triggers DPA Thread which prints completion info and sets DOCA Sync Event to release the host application that waits on that event before destroying all resources and finish.

The sample logic includes:

1. Allocating DOCA DPA and DOCA RDMA resources for both initiator and target endpoints.
2. Target – Attaching DOCA RDMA to DPA Completion Context which is attached to DPA Thread.
3. Running DPA thread.
4. Target – DPA RPC to execute RDMA post receive operation.
5. Initiator – DPA RPC to execute RDMA post send operation.
6. The completion on the post receive operation triggers DPA Thread.
7. Waiting on completion event to be set from DPA Thread.
8. Destroying all resources.



Reference:

- **/opt/mellanox/doca/samples/doca_dpa/dpa_basic_initiator_target/dpa_basic_initiator_target_main.c**

- `/opt/mellanox/doca/samples/doca_dpa/dpa_basic_initiator_target/host/dpa_basic_initiator_target_sample.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_basic_initiator_target/device/dpa_basic_initiator_target_kernels_dev.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_basic_initiator_target/device/dpa_basic_initiator_target_attributes.yaml`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_basic_initiator_target/meson.build`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.h`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.c`
- `/opt/mellanox/doca/samples/doca_dpa/build_dpacc_samples.sh`

5.4.3.6.7.2 Advanced Initiator Target

This sample illustrates how to trigger DPA threads using both DPA Notification Completion and DPA Completion Context which is attached to a DOCA RDMA context with 2 connections.

This sample consists of initiator and target endpoints.

In the initiator endpoint, two DOCA RDMA connections execute an RDMA post send operation using DPA RPC in the following order:

1. Connection #0 executes the RDMA post send operation on buffer with value 1.
2. Connection #1 executes the RDMA post send operation on buffer with value 2.
3. Connection #0 executes the RDMA post send operation on buffer with value 3.
4. Connection #1 executes the RDMA post send operation on buffer with value 4.

In the target endpoint, DOCA RDMA context with two connections is attached to a single DPA Completion Context which is attached to DPA Thread #1.

Target RDMA executes the initial RDMA post receive operation using DPA RPC.

Completions on the post receive operations trigger DPA Thread #1 which:

1. Prints completion info including user data.
2. Updates a local data base with the receive buffer value.
3. Repost RDMA receive operation.
4. Ack, request completion and reschedule.

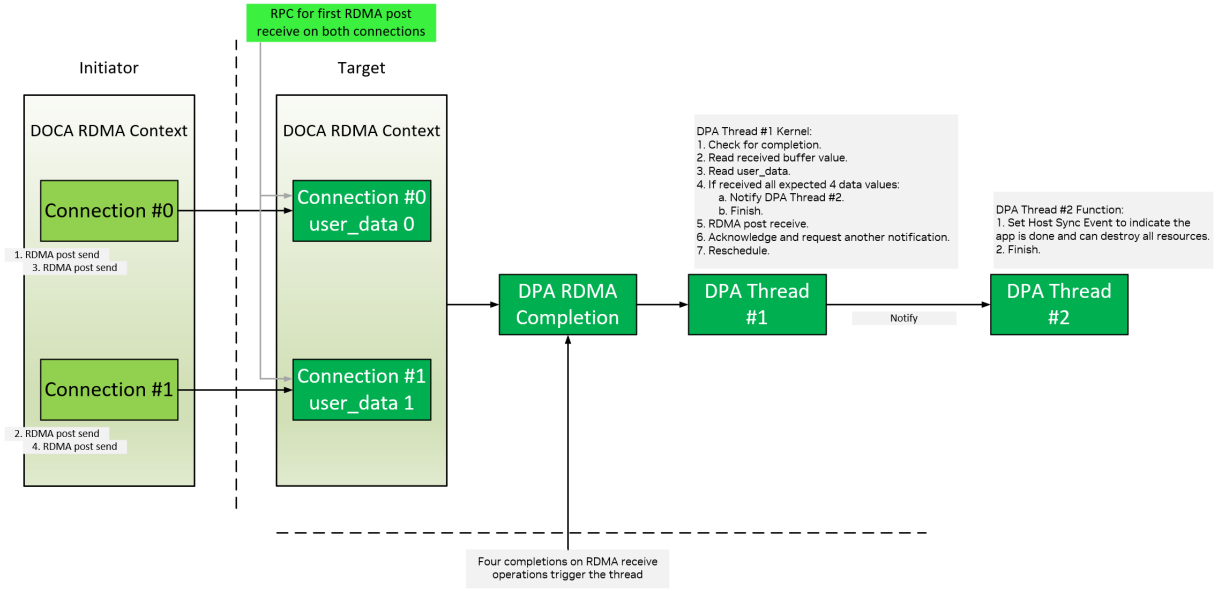
Once target DPA Thread #1 receives all expected values "1, 2, 3, 4", it notifies DPA Thread #2 and finishes.

Once DPA Thread #2 is triggered, it sets DOCA Sync Event to release the host application that waits on that event before destroying all resources and finishing.

The sample logic includes:

1. Allocating DOCA DPA and DOCA RDMA resources for both initiator and target endpoints.
2. Target: Attaching DOCA RDMA to DPA Completion Context which is attached to DPA Thread #1.
3. Target: Attaching DPA Notification Completion to DPA Thread #2.
4. Run DPA threads.
5. Target: DPA RPC to execute the initial RDMA post receive operation.
6. Initiator: DPA RPC to execute all RDMA post send operations.

7. Completions on the post receive operations (4 completions) trigger DPA Thread #1.
8. Once all expected values are received, DPA Thread #1 notifies DPA Thread #2 and finishes.
9. Waiting on completion event to be set from DPA Thread #2.
10. Destroying all resources.



Reference:

- **/opt/mellanox/doca/samples/doca_dpa/dpa_initiator_target/dpa_initiator_target_main.c**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_initiator_target/host/dpa_initiator_target_sample.c**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_initiator_target/device/dpa_initiator_target_kernels_dev.c**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_initiator_target/device/dpa_initiator_target_attributes.yaml**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_initiator_target/meson.build**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_common.h**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_common.c**
- **/opt/mellanox/doca/samples/doca_dpa/build_dpacc_samples.sh**

5.4.3.6.7.3 Ping Pong

This sample illustrates the functionality of the following DPA objects:

- DPA Thread
- DPA Completion Context
- DOCA RDMA

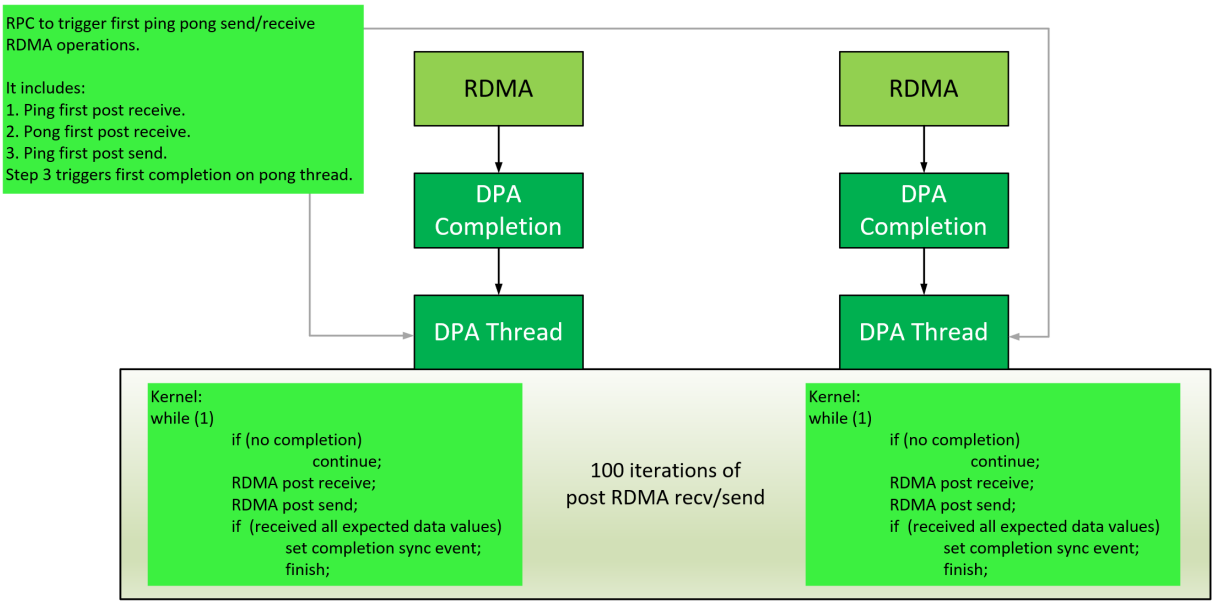
This sample consists of ping and pong endpoints which run for 100 iterations. On each iteration, DPA threads (ping and pong) post RDMA receive and send operations for data buffers with values [0-99].

Once all expected values are received on each DPA thread, it sets a DOCA Sync Event to release the host application waiting on that event before destroying all resources and finishes.

To trigger DPA threads, the sample uses a DPA RPC.

The sample logic includes:

1. Allocating DOCA DPA and DOCA RDMA resources.
2. Attaching DOCA RDMA to DPA completion context which is attached to DPA thread.
3. Run DPA threads.
4. DPA RPC to trigger DPA threads.
5. 100 ping pong iterations of RDMA post receive and send operations.
6. Waiting on completion events to be set from DPA threads.
7. Destroying all resources.



Reference:

- **/opt/mellanox/doca/samples/doca_dpa/dpa_ping_pong/dpa_ping_pong_main.c**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_ping_pong/host/dpa_ping_pong_sample.c**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_ping_pong/device/dpa_ping_pong_kernels_dev.c**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_ping_pong/device/dpa_ping_pong_attributes.yaml**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_ping_pong/meson.build**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_common.h**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_common.c**
- **/opt/mellanox/doca/samples/doca_dpa/build_dpacc_samples.sh**

5.4.3.6.7.4 Kernel Launch

This sample illustrates how to launch a DOCA DPA kernel with wait and completion DOCA sync events.

The sample logic includes:

1. Allocating DOCA DPA resources.
2. Initializing wait and completion DOCA sync events for the DOCA DPA kernel.
3. Running **hello_world** DOCA DPA kernel that waits on the wait event.
4. Running a separate thread that triggers the wait event.
5. **hello_world** DOCA DPA kernel prints "Hello from kernel".
6. Waiting for the completion event of the kernel.
7. Destroying the events and resources.

Reference:

- **/opt/mellanox/doca/samples/doca_dpa/dpa_wait_kernel_launch/dpa_wait_kernel_launch_main.c**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_wait_kernel_launch/host/dpa_wait_kernel_launch_sample.c**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_wait_kernel_launch/device/dpa_wait_kernel_launch_kernels_dev.c**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_wait_kernel_launch/device/dpa_wait_kernel_launch_attributes.yaml**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_wait_kernel_launch/meson.build**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_common.h**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_common.c**
- **/opt/mellanox/doca/samples/doca_dpa/build_dpacc_samples.sh**

5.4.3.6.7.5 NVQual

This sample illustrates a stressor for DOCA DPA by running multiple DPA threads on all available EUs simultaneously. Some EUs perform arithmetic operations, and some perform memory stress operations while measuring the total DPA run time.

The sample logic includes:

1. Allocating DOCA DPA resources, including:
    a. Initializing DOCA sync event to synchronize host and DPA between iterations.
    b. Initializing and configuring DPA threads for each available EU.
    c. Initializing a notification completion for each thread to schedule the thread on available EU in every iteration.
2. Running the following loop, while sufficient time remains:
    a. Notify DPA threads by DPA RPC.
    b. Threads perform memory and arithmetic operations according to their assigned kernels.
    c. Wait for all the completion events of all threads.
    d. Calculate average latency of copying a single byte to/from the DPA heap.
3. Printing sample data summary.
4. Destroying all resources.

Sample usage:

```
Usage: dpa_nvqual [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
```

```
  -l, --log-level                  Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                  Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                Parse command line flags from an input json file

Program Flags:
  -pf_dev, --pf-device <PF DOCA device name> PF device name that supports DPA (mandatory).
  -test_dur, --test-duration-sec <test duration sec> test duration seconds (mandatory).
  -user_fac, --user-factor [user factor] user factor from type float (optional). If not provided then 0.75f be
chosen.
  -ex_eu, --excluded-eus [excluded eus] excluded eus list, devided by ',' with no spaced (optional). If not
provided then no eu will be excluded.
```

Use smaller user factor value (e.g., 0.6) when running the sample on NVIDIA® ConnectX®-8 and use higher user factor value (e.g., 0.8) when running it on NVIDIA® ConnectX®-9.

Reference:

- **/opt/mellanox/doca/samples/doca_dpa/dpa_nvqual/dpa_nvqual_main.c**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_nvqual/host/dpa_nvqual_sample.c**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_nvqual/device/dpa_nvqual_kernels_dev.c**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_nvqual/device/dpa_nvqual_attributes.yaml**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_nvqual/meson.build**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_common.h**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_common.c**
- **/opt/mellanox/doca/samples/doca_dpa/build_dpacc_samples.sh**

5.4.3.6.7.6 Verbs Initiator Target

This sample illustrates how to perform RDMA operations using DOCA Verbs API.

Sample logic:

1. Allocating DOCA DPA resources:
    - Initializing and configuring DPA threads and kernels.
    - Initializing a buffer on the DPA heap for Verbs operations and attaching mmap.
    - Initializing a DPA completion context to replace CQ.
    - Initializing a DOCA sync event to synchronize the host and DPA when the data path is finished.
2. Allocating Verbs resources:
    - Verbs context
    - Verbs QP attached to the DPA completion context
    - Verbs PD
    - Verbs AH
3. Establishing a connection between the initiator and target using a TCP socket.
4. Exchanging RDMA parameters:
    - Local buffer addresses
    - MKEYs
    - QP numbers
    - GID addresses
5. Configuring QP attributes and modifying QP states from Reset to Ready to Send.
6. Data path:
    - Initiator:
        - Waits for completion

        - Polls the local buffer value to sync with the target's Write operation
        - Increments the buffer value
        - Posts a Send work request
        - Acknowledges completion
    - Target:
        - Waits for completions
        - Posts a Receive work request
        - Increments the buffer value
        - Posts a Write work request
        - Acknowledges 2 completions
7. Verifying the final buffer value (initiator).
8. Updating the sync event to threshold value (both initiator and target).
9. Printing sample data summary.
10. Destroying all resources.

Sample usage:

```
Usage: dpa_verbs_initiator_target [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>

Program Flags:
  -i, --initiator <Sample is initiator, requires target OOB IP> Sample is initiator, requires target OOB IP
  -d, --device <PF device name>     PF device name
  -sf_dev, --sf-device <SF device name> SF device name that supports RDMA.
  -gid, --gid-index                 GID index for DOCA RDMA (optional)
```

Reference:

- **/opt/mellanox/doca/samples/doca_dpa/dpa_verbs_initiator_target/dpa_verbs_initiator_target_main.c**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_verbs_initiator_target/host/dpa_verbs_initiator_target_sample.c**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_verbs_initiator_target/device/dpa_verbs_initiator_target_kernels_dev.c**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_verbs_initiator_target/device/dpa_verbs_initiator_target_attributes.yaml**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_verbs_initiator_target/common/dpa_verbs_initiator_target_common_defs.h**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_verbs_initiator_target/meson.build**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_common.h**
- **/opt/mellanox/doca/samples/doca_dpa/dpa_common.c**
- **/opt/mellanox/doca/samples/doca_dpa/build_dpacc_samples.sh**

5.4.3.7 DOCA DPA Comms

5.4.3.7.1 Introduction

> ⚠ This feature is supported at the beta level.

The DOCA DPA device communication library provides a set of communication utilities (e.g., RDMA) to facilitate data transfers between nodes using efficient communication primitives.

The primary communication object is called an RDMA DPA handle. It represents a unidirectional communication pipe between two nodes and is exclusively accessible from the kernel. These handles are established by setting a DOCA RDMA context on the DPA data path.

> ℹ For more information on configuring and using RDMA contexts, refer to DOCA RDMA documentation.

To monitor the completion of communication operations, users can attach a DOCA RDMA context to a DPA Completion Context.

There are two options for tracking completion events:

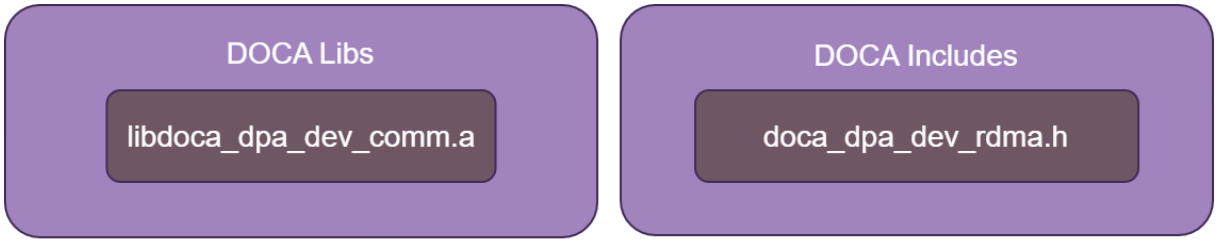
- DPA thread association – Users can associate the Completion Context with a DPA Thread, which is automatically triggered when a communication operation completes.
- Manual polling – Alternatively, users may manually poll the Completion Context without assigning it to a DPA Thread. This approach provides greater control in scenarios where asynchronous event management is not required.

This flexible design allows users to optimize their applications based on specific performance and operational needs.

5.4.3.7.2 Architecture

> ⚠ The DOCA DPA SDK does not implement any multi-thread synchronization primitives. All DOCA DPA objects are non-thread-safe. Developers must ensure that both the user program and kernels are designed to prevent race conditions.

DOCA Comms library and header files:



5.4.3.7.2.1 Relevant Host-side APIs

- To create a DOCA RDMA context on DPA, the following API must be used for the DOCA RDMA context:

```
doca_error_t doca_ctx_set_datapath_on_dpa(struct doca_ctx *ctx, struct doca_dpa *dpa)
```

> When running from BlueField and creating DOCA RDMA using an SF DOCA device, or when creating a DOCA RDMA context on DPA, an extended DPA context (created on the same SF DOCA device) must be provided to the **doca_ctx_set_datapath_on_dpa()** API.

- To attach a DOCA RDMA context to a DPA completion context:

```
doca_error_t doca_rdma_dpa_completion_attach(struct doca_rdma *rdma, struct doca_dpa_completion *dpa_comp)
```

- To obtain a DPA RDMA handle and a connection ID:

```
doca_error_t doca_rdma_get_dpa_handle(struct doca_rdma *rdma, doca_dpa_dev_rdma_t *dpa_rdma)
doca_error_t doca_rdma_connection_get_id(const struct doca_rdma_connection *rdma_connection, uint32_t
*connection_id)
```

Use output parameters **dpa_handle** and **connection_id** in relevant device APIs in the thread kernel.

> DPA RDMAs are not thread safe and, therefore, must not be used from different kernels/threads concurrently.

- To support multiple DOCA RDMA connection management in DPA, DOCA offers the following APIs:
  - To get the RDMA connection ID using the host API:

```
doca_error_t doca_rdma_connection_get_id(const struct doca_rdma_connection *rdma_connection,
uint32_t *connection_id)
```

This can also be retrieved in the device using the completion API:

```
uint32_t doca_dpa_dev_get_completion_user_data(doca_dpa_dev_completion_element_t comp_element)
```

The returned value should be used to distinguish which DOCA RDMA connection has triggered this completion.

  - To retrieve the RDMA work request index for an RDMA completion:

```
uint32_t doca_dpa_dev_rdma_completion_get_wr_index(doca_dpa_dev_completion_element_t comp_element)
```

work request index should be used to get the operation index of the DOCA RDMA context which triggered this completion.

5.4.3.7.2.2 Device-side APIs

DOCA DPA offers two work models for each device RDMA operation:

- An API for RDMA operation using DOCA buffer
- An API for RDMA operation using DOCA mmap and an explicit memory address

The user may control **work request submit** configuration using the following enum:

```
/**
 * @brief DPA submit flag type
 */
__dpa_global__ enum doca_dpa_dev_submit_flag {
    DOCA_DPA_DEV_SUBMIT_FLAG_NONE = 0U,
    DOCA_DPA_DEV_SUBMIT_FLAG_FLUSH = (1U << 0),           /**
                                * Use flag to inform related DPA context
                                * (such as RDMA or DPA Async ops) to flush related
                                * operation and previous operations to HW,
                                * otherwise the context may aggregate the operation
                                * and not flush it immediately
                                */
    DOCA_DPA_DEV_SUBMIT_FLAG_OPTIMIZE_REPORTS = (1U << 1), /**
                                * Use flag to inform related DPA context that it may
                                * defer completion of the operation to a later time. If
                                * flag is not provided then a completion will be raised
                                * as soon as the operation is finished, and any
                                * preceding completions that were deferred will also be
                                * raised. Use this flag to optimize the amount of
                                * completion notifications it receives from HW when
                                * submitting a batch of operations, by receiving a
                                * single completion notification on the entire batch.
                                */
};
```

This enum can be used in the **flags** parameter in the following device APIs:

- To aggregate multiple operations and flush them at once to hardware using the **DOCA_DPA_DEV_SUBMIT_FLAG_FLUSH** flag:

  ```
  doca_dpa_dev_rdma_post_write(..., DOCA_DPA_DEV_SUBMIT_FLAG_NONE); // This operation is not flushed to HW as
  DOCA_DPA_DEV_SUBMIT_FLAG_FLUSH was not used
  doca_dpa_dev_rdma_post_write(..., DOCA_DPA_DEV_SUBMIT_FLAG_NONE); // This operation is not flushed to HW as
  DOCA_DPA_DEV_SUBMIT_FLAG_FLUSH was not used
  doca_dpa_dev_rdma_post_write(..., DOCA_DPA_DEV_SUBMIT_FLAG_FLUSH); // This operation is flushed to HW
  together with all non-flushed previous operations as DOCA_DPA_DEV_SUBMIT_FLAG_FLUSH was used
  ```

- To defer completion of an operation to a later time using the **DOCA_DPA_DEV_SUBMIT_FLAG_OPTIMIZE_REPORTS** flag:

  ```
  doca_dpa_dev_rdma_post_write(..., DOCA_DPA_DEV_SUBMIT_FLAG_OPTIMIZE_REPORTS); // This operation is not
  flushed to HW as DOCA_DPA_DEV_SUBMIT_FLAG_FLUSH was not used
                                                                                 // and will not trigger a
  completion as DOCA_DPA_DEV_SUBMIT_FLAG_OPTIMIZE_REPORTS was used
  doca_dpa_dev_rdma_post_write(..., DOCA_DPA_DEV_SUBMIT_FLAG_OPTIMIZE_REPORTS); // This operation is not
  flushed to HW as DOCA_DPA_DEV_SUBMIT_FLAG_FLUSH was not used
                                                                                 // and will not trigger a
  completion as DOCA_DPA_DEV_SUBMIT_FLAG_OPTIMIZE_REPORTS was used
  doca_dpa_dev_rdma_post_write(..., DOCA_DPA_DEV_SUBMIT_FLAG_FLUSH); // This operation is flushed to HW
  together with all non-flushed previous operations as
                                                                    // DOCA_DPA_DEV_SUBMIT_FLAG_FLUSH was
  used.
                                                                    // in addition this operation will
  trigger a completion once its done as DOCA_DPA_DEV_SUBMIT_FLAG_OPTIMIZE_REPORTS was NOT used
  ```

- To read to a local buffer from the remote side buffer:

  ```
  void doca_dpa_dev_rdma_post_read(doca_dpa_dev_rdma_t rdma,
                                   uint32_t connection_id,
                                   doca_dpa_dev_mmap_t dst_mmap_handle,
                                   uint64_t dst_addr,
                                   doca_dpa_dev_mmap_t src_mmap_handle,
                                   uint64_t src_addr,
                                   size_t length,
                                   uint32_t flags)
  void doca_dpa_dev_rdma_post_buf_read(doca_dpa_dev_rdma_t rdma,
                                       uint32_t connection_id,
                                       doca_dpa_dev_buf_t dst_buf_handle,
                                       doca_dpa_dev_buf_t src_buf_handle,
  ```

```
                                    uint32_t flags)
```

- To write local memory to the remote side buffer:

```
void doca_dpa_dev_rdma_post_write(doca_dpa_dev_rdma_t rdma,
                                  uint32_t connection_id,
                                  doca_dpa_dev_mmap_t dst_mmap_handle,
                                  uint64_t dst_addr,
                                  doca_dpa_dev_mmap_t src_mmap_handle,
                                  uint64_t src_addr,
                                  size_t length,
                                  uint32_t flags)
void doca_dpa_dev_rdma_post_buf_write(doca_dpa_dev_rdma_t rdma,
                                      uint32_t connection_id,
                                      doca_dpa_dev_buf_t dst_buf_handle,
                                      doca_dpa_dev_buf_t src_buf_handle,
                                      uint32_t flags)
```

- To write local memory to the remote buffer and include immediate data which can be retrieved upon receiving a completion for this operation:

```
void doca_dpa_dev_rdma_post_write_imm(doca_dpa_dev_rdma_t rdma,
                                      uint32_t connection_id,
                                      doca_dpa_dev_mmap_t dst_mmap_handle,
                                      uint64_t dst_addr,
                                      doca_dpa_dev_mmap_t src_mmap_handle,
                                      uint64_t src_addr,
                                      size_t length,
                                      uint32_t immediate,
                                      uint32_t flags)
void doca_dpa_dev_rdma_post_buf_write_imm(doca_dpa_dev_rdma_t rdma,
                                          uint32_t connection_id,
                                          doca_dpa_dev_buf_t dst_buf_handle,
                                          doca_dpa_dev_buf_t src_buf_handle,
                                          uint32_t immediate,
                                          uint32_t flags)
// use the following API to retrieve immediate data on completion
uint32_t doca_dpa_dev_get_completion_immediate(doca_dpa_dev_completion_element_t comp_element)
```

- To send local memory:

```
void doca_dpa_dev_rdma_post_send(doca_dpa_dev_rdma_t rdma,
                                 uint32_t connection_id,
                                 doca_dpa_dev_mmap_t mmap_handle,
                                 uint64_t addr,
                                 size_t length,
                                 uint32_t flags)
void doca_dpa_dev_rdma_post_buf_send(doca_dpa_dev_rdma_t rdma,
                                     uint32_t connection_id,
                                     doca_dpa_dev_buf_t send_buf_handle,
                                     uint32_t flags)
```

- To send local memory with an immediate data which can be retrieved when receiving a completion for this operation:

```
void doca_dpa_dev_rdma_post_send_imm(doca_dpa_dev_rdma_t rdma,
                                     uint32_t connection_id,
                                     doca_dpa_dev_mmap_t mmap_handle,
                                     uint64_t addr,
                                     size_t length,
                                     uint32_t immediate,
                                     uint32_t flags)
void doca_dpa_dev_rdma_post_buf_send_imm(doca_dpa_dev_rdma_t rdma,
                                         uint32_t connection_id,
                                         doca_dpa_dev_buf_t send_buf_handle,
                                         uint32_t immediate,
                                         uint32_t flags)
// use the following API to retrieve immediate data on completion
uint32_t doca_dpa_dev_get_completion_immediate(doca_dpa_dev_completion_element_t comp_element)
```

- To handle posting an RDMA receive operation, use the following APIs:
  - To post RDMA receive operation:

    ```
    void doca_dpa_dev_rdma_post_receive(doca_dpa_dev_rdma_t rdma, doca_dpa_dev_mmap_t mmap_handle,
    uint64_t addr, size_t length)
    void doca_dpa_dev_rdma_post_buf_receive(doca_dpa_dev_rdma_t rdma, doca_dpa_dev_buf_t
    receive_buf_handle)
    ```

  - To acknowledge that post receive operations are done (data has been received on associated data buffers):

```
void doca_dpa_dev_rdma_receive_ack(doca_dpa_dev_rdma_t rdma, uint32_t num_acked)
```

This operation frees **num_acked** entries in the DOCA RDMA context.

- To perform an atomic add operation on the remote buffer:

```
void doca_dpa_dev_rdma_post_atomic_fetch_add(doca_dpa_dev_rdma_t rdma,
                                             uint32_t connection_id,
                                             doca_dpa_dev_mmap_t dst_mmap_handle,
                                             uint64_t dst_addr,
                                             uint64_t value,
                                             uint32_t flags)
void doca_dpa_dev_rdma_post_buf_atomic_fetch_add(doca_dpa_dev_rdma_t rdma,
                                                 uint32_t connection_id,
                                                 doca_dpa_dev_buf_t dst_buf_handle,
                                                 uint64_t value,
                                                 uint32_t flags)
```

- To signal a remote event:

```
doca_dpa_dev_rdma_signal_<add|set>(doca_dpa_dev_rdma_t rdma, uint32_t connection_id,
doca_dpa_dev_sync_event_remote_t remote_sync_event, uint64_t count)
```

5.4.3.8 DOCA DPA Verbs

5.4.3.8.1 Overview

The DOCA DPA Device Verbs library provides RDMA operations support for DPA applications, enabling high-performance RDMA operations (Send, Receive, Read, Write, Atomic) directly on the DPA device without CPU involvement.

The library follows an RDMA-core/ibverbs-like API design pattern, making it intuitive for developers experienced with traditional RDMA programming. It maintains the same conceptual model of work requests (WRs), queue pairs (QPs), and completion queues (QPs) but is optimized for execution within DPA kernels.

The library uses a WR-based model for posting RDMA operations:

- Send WRs (`doca_dpa_dev_verbs_send_wr`) – For outbound RDMA operations, including Send, RDMA Write, RDMA Read, and Atomic operations.
- Receive WRs (`doca_dpa_dev_verbs_recv_wr`) – For posting receive buffers for incoming data.

Key features:

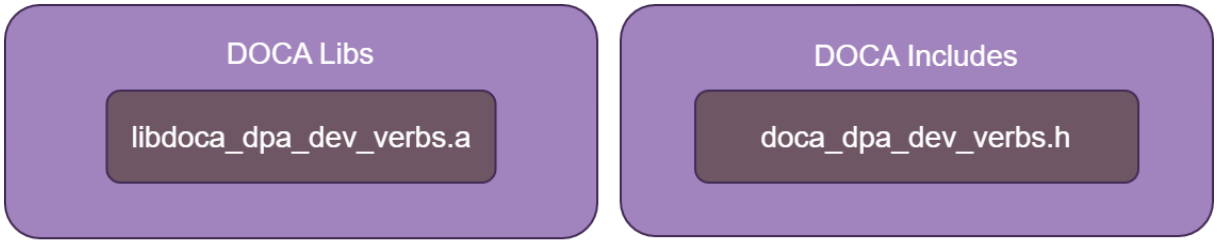
- DPA-native RDMA operations – Execute RDMA operations directly on the DPA device.
- RDMA-core compatible interface – Familiar API design following rdma-core/ibverbs patterns.
- WR-based model – Post send and receive WRs similar to traditional RDMA programming.
- Support for QPs and SRQs – Enable QPs and Shared Receive Queues (SRQ).
- Scatter-gather (SG) operations – Support complex memory layouts with multiple regions per operation.

5.4.3.8.2 Deployment

The DOCA DPA Verbs library and header files are:

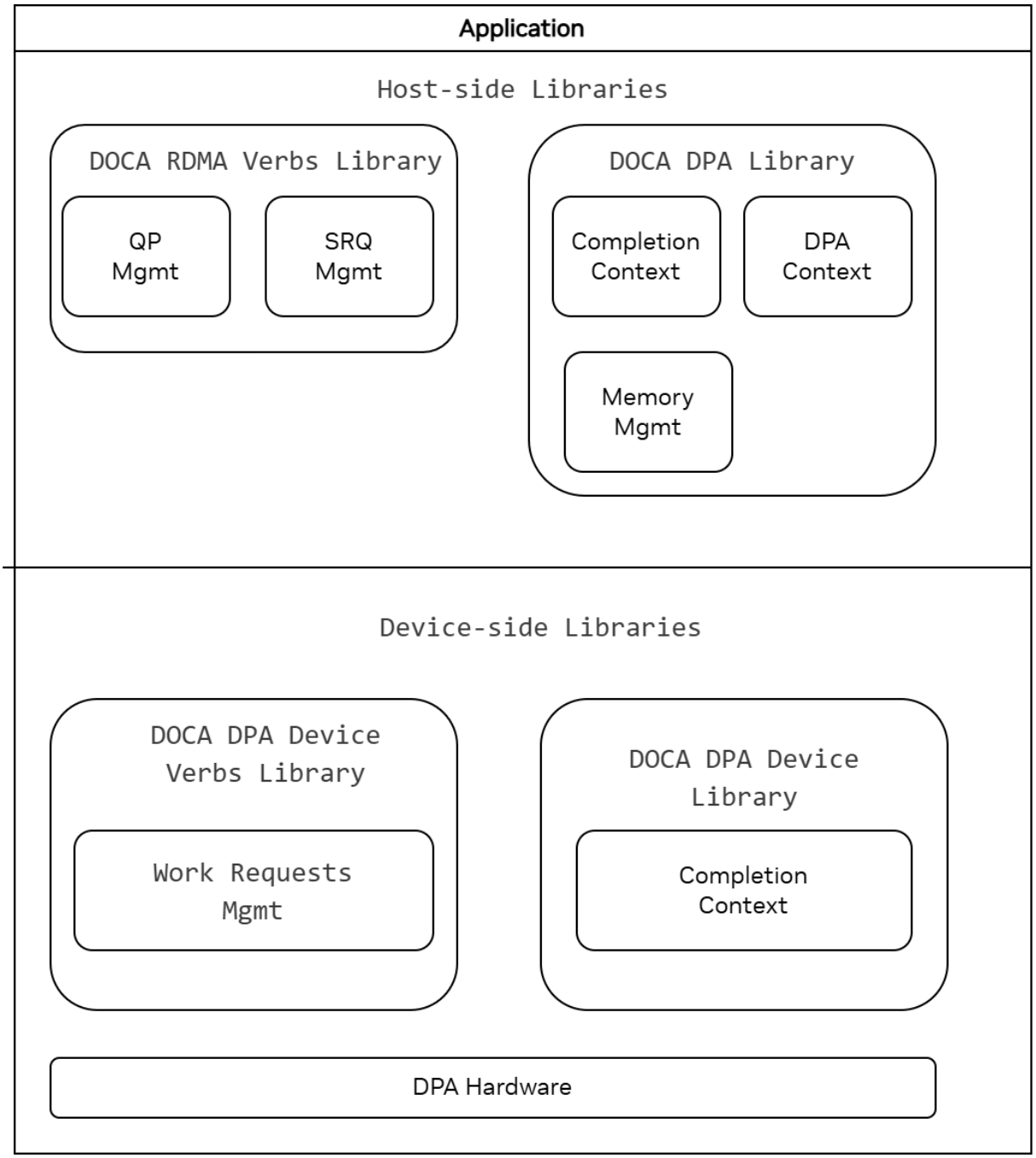
- `libdoca_dpa_dev_verbs.a` (DOCA Libs)

- `doca_dpa_dev_verbs.h` (DOCA Includes)



5.4.3.8.3 Architecture

> The DOCA DPA SDK does not implement multi-thread synchronization primitives, and all DOCA DPA objects are non-thread-safe. Developers must ensure that both the user program and kernels are designed to prevent race conditions.



Component breakdown:

1. Host-side libraries:
    - DOCA RDMA Verbs Library: Handles QP/SRQ creation, configuration, and lifecycle management.
    - DOCA DPA Library: Manages DPA context creation, DPA completion contexts, threads, memory management, and overall DPA orchestration.
2. DPA device libraries:
    - DPA Device Verbs Library: Device-side library for direct RDMA operations.

- DPA Device Library: Device-side completion context processing and completion element handling.

3. DPA hardware: The underlying DPA device that executes operations.

5.4.3.8.4 Core Concepts

5.4.3.8.4.1 DPA Device Handles

The DOCA DPA Verbs library uses specific handles to represent DPA device resources:

- `doca_dpa_dev_verbs_qp_t`: Handle for a QP on the DPA device.
- `doca_dpa_dev_verbs_srq_t`: Handle for a SRQ on the DPA device.
- `doca_dpa_dev_completion_t`: Handle for a CQ on the DPA device.

5.4.3.8.4.2 Work Request Structures

The library defines structures for WRs:

- `doca_dpa_dev_verbs_send_wr`: Structure for send WRs.
- `doca_dpa_dev_verbs_recv_wr`: Structure for receive WRs.
- `doca_dpa_dev_verbs_sge`: SG element for describing memory regions.

These structures are used to configure and post WRs to QPs or SRQs.

5.4.3.8.4.3 Operation Types

The DOCA DPA Verbs library supports various RDMA operation types:

- Send operations: `SEND` and `SEND_WITH_IMM`
- RDMA operations: `RDMA_WRITE`, `RDMA_WRITE_WITH_IMM`, and `RDMA_READ`
- Atomic operations: `ATOMIC_FETCH_ADD`

5.4.3.8.4.4 DPA Completion Context Management

The library provides a completion context management system that spans both host and device sides:

- Host-side completion contexts (`struct doca_dpa_completion`): Managed by the DOCA DPA Library, with configurable queue size, thread attachment, and lifecycle control.
- Device-side completion context processing: Utilizes the `doca_dpa_dev_completion_t` handle for processing completions on the DPA device.
- DPA completion elements (`doca_dpa_dev_completion_element_t`): Represent individual completion events containing metadata about completed operations.
- Supported completion types include send, receive (RDMA Write with Immediate, Send, Send with Immediate), and error completions.

5.4.3.8.5 API Reference

5.4.3.8.5.1 Enumerations and Constants

The `doca_dpa_dev_verbs.h` header file contains complete enum definitions and values.

Send Work Request Opcodes

The `enum doca_dpa_dev_verbs_send_wr_opcode` defines RDMA operation types:

- `WRITE`: One-sided RDMA write operation.
- `WRITE_WITH_IMM`: RDMA write with immediate data, generating a completion on the receiver.
- `SEND`: Two-sided send operation requiring a posted receive buffer on the remote side.
- `SEND_WITH_IMM`: Send operation with immediate data attached.
- `READ`: One-sided RDMA read operation retrieving data from remote memory.
- `ATOMIC_FETCH_ADD`: Atomic fetch-and-add operation on a remote memory location.

Send Work Request Flags

The `enum doca_dpa_dev_verbs_send_wr_flags` controls WR behavior:

- `SIGNALED`: Generates a CQ entry (CQE) when the operation completes.
- `SOLICITED`: Requests a solicited event on the remote side (used with send operations).

Fence Modes

The `enum doca_dpa_dev_verbs_send_wr_fm` controls ordering and synchronization between WRs:

- `NO_FENCE`: No ordering constraints for maximum performance.
- `INITIATOR_SMALL_FENCE`: Light ordering constraint for local operations.
- `FENCE`: Standard fence ensuring previous operations complete before this one.
- `STRONG_ORDERING_FENCE`: Strongest ordering guarantee for critical operations.
- `FENCE_AND_INITIATOR_SMALL_FENCE`: Combined fence modes for specific use cases.

SRQ Types

The `enum doca_dpa_dpa_dev_verbs_srq_type` defines SRQ implementation types:

- `LINKED_LIST`: SRQ implemented as a linked list structure.
- `CONTIGUOUS`: SRQ implemented as a contiguous memory buffer.

SG Element (SGE) Structure

The `struct doca_dpa_dev_verbs_sge` represents a memory region for data transfer:

- `addr`: Virtual address of the memory region (**`uint64_t`**).
- `length`: Length of the memory region in bytes (**`uint32_t`**).
- `lkey`: Local key for the memory region (**`uint32_t`**).

For non-fully occupied SG lists, set the last entry's `lkey` field to `DOCA_DPA_DEV_VERBS_SGE_TERMINATING_LKEY` (0x100) to indicate the end of valid entries.

5.4.3.8.5.2 Send Work Request Configuration

> See `doca_dpa_dev_verbs_send_wr_set_*` and `doca_dpa_dev_verbs_send_wr_get_*` functions in `doca_dpa_dev_verbs.h` for complete API signatures.

These APIs configure send WRs before posting them to a QP. Configuration must be completed before posting.

Operation Type

`doca_dpa_dev_verbs_send_wr_set/get_opcode()` : Sets/gets the RDMA operation type (`SEND` , `WRITE` , `READ` , `ATOMIC_FETCH_ADD`), determining the WR's fundamental behavior

Memory Settings

- `doca_dpa_dev_verbs_send_wr_set/get_sg_list()` : Sets/gets the SG list pointing to local memory regions containing the data to send
- `doca_dpa_dev_verbs_send_wr_set/get_sg_num_sge()` : Sets/gets the number of SG elements in the list

Control and Synchronization

- `doca_dpa_dev_verbs_send_wr_set/get_send_flags()` : Sets/gets completion generation and solicited event control flags
- `doca_dpa_dev_verbs_send_wr_set/get_fence_mode()` : Sets/gets ordering constraints between WRs

Optional Data

- `doca_dpa_dev_verbs_send_wr_set/get_imm_data()` : Sets/gets immediate data for `SEND_WITH_IMM` and `WRITE_WITH_IMM` operations
- `doca_dpa_dev_verbs_send_wr_set/get_invalidate_rkey()` : Sets/gets remote key to invalidate for `SEND_WITH_INV` operations

RDMA-specific Configuration

Required for RDMA WRITE and READ operations:

- `doca_dpa_dev_verbs_send_wr_set/get_rdma_remote_addr()` : Sets/gets the target memory address on the remote node
- `doca_dpa_dev_verbs_send_wr_set/get_rdma_rkey()` : Sets/gets the remote memory key for accessing the target memory region

Atomic Operations Configuration

Required for atomic operations:

- `doca_dpa_dev_verbs_send_wr_set/get_atomic_remote_addr()` : Sets/gets the target memory address for the atomic operation

- **doca_dpa_dev_verbs_send_wr_set/get_atomic_rkey()** : Sets/gets the remote memory key for the atomic operation
- **doca_dpa_dev_verbs_send_wr_set/get_atomic_compare_add()** : Sets/gets the value to add in fetch-and-add operations
- **doca_dpa_dev_verbs_send_wr_set/get_atomic_swap()** : Sets/gets the swap value for compare-and-swap operations

5.4.3.8.5.3 Receive Work Request Configuration

> See **doca_dpa_dev_verbs_recv_wr_set_*** and **doca_dpa_dev_verbs_recv_wr_get_*** functions in **doca_dpa_dev_verbs.h** for complete API signatures.

These APIs configure and query receive WRs before posting them to a QP or SRQ. Receive WRs prepare buffers to receive incoming data from remote nodes. All configuration must be completed before calling the posting APIs.

Configuration

- **doca_dpa_dev_verbs_recv_wr_set/get_sg_list()** : Sets/gets the SG list pointing to local memory regions where incoming data will be stored
- **doca_dpa_dev_verbs_recv_wr_set/get_sg_num_sge()** : Sets/gets the number of SG elements in the list

The receive WR configuration is simpler than send WRs since receive operations are passive - they only specify where to store incoming data.

5.4.3.8.5.4 Work Request Posting

> See **doca_dpa_dev_verbs_qp_post_*** functions in **doca_dpa_dev_verbs.h** for complete API signatures.

This section provides APIs for posting send and receive WRs to QPs. All posting functions return a WR counter that can be matched with completion events.

Standard Work Request Posting

- **doca_dpa_dev_verbs_qp_post_send_wr()** : Posts a configured send WR to the send queue. Returns the send WR counter for completion tracking.
- **doca_dpa_dev_verbs_qp_post_recv_wr()** : Posts a receive WR to the receive queue. Prepares the queue to receive incoming data. Returns the send WR counter for completion tracking.

Raw WQE Posting

- **doca_dpa_dev_verbs_qp_post_send_raw_wqe()** : Posts a custom-built send Work Queue Element directly to hardware, bypassing high-level WR processing.
- **doca_dpa_dev_verbs_qp_post_recv_raw_wqe()** : Posts a custom-built receive WQE with specified size.

SG List Usage

When using a SG list (**sg_list**) that is not fully occupied, set the last entry's `lkey` field to `DOCA_DPA_DEV_VERBS_SGE_TERMINATING_LKEY` to indicate the end of valid entries.

5.4.3.8.5.5 Commit Operations

> See **doca_dpa_dev_verbs_qp_*commit*** functions in **doca_dpa_dev_verbs.h** for complete API signatures.

This section provides APIs for committing posted WRs to hardware, making them available for processing by the DPA device. WRs must be committed for the hardware to process them.

Standard Commit Operations

- **doca_dpa_dev_verbs_qp_commit_send()** : Commits all pending send WRs to hardware with internal memory fence for ordering guarantees.
- **doca_dpa_dev_verbs_qp_commit_recv()** : Commits all pending receive WRs to hardware with internal memory fence.

Lightweight Commit Operations

> User must perform memory fence operations before calling these functions.

- **doca_dpa_dev_verbs_qp_lw_commit_send()** : Lightweight send commit without internal memory fence. Higher performance but requires manual memory synchronization.
- **doca_dpa_dev_verbs_qp_lw_commit_recv()** : Lightweight receive commit without internal memory fence. User responsible for proper memory ordering.

5.4.3.8.5.6 Shared Receive Queue Operations

Basic SRQ Operations

- **doca_dpa_dev_verbs_srq_post_recv_wr()** : Posts a receive WR to the SRQ, specifying the SRQ type and receive WR structure.
- **doca_dpa_dev_verbs_srq_commit_recv()** : Commits pending receive WRs to the SRQ with internal memory fence.
- **doca_dpa_dev_verbs_srq_lw_commit_recv()** : Lightweight commit for SRQ receive WRs without internal memory fence. User responsible for memory fence operations.

SRQ Raw WQE Operations

doca_dpa_dev_verbs_srq_post_recv_raw_wqe() : Posts a custom-built receive WQE to the SRQ with specified type and size. Follows same principles as QP raw WQE posting.

SG List Usage

When providing a SG list (**sg_list**) that is not fully occupied, the user must set the last entry's lkey field to `DOCA_DPA_DEV_VERBS_SGE_TERMINATING_LKEY` to indicate the end of the valid entries.

SRQ Management

> ⚠ This API is relevant only for Linked-List SRQ.

doca_dpa_dev_verbs_srq_linked_list_ack_wr() : Acknowledges processed receive WR in linked list SRQ implementation, specifying the receive WR count (**doca_dpa_dev_completion_element_get_wqe_counter()** return value) to acknowledge.

5.4.3.8.5.7 Query APIs

> ⚠ The following APIs are for debug and inspection purposes only. Do not modify the returned WQ buffers or DBR addresses, as this can cause undefined behavior and system instability.

Queue Pair Query APIs

- **doca_dpa_dev_verbs_qp_get_wq()** : Retrieves work queue attributes, including SQ/RQ buffer addresses, entry counts, and receive WQE size. The returned buffers are read-only; modifying them can cause undefined behavior.
- `doca_dpa_dev_verbs_qp_get_dbr_addr()`: Returns the doorbell record address for the QP. The returned address is read-only; modifying it can cause undefined behavior.
- **doca_dpa_dev_verbs_qp_get_qpn()** : Gets the QP number.
- **doca_dpa_dev_verbs_qp_get_user_index()** : Retrieves the user-assigned index for the QP.

SRQ Query APIs

- **doca_dpa_dev_verbs_srq_get_srqn()** : Gets the SRQ number.
- `doca_dpa_dev_verbs_srq_get_wq()`: Retrieves SRQ work queue attributes, including buffer address, entry count, and WQE size. The returned buffers are read-only; modifying them can cause undefined behavior.

5.4.3.8.6 Integration with Host-side DOCA RDMA Verbs

The DPA Device Verbs library integrates with host-side DOCA RDMA Verbs to provide a complete RDMA solution.

5.4.3.8.6.1 QP/SRQ Configuration Modes

Two mutually exclusive modes are available for configuring QPs and SRQs with DPA integration.

Mode 1: DPA Context Integration (Basic)

This mode provides full DPA integration, where the DPA context manages all DPA-related resources:

- Set the DPA context using `doca_verbs_qp_init_attr_set_dpa_ctx()` for QPs or `doca_verbs_srq_init_attr_set_dpa_ctx()` for SRQs.
- The DPA context automatically handles memory allocation, doorbell records, and user access regions.

- This is the basic mode for DPA applications due to simplified resource management.

Mode 2: External Resource Management (Advanced)

This mode provides fine-grained control over DPA resources:

- Set external user memory (UMEM) for the work queue buffer using `doca_verbs_qp_init_attr_set_external_umem()` or `doca_verbs_srq_init_attr_set_external_umem()`.
- Set external doorbell record (DBR) using `doca_verbs_qp_init_attr_set_external_dbr_umem()` or `doca_verbs_srq_init_attr_set_external_dbr_umem()`.
- Set external user access region (UAR) using `doca_verbs_qp_init_attr_set_external_uar()`.
- The application is responsible for manual management of all DPA resources.
- This mode requires a deep understanding of DPA resource management and memory allocation.
- It is used for scenarios requiring custom resource allocation strategies and full user control.

Notes:

- These two modes are mutually exclusive; you cannot mix DPA context mode with external resource mode.
- Mode 1 is the basic mode for applications.
- Mode 2 should only be used when custom resource management is required.
- All resources in Mode 2 must be properly aligned and configured according to DPA hardware requirements.

5.4.3.8.6.2 Host-side Setup Requirements

For Mode 1 (DPA Context Integration)

1. Create a DOCA RDMA Verbs context and protection domain.
2. Retrieve the DOCA device from the Verbs PD.
3. Create a DPA context using `doca_dpa_create()`.
4. Configure QP initialization attributes.
5. Set the DPA context for the QP/SRQ.
6. Create and configure DPA completion contexts.
7. Associate completion contexts with QPs.

For Mode 2 (External Resource Management)

1. Create a DOCA RDMA Verbs context and protection domain.
2. Retrieve the DOCA device from the Verbs PD.
3. Allocate and configure external WQ UMEM.
4. Allocate and configure external DBR UMEM.
5. Allocate and configure external UAR.
6. Configure QP initialization attributes.
7. Set external resources for the QP/SRQ.
8. Create and configure DPA completion contexts.

9. Associate completion contexts with QPs.

5.4.3.8.7 Usage Patterns

5.4.3.8.7.1 Basic DPA QP Setup Pattern

```
// Create DOCA RDMA Verbs context and PD
doca_verbs_context_create(devinfo, 0, &verbs_ctx);
doca_verbs_pd_create(verbs_ctx, &pd);


// Retrieve DOCA device from Verbs PD
doca_verbs_pd_as_doca_dev(pd, &dev);


// Create DPA context and completion contexts
doca_dpa_create(dev, &dpa_ctx);
doca_dpa_completion_create(dpa_ctx, 256, &send_completion);
doca_dpa_completion_create(dpa_ctx, 256, &recv_completion);

// Configure QP with DPA completions
doca_verbs_qp_init_attr_create(&qp_init_attr);
doca_verbs_qp_init_attr_set_pd(qp_init_attr, pd);
doca_verbs_qp_init_attr_set_external_datapath_en(qp_init_attr, 1);
doca_verbs_qp_init_attr_set_dpa_ctx(qp_init_attr, dpa_ctx);
doca_verbs_qp_init_attr_set_send_dpa_completion(qp_init_attr, send_completion);
doca_verbs_qp_init_attr_set_receive_dpa_completion(qp_init_attr, recv_completion);

// Create queue pair
doca_verbs_qp_create(verbs_ctx, qp_init_attr, &qp);

// Get DPA handles
doca_dpa_dev_verbs_qp_t dpa_qp;
doca_dpa_dev_completion_t send_comp, recv_comp;

doca_verbs_qp_get_dpa_handle(qp, dpa_ctx, &dpa_qp);
doca_dpa_completion_get_dpa_handle(send_completion, &send_comp);
doca_dpa_completion_get_dpa_handle(recv_completion, &recv_comp);
```

5.4.3.8.7.2 Advanced DPA QP Setup Pattern

```
// Create DOCA RDMA Verbs context and PD
doca_verbs_context_create(devinfo, 0, &verbs_ctx);
doca_verbs_pd_create(verbs_ctx, &pd);

// Retrieve DOCA device from Verbs PD
doca_verbs_pd_as_doca_dev(pd, &dev);

// Create DPA context and completion contexts
doca_dpa_create(dev, &dpa_ctx);
doca_dpa_completion_create(dpa_ctx, 256, &send_completion);
doca_dpa_completion_create(dpa_ctx, 256, &recv_completion);

// Allocate external UMEM for QP using DPA heap addresses
uint64_t qp_wq_dpa_addr = doca_dpa_mem_alloc(dpa_ctx, qp_umem_size);
doca_umem_dpa_create(dpa_ctx, qp_wq_dpa_addr, &qp_umem);

uint64_t dbr_dpa_addr = doca_dpa_mem_alloc(dpa_ctx, qp_dbr_umem_size);
doca_umem_dpa_create(dpa_ctx, dbr_dpa_addr, &qp_dbr_umem);

// Create UAR for doorbell access
doca_uar_dpa_create(dpa_ctx, &dpa_uar);

// Configure QP with external resources
doca_verbs_qp_init_attr_create(&qp_init_attr);
doca_verbs_qp_init_attr_set_pd(qp_init_attr, pd);
doca_verbs_qp_init_attr_set_external_datapath_en(qp_init_attr, 1);
doca_verbs_qp_init_attr_set_external_umem(qp_init_attr, qp_umem, 0);
doca_verbs_qp_init_attr_set_external_dbr_umem(qp_init_attr, qp_dbr_umem, 0);
doca_verbs_qp_init_attr_set_external_uar(qp_init_attr, dpa_uar);
doca_verbs_qp_init_attr_set_send_dpa_completion(qp_init_attr, send_completion);
doca_verbs_qp_init_attr_set_receive_dpa_completion(qp_init_attr, recv_completion);

// Create queue pair
doca_verbs_qp_create(verbs_ctx, qp_init_attr, &qp);

// Get DPA handles
doca_dpa_dev_verbs_qp_t dpa_qp;
doca_dpa_dev_completion_t send_comp, recv_comp;


doca_dpa_dev_verbs_qp_t dpa_qp;
doca_verbs_qp_get_dpa_handle(qp, dpa_ctx, &dpa_qp);
```

5.4.3.8.7.3 SRQ Setup Pattern

```
// Create SRQ with DPA context
doca_verbs_srq_init_attr_create(&srq_init_attr);
doca_verbs_srq_init_attr_set_pd(srq_init_attr, pd);
doca_verbs_srq_init_attr_set_dpa(srq_init_attr, dpa_ctx);
doca_verbs_srq_create(verbs_ctx, srq_init_attr, &srq);

// Get SRQ handle and post receives
doca_dpa_dev_verbs_srq_t dpa_srq;
doca_verbs_srq_get_dpa_handle(srq, dpa_ctx, &dpa_srq);
```

5.4.3.8.7.4 RDMA Write Operation (Data path example)

```
__dpa_global__ void dpa_rdma_write_example(doca_dpa_dev_verbs_qp_t qp_handle,
                                           uint64_t local_addr,
                                           uint32_t length,
                                           uint32_t lkey,
                                           uint64_t remote_addr,
                                           uint32_t rkey)
{
    struct doca_dpa_dev_verbs_send_wr send_wr;
    struct doca_dpa_dev_verbs_sge sge;

    // Configure scatter-gather element
    sge.addr = local_addr;
    sge.length = length;
    sge.lkey = lkey;

    // Configure RDMA write work request
    doca_dpa_dev_verbs_send_wr_set_opcode(&send_wr, DOCA_DPA_DEV_VERBS_SEND_WR_OPCODE_WRITE);
    doca_dpa_dev_verbs_send_wr_set_sg_list(&send_wr, &sge);
    doca_dpa_dev_verbs_send_wr_set_sg_num_sge(&send_wr, 1);
    doca_dpa_dev_verbs_send_wr_set_rdma_remote_addr(&send_wr, remote_addr);
    doca_dpa_dev_verbs_send_wr_set_rdma_rkey(&send_wr, rkey);
    doca_dpa_dev_verbs_send_wr_set_fence_mode(&send_wr, DOCA_DPA_DEV_VERBS_SEND_WR_FM_NO_FENCE);
    doca_dpa_dev_verbs_send_wr_set_send_flags(&send_wr, DOCA_DPA_DEV_VERBS_SEND_WR_FLAGS_SIGNALED);

    // Post and commit
    uint32_t wr_count = doca_dpa_dev_verbs_qp_post_send_wr(qp_handle, &send_wr);
    doca_dpa_dev_verbs_qp_commit_send(qp_handle);
}
```

5.4.3.8.8 Optimizing Performance with Compile-Time Constants

For better performance:

- Use compile-time constant values for the `opcode` parameter in `doca_dpa_dev_verbs_send_wr_set_opcode()`. This allows the compiler to optimize the code path.
- Use compile-time constant values for the `srq_type` parameter in `doca_dpa_dev_verbs_srq_post_recv_wr()`. This enables the compiler to generate more efficient code paths for SRQ operations.

Example: Use enum values like `DOCA_DPA_DEV_VERBS_SEND_WR_OPCODE_WRITE`, `DOCA_DPA_DEV_VERBS_SRQ_TYPE_LINKED_LIST`, or `DOCA_DPA_DEV_VERBS_SRQ_TYPE_CONTIGUOUS` as literal constants instead of passing them through variables.

5.4.4 DOCA PCC

This guide provides an overview and configuration instructions for the DOCA Programmable Congestion Control (PCC) API.

5.4.4.1 1. Introduction

> The quality status of DOCA libraries is listed here.

The DOCA PCC library offers a high-level programming interface that enables users to implement customized congestion control (CC) algorithms. By leveraging the NVIDIA® BlueField®-3 platform hardware acceleration, it facilitates efficient network congestion management while abstracting away hardware complexities.

With the DOCA PCC API, users can:

- Configure probe packets for sending and receiving
- Retrieve CC events or packets and access their fields
- Set flow rate limits to regulate network traffic
- Maintain per-flow contexts for individualized management
- Initiate and configure CC algorithms tailored to application needs
- Process incoming request packets and generate appropriate response packets

This streamlined API allows developers to focus on designing and implementing congestion control logic without worrying about low-level hardware operations.

5.4.4.2 2. Prerequisites

DOCA PCC-based applications can run on either the host machine or the NVIDIA BlueField-3 Platform (or later) target.

> Currently, DOCA PCC is supported only for the ETHERNET link type.

5.4.4.2.1 2.1.1. Enabling DOCA PCC

To enable DOCA PCC RP:

1. On the host/VM, run:

```
mlxconfig -d <mlx_device> -y s USER_PROGRAMMABLE_CC=1
```

2. Perform a graceful shutdown and then power cycle the host.

To enable DOCA PCC NP:

1. On the host/VM, run:

```
mlxconfig -d <mlx_device> -y s PCC_INT_EN=0
```

2. Perform a graceful shutdown and then power cycle the host.

5.4.4.2.2 2.1.2. Configuration Notes

- Setting `PCC_INT_EN=1` blocks the creation of DOCA PCC NP context and enables the legacy NP solution. It also only supports DOCA PCC RP context for setting Congestion Control Message After Drop (CCMAD) probe packet format.
- For IFA2.0 support, enable both DOCA PCC RP and DOCA PCC NP on all cluster nodes.
- The DOCA PCC NP process requires root access.
- For BlueField-3 devices in DPU mode, executing the DOCA PCC NP process on an x86 host is not supported.
- When running from an x86 host in NIC mode, privileged permissions are required. Check the privilege level using `mlxprivhost -d <mlx_device> q`.
- To enable the injection of response timestamp information into probe response payloads, you must set `FLEX_PARSER_PROFILE_ENABLE=10` in the device's non-volatile (NV) configuration.

```
mlxconfig -d <mlx_device> -y s FLEX_PARSER_PROFILE_ENABLE=10
```

5.4.4.2.3 2.1.3. DPACC Tool

The DPACC tool compiles and links user algorithms and device code with the DOCA PCC device library to create loadable applications. DPACC is included in the DOCA SDK installation package. For more information, refer to *DOCA DPACC Compiler* documentation.

5.4.4.3 3. Changes From Previous Releases

5.4.4.3.1 3.1. Changes in 3.3.0

5.4.4.3.1.1 3.1.1. Added

- Support for injecting response timestamp in DOCA PCC NP.

5.4.4.4 4. Dependencies

The library requires firmware version 32.38.1000 and higher.

5.4.4.5 5. Architecture

DOCA PCC comprises three main components which are part of the DOCA SDK installation package.

5.4.4.5.1 5.1. Host Library

The host library offers a unified interface for managing the DOCA PCC context configuration.

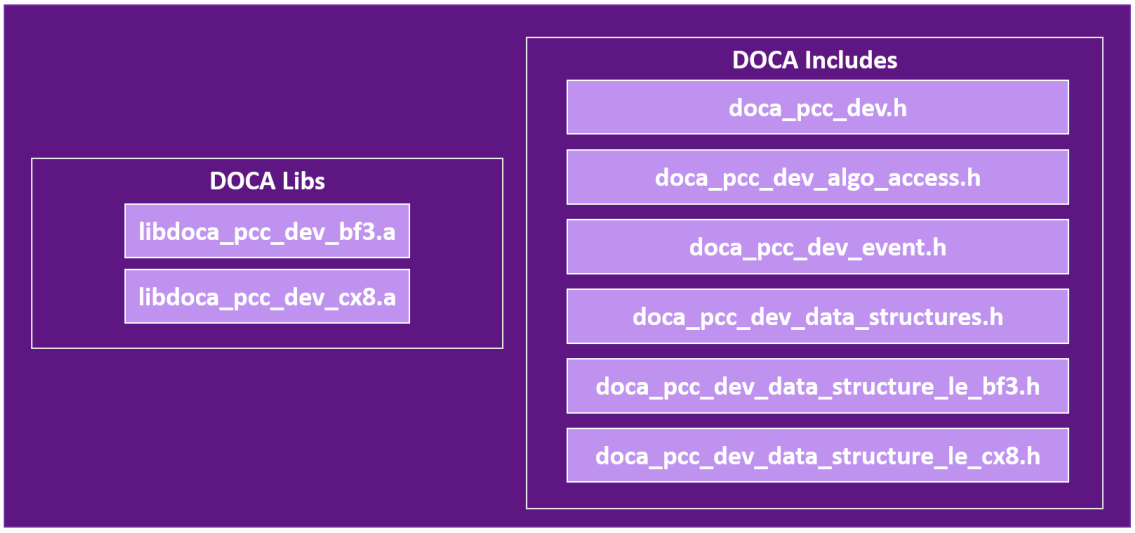
As part of the control path, the host library integrates passively within the application, orchestrating congestion control activities without directly handling data transmission.

Host/device library and header files:

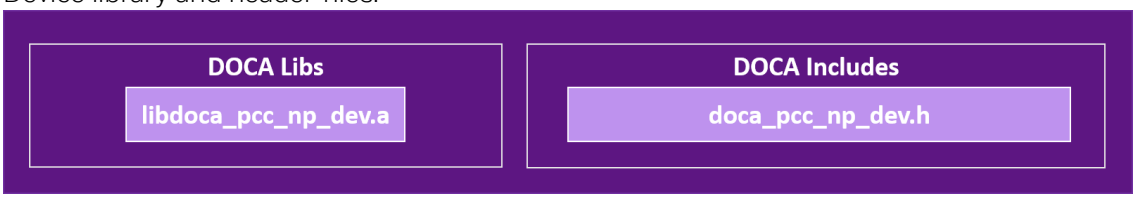
5.4.4.5.2 5.2. Device Libraries

The DOCA PCC context assumes one of two roles:

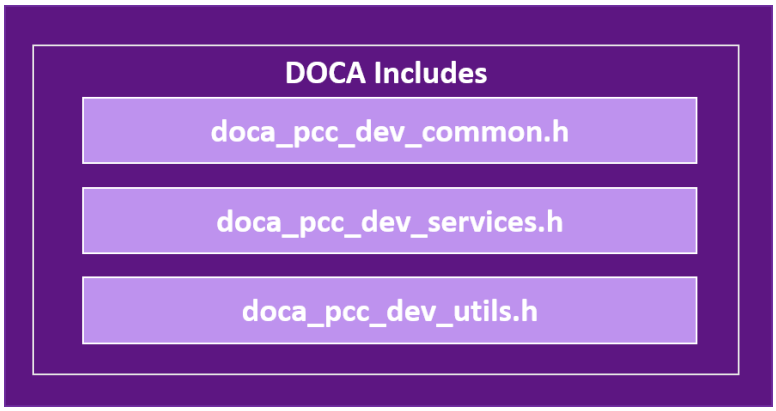
- Reaction point (RP): Monitors network conditions actively, dynamically adjusting data transmission rates to alleviate congestion promptly. RP context is global per NIC.
  Device library and header files:



- Notification point (NP): Passively receives congestion notifications from external sources, processing them intelligently to facilitate informed decisions within the application. NP context is global per e-switch owner.
  Device library and header files:



Both RP and NP device libraries share common headers:



Currently, the device library and the user algorithm are implemented and managed over the BlueField's data-path accelerator (DPA) subsystem.

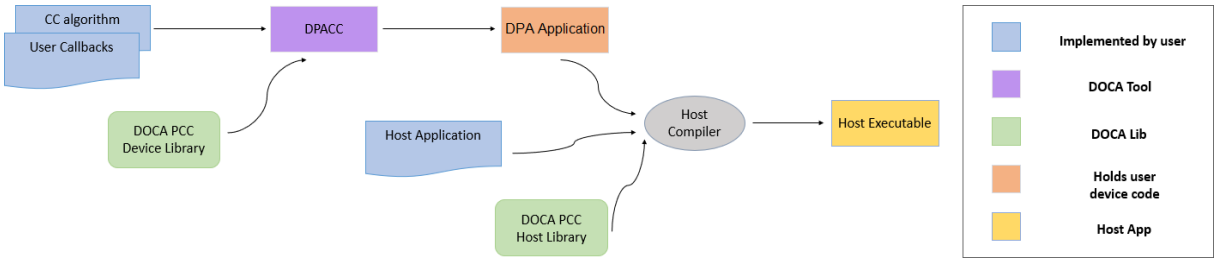
For more info on DPA, refer to *DPA Subsystem*.

5.4.4.5.3 5.3. Development Flow

DOCA enables developers to program the congestion control algorithm into the system using the DOCA PCC library.

The following are the required steps to start programming:

1. Implement CC algorithms and probe packet handling using the API provided by the device header files.
2. Implement the user callbacks defined by the library for DataPath:
    - For RP: **doca_pcc_dev_user_init()**, **doca_pcc_dev_user_set_algo_params()**, **doca_pcc_dev_user_algo()**.
    - For NP: **doca_pcc_dev_np_user_packet_handler()**
3. Use DPACC to build a DPA application (i.e., a host library which contains an embedded device executable). Input for DPACC are the files containing the implementation of the previous steps.
4. Build host executable using a host compiler. Inputs for the host compiler are the DPA application generated in the previous step and the user application host source files.
5. In the host executable, create and start a DOCA PCC context which is set with the DPA application containing the device code.

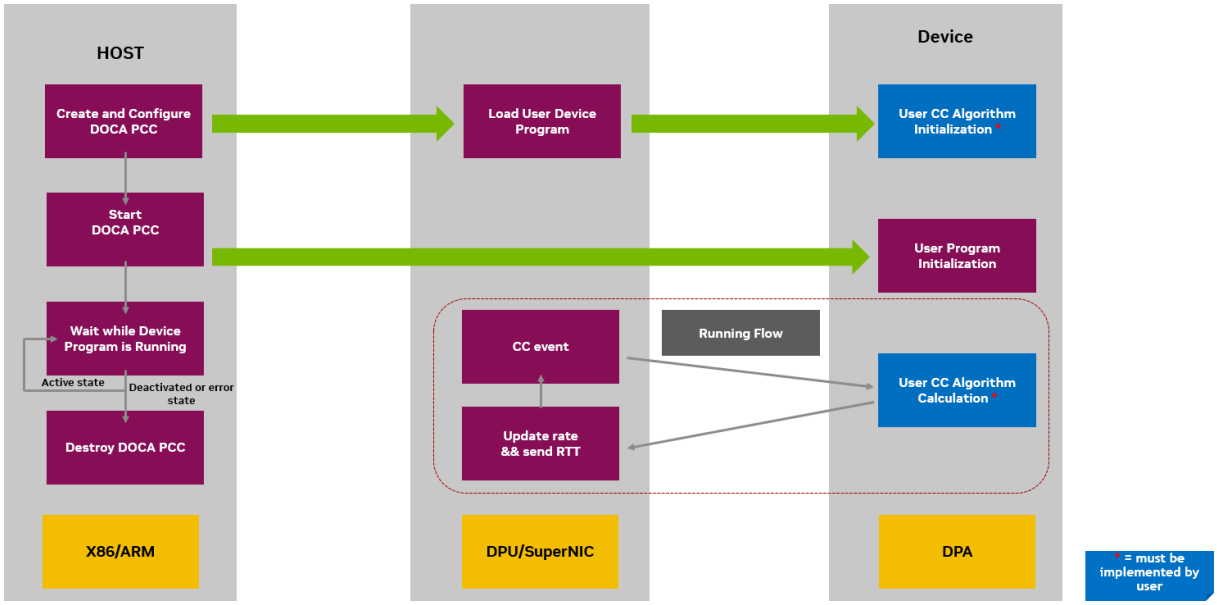


For a more descriptive example, refer to *NVIDIA DOCA PCC Application Guide*.

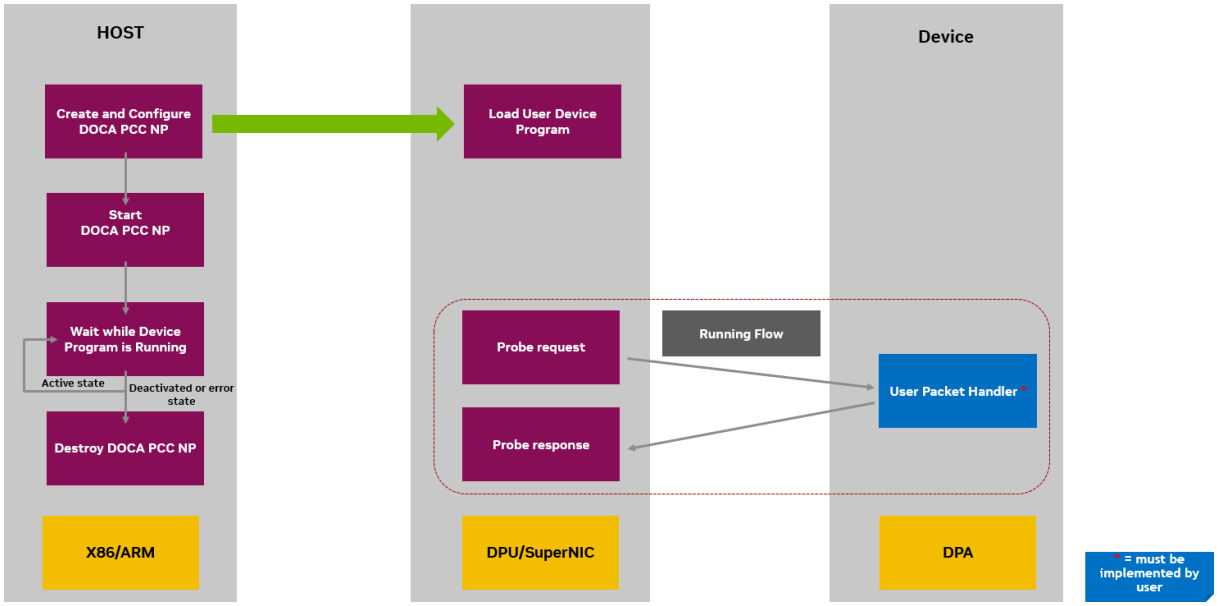
> **Initialization Sequence**
>
> The PCC program must be loaded before the QP is created. If the QP is created first, it will not be able to access the PCC program context. This dependency is strict; reversing the order causes PCC features to fail or result in undefined library behavior.

5.4.4.5.4 5.4. System Design

DOCA PCC flow for implementing an RP program:



DOCA PCC flow for implementing an NP program:



5.4.4.6 6. API

For the library API reference, refer to PCC API documentation in the *API References*.

The following sections provide additional details about the library API.

5.4.4.6.1 6.1. Host API

The host library API consists of calls to set the PCC context attributes and observe availability of the process.

5.4.4.6.1.1 6.1.1. Selecting and Opening DOCA Device

To perform PCC operations, a device must be selected. To select a device, users may iterate over all DOCA devices using **doca_devinfo_list_create()** and check whether the device supports the desired PCC role either via **doca_devinfo_get_is_pcc_supported()** for RP, or **doca_pcc_np_cap_is_supported()** for NP.

5.4.4.6.1.2 6.1.2. Setting Up and Starting DOCA PCC Context

After selecting a DOCA device, a PCC context can be created.

As described in the Architecture section, The DOCA PCC library provides APIs to leverage Reaction Points (RP) and Notification Points (NP) to implement programmable congestion control strategies.

Call **doca_pcc_create()** to create a DOCA PCC RP context, and **doca_pcc_np_create()** to create a DOCA PCC NP context.

Afterwards, the following attributes must be set for the PCC context:

- Context app – the name of the DPA application compiled using DPACC, consisting of the device algorithm and code. This is set using the call **doca_pcc_set_app()**.
- Context threads – the affinity of DPA threads to be used to handle CC events. This is set using the call **doca_pcc_set_thread_affinity()**. The number of threads to be used must be constrained between the minimum and maximum number of threads allowed to run the PCC process (see **doca_pcc_get_min_num_threads()** and **doca_pcc_get_max_num_threads()**). The availability and usage of the threads for PCC is dependent on the complexity of the CC algorithm, link rate, and other potential DPA users.

  > ⚠ Users can manage DPA threads in the system using EU pre-configuration with the **dpaeumgmt** tool. For more information, refer to Single Point of Resource Distribution.

After setting up the context attributes, the context can be started using **doca_pcc_start()**. Starting the context initiates the CC algorithm supplied by the user.

5.4.4.6.1.3 6.1.3. Configuring Probe Packets

The DOCA PCC library provides APIs to configure the probe packet settings to tailor congestion control behaviors according to specific network conditions.

The probe packet serves to probe the network for congestion and gather essential feedback for congestion control algorithms.

The DOCA PCC Library supports the following probe packet types:

- CCMAD – Provides information about the network's round-trip time so the algorithm can detect and adapt to congestion proactively
- IFA1 – In-band Flow Analyzer 1 packets provide in-band congestion feedback for proactive congestion control
- IFA2 – In-band Flow Analyzer 2 packets offer an alternative method for in-band congestion feedback, optimized for specific network environments

5.4.4.6.1.4 6.1.4. Configuring Dedicated Fields for Different Probe Types

The DOCA PCC library provides APIs to configure specific fields in different supported probe packet types.

- IFA1 – support to configure probe marker
- IFA2 – support to configure GNS and hop limit

5.4.4.6.1.5 6.1.5. Configuring Remote NP Handler

To enable Reaction Point contexts to interact with remote Notification Point contexts, the DOCA PCC library provides an API to set the expected remote handler type.

When the DOCA PCC RP process expects CCMAD probe packet responses from a DOCA PCC NP process, it should set it as so using the API **doca_pcc_rp_set_ccmad_remote_sw_handler()**. If not set, the DOCA PCC RP process expects that no remote DOCA PCC NP process is activated, and that responses are handled by the remote node's hardware. Note that if using probe types other than CCMAD, probe packet responses are always expected to be generated from a remote DOCA Notification Point process.

5.4.4.6.1.6 6.1.6. RTC Timestamps for PCC

To use the real-time clock (RTC) as the timetable for both DOCA PCC RP (requesting party) and DOCA PCC NP (notifying party), the clocks on both endpoints must be synchronized (e.g., using PTP). Synchronization details are outside the scope of this documentation.

By default, RTT probe packet timestamps are taken from the device's free-running clock. The steps below describe how to configure and use RTC-based timestamps.

6.1.6.1. Configuring RTC in DOCA PCC RP

1. Enable RTC on the device.

```
mlxconfig -d <mlx_device> -y s REAL_TIME_CLOCK_ENABLE=1
```

2. Configure RTT timestamp format to use RTC (value 2).

```
mlxconfig -d <mlx_device> -y s ROCE_CC_RTT_TIMESTAMP_FORMAT=2
```

> ⚠ This NV config setting requires MFT version 4.34 or newer.

3. retrieve the RTT request timestamp (T1) using the API **`doca_pcc_dev_get_rtt_req_recv_timestamp();`**. This function is compatible with both RTC and free-running clock sources.

6.1.6.2. Configure RTC in DOCA PCC NP

1. Enable RTC on the device.

```
mlxconfig -d <mlx_device> -y s REAL_TIME_CLOCK_ENABLE=1
```

2. In the host-side code, configure the NP to use the RTC timestamp source using the following APIs:
   - **`doca_pcc_np_cap_is_ts_source_supported()`**: Verifies if the device supports the specific timestamp source (defined in **`doca_pcc_np_ts_source_t`**).
   - **`doca_pcc_np_set_ts_source()`**: Sets the NP to use the specified timestamp source.

     > ⚠ Configuring the timestamp source in the DOCA PCC NP requires **`flexio-sdk`** version 25.10.xxxx or newer.

3. In the device code, retrieve the RTT request receive timestamp (T2) using **`doca_pcc_np_dev_get_t2_ns()`**.

   > ⚠ This API returns a 30-bit value for nanoseconds in little-endian format. It is compatible with both RTC and free-running clock sources.

5.4.4.6.1.7 6.1.7. DOCA PCC Notification Point: Response Timestamp

Response timestamp allows the DOCA PCC Notification Point (NP) to inject timestamp information directly into probe response payloads. This facilitates precise timing measurements in congestion control scenarios by capturing a timestamp at the moment the probe response is transmitted from the port.

When enabled, the timestamp is automatically inserted into the least-significant bits of the last DWORD of the Congestion Control (CC) probe response payload. This allows the Reaction Point (RP) to calculate accurate round-trip times (RTT) and make informed congestion control decisions.

6.1.7.1. Configuring Response Timestamp

To enable the injection of response timestamps, you must set the Flex Parser profile to 10 in the device's non-volatile (NV) configuration.

```
mlxconfig -d <mlx_device> -y s FLEX_PARSER_PROFILE_ENABLE=10
```

6.1.7.2. Response Timestamp API Reference

- **doca_pcc_np_cap_is_resp_ts_supported()** – Queries whether the device supports injecting response timestamps into probe response payloads for Notification Point operations.
- **doca_pcc_np_set_resp_ts_size()** – Configures the number of bits (**ts_size**) used for the timestamp within the last DWORD of the payload.
  - **0** : Response timestamp injection is disabled (Default).
  - **1-32** : The timestamp is injected into the least-significant **ts_size** bits of the last DWORD.

    > ⚠ Larger sizes provide higher precision but consume more payload space. Use **doca_pcc_np_set_resp_ts_resolution()** to adjust the time scale if the value range is insufficient.

- **doca_pcc_np_set_resp_ts_resolution()** – Sets the granularity of the timestamp by configuring a right-bit shift applied to the raw timestamp before injection. This allows trading precision for an extended time range.
  - **0** : No shift applied (maximum precision) (Default).
  - **N** : The raw timestamp is right-shifted by **N** bits (effectively dividing by 2^N).
- **doca_pcc_np_set_ts_source()** – Selects the hardware clock source. This setting applies globally to NP operations, affecting both the response timestamp injection and the **doca_pcc_np_dev_get_t2_ns()** device API.

> ⛔ Deprecation of Default Timestamp Source
>
> The **DOCA_PCC_NP_TS_SOURCE_DEFAULT** enum value is deprecated in this release and will be removed in an upcoming version. Applications must explicitly select a valid timestamp source (e.g., **FREE_RUNNING** or **REAL_TIME**).

```
typedef enum {
    DOCA_PCC_NP_TS_SOURCE_FREE_RUNNING = 0,                    /**< Free running timestamp. */
    DOCA_PCC_NP_TS_SOURCE_DEFAULT DOCA_DEPRECATED_ENUM = 0x1, /**< @deprecated Deprecated in this
release; will be removed in an upcoming release. */
    DOCA_PCC_NP_TS_SOURCE_REAL_TIME = 0x2,                     /**< Real time timestamp. */
} doca_pcc_np_ts_source_t;
```

5.4.4.6.1.8 6.1.8. Debuggability

The DOCA PCC library provides a comprehensive set of debugging APIs. These tools allow you to diagnose issues, pinpoint bottlenecks, and access real-time information from your running device-side application.

6.1.8.1. PCC Tracer

The library's tracer is optimized for high-frequency use. It allows you to observe device algorithm behavior and diagnose issues with negligible impact on application performance.

PCC tracing is enabled by default when **doca_pcc_start()** is called. By default, trace output is routed to **stdout**.

Tracer API reference:

Category	API Function	Description
State control	**doca_pcc_deactivate_tracer()**	Disables runtime trace printing.
	doca_pcc_activate_tracer()	Re-enables tracing dynamically without restarting the PCC context.
Destination config	**doca_pcc_trace_buf_set()**	Routes trace output to a user-supplied buffer.
	doca_pcc_trace_file_set()	Routes trace output to a specified file.
	doca_pcc_trace_buf_get()	Queries the currently configured destination buffer.
	doca_pcc_trace_file_get()	Queries the currently configured destination file. Returns **DOCA_ERROR_BAD_CONFIG** if no file is set.
Formatting and handling	**doca_pcc_set_trace_message()**	Sets the specific trace message string used for device printing.
	doca_pcc_register_trace_handler()	Registers a custom callback function for programmatic handling. The callback receives a user context pointer and an array of arriving device trace reports.

We don't have a way to export this macro.

6.1.8.2. PCC Logger

The logger handles explicit messages generated by the device API **doca_pcc_dev_printf()**.

Performance

Unlike the tracer, device-side prints incur measurable performance overhead. Use them sparingly for short-term debugging or targeted diagnostics. For ongoing observability, use the PCC Tracer.

Logger API reference:

Category	API Function	Description
Destination config	`doca_pcc_set_print_buffer_size()`	Configures the size of the buffer used to accumulate device-side print data (`doca_pcc_dev_printf()`) before sending it to the host. Incurs measurable performance overhead; use sparingly.
	`doca_pcc_log_file_set()`	Routes log output to a specified file.
	`doca_pcc_log_file_get()`	Queries the currently configured destination file. Returns `DOCA_ERROR_BAD_CONFIG` if no file is set.

We don't have a way to export this macro.

6.1.8.3. Device Coredump File

The coredump utility captures crucial device crash data when an unrecoverable error occurs on the device side of the application. The resulting file includes a memory snapshot at the exact time of the crash, detailing the program's state, variable values, and the call stack.

Coredump API reference:

- `doca_pcc_set_dev_coredump_file()` – Configures the host-side file path to capture device crash data (memory snapshot, call stack, variables) when an unrecoverable error occurs on the device side.

5.4.4.6.1.9 6.1.9. Device Mailbox

The DOCA PCC library provides a set of APIs for sending and receiving messages through a mailbox. This service allows communication between the host and device:

- `doca_pcc_set_mailbox()` – API to set the mailbox attributes for the process.
- `doca_pcc_mailbox_get_request_buffer()` and `doca_pcc_mailbox_get_response_buffer()` – API to get the buffers with which the communication will be handled. User can set the request he wants to send to the device, and get a response back.
- `doca_pcc_mailbox_send()` – API to send the mailbox request to the device. This is a blocking call which invokes a callback on the device `doca_pcc_dev_user_mailbox_handle()` which user can handle.

5.4.4.6.1.10 6.1.10. High Availability

The DOCA PCC library supports High Availability (HA) to ensure continuous operation and recovery if the running PCC process malfunctions. You can achieve this by running multiple PCC processes in parallel.

6.1.10.1. Process Lifecycle and Failover

1. Call `doca_pcc_start()` to register multiple PCC processes in parallel with the NIC firmware.

2. The firmware designates the first process to register as the ACTIVE process (running on the DPA and handling Congestion Control events). All subsequent processes are automatically placed in STANDBY mode.
3. Continuously observe the status of your processes using `doca_pcc_get_process_state()`. If a state change occurs, the `doca_pcc_wait()` function will return.
4. If the currently ACTIVE process encounters an error or stops processing events, the firmware automatically promotes one of the STANDBY processes to become the new ACTIVE process.
5. The defunct (failed) process must explicitly call `doca_pcc_destroy()` to safely free its allocated resources.

> ⚠ Configuration state is not replicated across processes. When a failover occurs, the replacement process does not automatically receive new algorithm configurations, and any user-applied PPCC commands are lost and must be re-applied manually.

6.1.10.2. Process States (`doca_pcc_process_state_t`)

The following table details the possible states of a PCC process at any given time:

State Enum	Value	Description	Action Required
`DOCA_PCC_PS_ACTIVE`	`0`	The process is actively handling CC events.	None (Only one process is active at a time).
`DOCA_PCC_PS_STANDBY`	`1`	The process is waiting in standby mode.	None (Another process is currently ACTIVE).
`DOCA_PCC_PS_DEACTIVATED`	`2`	The process was deactivated by the NIC Firmware.	Must call `doca_pcc_destroy()`.
`DOCA_PCC_PS_ERROR`	`3`	The process has encountered an error.	Must call `doca_pcc_destroy()`.

5.4.4.6.1.11 6.1.11. PCC Resources

The PCC Resources API provides a mechanism to parse and query pre-defined PCC resource configurations. This enables applications to discover and utilize the Execution Units (EUs) allocated to them in the Single Point of Resource Distribution (SPRD) file.

The SPRD file is a YAML-formatted configuration file that defines PCC application resources. It specifies:

- Application names (keys).
- The number of allocated EUs per application.
- The specific EU IDs assigned to each application.

For file syntax and examples, refer to the Single Point of Resource Distribution documentation.

Key functions:

Function	Description
`doca_pcc_resources_create`	Creates a PCC resources object by parsing an SPRD buffer and extracting the configuration for a specific application
`doca_pcc_resources_get_num_eus`	Releases all resources associated with a PCC resources object
`doca_pcc_resources_get_eus`	Retrieves the number of EUs allocated to the application in the PCC resources object
`doca_pcc_resources_destroy`	Retrieves the array of EU IDs allocated to the application

5.4.4.6.2 6.2. Device API

The device library API provides the necessary calls to set up and manage your congestion control (CC) algorithms so they can handle CC events arriving directly on the hardware.

5.4.4.6.2.1 6.2.1. Counter groups

Counter groups are port-level arrays of 32-bit counters, each owned by either an algorithm slot or a histogram. The library natively supports between 1 and 15 groups per port (defaulting to 2).

Function	Description
`doca_pcc_dev_declare_num_counter_groups()`	Declares the number of counter groups per port. *(Note: Must be called within* `doca_pcc_dev_user_declare_memory()` *).*
`doca_pcc_dev_set_slot_counter_group()`	Binds a specific counter group to an algorithm slot.
`doca_pcc_dev_algo_init_counter()`	Initializes a counter's metadata, including its maximum value, permissions, and description.
`doca_pcc_dev_get_counters()`	Retrieves a pointer to the counter array for a specific algorithm slot.
`doca_pcc_dev_get_counters_num()`	Retrieves the total number of counters assigned to an algorithm slot.

5.4.4.6.2.2 6.2.2. Counter Sampling

These APIs allow you to sample NIC byte counters to monitor the amount of data transmitted and received through the NIC.

Best Practice

It is highly recommended to configure the counters inside the `doca_pcc_dev_user_port_info_changed()` callback, as this indicates the correct port state to sample from.

API Function	Description
`doca_pcc_dev_nic_counters_config()`	Prepares the list of counters you want to read.
`doca_pcc_dev_nic_counters_sample()`	Samples and retrieves the new counter values.

5.4.4.6.2.3 6.2.3. Algorithm Access

The Reaction Point (RP) device library supports running multiple PCC algorithms, useful for fast A/B testing and comparative runs. You can utilize the default library algorithm alongside your own custom algorithms.

Category	API Function	Description
Core	`doca_pcc_dev_default_internal_algo()`	Loads the default CC algorithm (can be used fully or partially).
	`doca_pcc_dev_init_algo_slot()`	Assigns a specific algorithm to run on a designated device port (slot).
Initialization	`doca_pcc_dev_algo_init_param()`	Initiates the algorithm's parameters.
	`doca_pcc_dev_algo_init_counter()`	Initiates the algorithm's counters.
	`doca_pcc_dev_algo_init_metadata()`	Initiates the algorithm's metadata base.

5.4.4.6.2.4 6.2.4. Dynamic Memory on DPA

By default, the DOCA PCC library allocates DPA global memory at the maximum possible size for all resources—assigning all 8 algorithm slots on every port, each equipped with the maximum number of parameters and counters.

In memory-constrained DPA environments, this default behavior is wasteful. To optimize resource usage, developers can use the `doca_pcc_dev_user_declare_memory()` callback to declare exactly what memory is required. The library utilizes these declarations to compute a right-sized memory allocation.

This callback is executed exactly once during process startup, strictly before `doca_pcc_dev_user_init()`.

5.4.4.6.2.5 6.2.5. Memory Declaration APIs

The functions below must be called exclusively inside `doca_pcc_dev_user_declare_memory()`.

Function	Declaration Purpose	Default Behavior (If Omitted)
`doca_pcc_dev_declare_slot_algo()`	Maps an algorithm index to a specific port slot. Slots left unassigned (`DOCA_PCC_DEV_SLOT_UNASSIGNED`) contribute zero to the memory budget and consume no parameter/counter memory.	All slots are assigned on all ports (maximum memory consumption).
`doca_pcc_dev_declare_algo()`	Sets the exact number of parameters and counters required for an algorithm.	44 parameters and 63 counters per algorithm.
`doca_pcc_dev_declare_num_counter_groups()`	Sets the number of counter groups per port (Range: 1–15).	2 counter groups.
`doca_pcc_dev_declare_num_of_histograms()`	Sets the number of histograms per port (Range: 0–15).	0 (Histograms disabled).
`doca_pcc_dev_declare_threads_per_counter_set()`	Sets the per-thread counter sets, effectively multiplying the counter pool.	1 shared set per group.

5.4.4.6.2.6 6.2.6. Algorithm Selection

Algorithms are enabled or disabled on specific "algo slots" using either `doca_pcc_dev_init_algo_slot()` or the `mlxreg` command (`cmd_type` 1 and 2).

The algorithm that is ultimately selected for traffic depends on the negotiation between the two connection endpoints:

- Successful negotiation (with ECE): Occurs if both endpoints support Enhanced Congestion Control (ECE). Each Queue Pair (QP) specifies its CC algo slot via ECE. If multiple algorithms are enabled, the one with the lowest shared slot index is selected.
- No negotiation (without ECE): If ECE is not supported or not enabled, no negotiation occurs. The system defaults to the default algorithm slot.
    - For example, When testing with `ib_write_bw` , algorithm negotiation only executes if you pass the `--rdma_cm` parameter on both the client and server (which enables ECE).

5.4.4.6.2.7 6.2.7. Events

The RP device library provides optimized helper functions to access CC events. These supply the runtime data needed to analyze and inspect hardware events and build out your CC algorithm logic.

5.4.4.6.2.8 6.2.8. Utilities

A set of optimized utility macros (such as fixed-point math operations, memory space fences, etc.) are included to streamline the programming of your CC algorithm on the device.

5.4.4.6.2.9 6.2.9. User Callbacks

The library relies on specific user-implemented callbacks to initiate the CC algorithm and handle packet I/O.

> ⚠ These callbacks must be implemented in your code and compiled by DPACC to be properly provided to the DOCA PCC context in your DPA application.

Reaction point (RP) callbacks:

Callback Function	Trigger/Purpose
`doca_pcc_dev_user_init()`	Called on PCC process load. Use this to initialize the data for all user algorithms.
`doca_pcc_dev_user_algo()`	The primary entry point to your custom user algorithm handling code.
`doca_pcc_dev_user_set_algo_params()`	Called whenever an algorithm parameter change is triggered externally.

Notification point (NP) callbacks:

Callback Function	Trigger/Purpose
`doca_pcc_dev_np_user_packet_handler()`	Called immediately upon the arrival of probe packets.

5.4.4.6.2.10 6.2.10. Debuggability

6.2.10.1. PCC Tracer

The device-side tracer is the preferred method for ongoing observability, designed to minimize performance impact compared to standard prints.

> ⚠ The message format must be pre-configured on the host using `doca_pcc_set_trace_message()` .

API Function	Description
`doca_pcc_dev_trace_5()`	Emits formatted trace records containing up to five arguments. **`Reading PCC trace output`** Each line is a single trace record. Every record begins with the same three fields, separated by slashes: **timestamp (microseconds)**, **sequence number** (ordering within the stream), and **thread identifier** (which logical thread produced the event). After that prefix, the rest of the line carries event-specific values for that trace point.

API Function	Description
`doca_pcc_dev_trace_flush()`	Forces a partially filled trace buffer to the host. Avoid frequent use; typically reserved for the end of a run.

6.2.10.2. PCC Logger

Intended strictly for short-term debugging convenience.

API Function	Description
`doca_pcc_dev_printf()`	Prints device-side messages directly to the host `stdout`. Frequent use degrades performance, and dropped messages may occur due to limited host buffering.

5.4.4.7 7. Histograms

The DOCA PCC library includes a comprehensive histogram API, allowing user algorithms to collect and expose distribution data at runtime (e.g., RTT latency distributions, rate distributions, or application-specific metrics). Histograms are a device-side feature managed through the Reaction Point (RP) device API.

Histogram characteristics:

- Each histogram owns a dedicated counter group for storing bin values.
- Each histogram is identified by a unique ID.
- Histograms support configurable binning modes and dynamic edge values.
- Histograms can be queried and reconfigured remotely via PPCC commands without requiring application recompilation.

5.4.4.7.1 7.1. Histogram Concepts

A histogram partitions a range of values into bins defined by an array of edges. When the user algorithm records a value via `doca_pcc_dev_histogram_update()`, the library identifies the matching bin and increments the corresponding counter.

Because each histogram requires its own dedicated counter group, the total number of histograms per port is strictly bounded by the number of available counter groups (up to 15).

5.4.4.7.1.1 7.1.1. Binning Modes

The histogram mode dictates how the edge values are interpreted. This is specified at initialization using the `doca_pcc_dev_histogram_mode_t` enum:

Mode	Enum Value	Description
Linear	`DOCA_PCC_DEV_HISTOGRAM_MODE_LINEAR`	Creates equally spaced bins. The edges array defines uniform intervals.
Exponential	`DOCA_PCC_DEV_HISTOGRAM_MODE_EXPONENTIAL`	Creates exponentially growing bin widths. This is ideal for distributions spanning several orders of magnitude (e.g., latency).
Free	`DOCA_PCC_DEV_HISTOGRAM_MODE_FREE`	Allows arbitrary, user-defined edge values, providing full control over bin boundaries.

5.4.4.7.1.2 7.1.2. Hardware Limits

Constant	Value	Description
`DOCA_PCC_DEV_HISTOGRAM_MAX_BINS`	32	The maximum number of bins permitted per histogram.
`DOCA_PCC_DEV_MAX_NUM_HISTOGRAMS`	`DOCA_PCC_DEV_MAX_NUM_COUNTER_GROUP`	The maximum histograms per port (which equals the max counter groups).
`DOCA_PCC_DEV_DEFAULT_NUM_HISTOGRAMS`	0	The default number of histograms (disabled by default unless explicitly declared).

5.4.4.7.2 7.2. Histogram Lifecycle

1. Declare: Optionally call `doca_pcc_dev_declare_num_of_histograms()` to allocate the required number of histograms.
2. Initialize: Within the `doca_pcc_dev_user_init()` callback, call `doca_pcc_dev_histogram_init()` for each histogram. This assigns the histogram to a counter group and sets its edges, mode, and description.
3. Register: Call `doca_pcc_dev_algo_register_hist_ids()` to bind histogram IDs to an algorithm slot, enabling remote discovery via PPCC.
4. Update: Inside the CC event handler (`doca_pcc_dev_user_algo()`), call `doca_pcc_dev_histogram_update()` to record observed values and increment the appropriate bin.
5. Query: Read histogram configuration and bin data remotely via PPCC commands (e.g., `0x13` for descriptions, `0x14` for configurations/edges, and `0xC` / `0xD` for bin values).
6. Reconfigure: Enable, disable, or reconfigure histograms remotely via PPCC command `0x15`, or programmatically using `doca_pcc_dev_histogram_disable()` and `doca_pcc_dev_histogram_enable()`.
7. Clear: Call `doca_pcc_dev_histogram_clear()` to reset all bin counters to zero.

> ⚠ The histogram must be disabled prior to clearing.

5.4.4.7.3 7.3. Device API Reference

All histogram functions are declared in `doca_pcc_dev_algo_access.h`, while the histogram handle type and mode enum are defined in `doca_pcc_dev_types.h`.

Category	API Function	Description
Initialization	`doca_pcc_dev_declare_num_of_histograms()`	Sets the number of histograms per port. If not implemented, the default (0) is used.
	`doca_pcc_dev_histogram_init()`	Initializes a histogram instance on a given counter group and histogram ID.
	`doca_pcc_dev_algo_register_hist_ids()`	Registers histogram IDs for an algorithm slot, enabling remote discovery via PPCC.
Data Path	`doca_pcc_dev_histogram_update()`	Records a value. The library automatically finds the matching bin and increments its counter.
Control	`doca_pcc_dev_histogram_enable()`	Enables a previously disabled histogram (re-enables without changing its parameters).
	`doca_pcc_dev_histogram_disable()`	Disables a histogram. This stops the bins from updating, but existing bin values are preserved.
	`doca_pcc_dev_histogram_clear()`	Clears all bin counters to zero. ⚠ The histogram must be disabled before clearing.
Query	`doca_pcc_dev_histogram_get_description()`	Returns a pointer to the histogram's description string.

5.4.4.7.3.1 7.3.1. Usage Example

The following example demonstrates how to initialize two histograms (for RTT and rate distributions) and update them directly from the CC event handler:

```
#include "doca_pcc_dev_algo_access.h"

#define NUM_FUNCTIONAL_COUNTER_GROUPS 1
#define NUM_HISTOGRAMS 2
#define HIST_ID_RTT    0
#define HIST_ID_RATE   1

static doca_pcc_dev_histogram_t rtt_hist;
static doca_pcc_dev_histogram_t rate_hist;

/* Declare the number of histograms needed in memory declaration callback */
void doca_pcc_dev_user_declare_memory(void)
{
    doca_pcc_dev_declare_num_counter_groups(NUM_FUNCTIONAL_COUNTER_GROUPS + NUM_HISTOGRAMS);
    doca_pcc_dev_declare_num_of_histograms(NUM_HISTOGRAMS);
}

void doca_pcc_dev_user_init(uint32_t port_num, uint32_t algo_slot, doca_pcc_dev_algo_meta_t *algo_meta)
{
    /* 10 linearly-spaced bins for RTT (edges in nanoseconds) */
    uint32_t rtt_edges[] = {0, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000};

    doca_pcc_dev_histogram_init(
        port_num, DOCA_PCC_DEV_TARGET_APP_RP,
        /* counter_group_id */ 1,
        HIST_ID_RTT, rtt_edges, 10,
        DOCA_PCC_DEV_HISTOGRAM_MODE_LINEAR,
```

```
        "RTT distribution (ns)", &rtt_hist);

    /* 8 exponentially-growing bins for rate */
    uint32_t rate_edges[] = {0, 1, 2, 4, 8, 16, 32, 64, 128};

    doca_pcc_dev_histogram_init(
        port_num, DOCA_PCC_DEV_TARGET_APP_RP,
        /* counter_group_id */ 2,
        HIST_ID_RATE, rate_edges, 8,
        DOCA_PCC_DEV_HISTOGRAM_MODE_EXPONENTIAL,
        "Rate distribution (Gbps)", &rate_hist);

    /* Register histogram IDs with the algorithm slot */
    uint32_t hist_ids[] = {HIST_ID_RTT, HIST_ID_RATE};
    uint32_t algo_idx = doca_pcc_dev_get_algo_index(port_num, algo_slot);
    doca_pcc_dev_algo_register_hist_ids(algo_idx, hist_ids, NUM_HISTOGRAMS);
}

/* In the CC event handler */
void doca_pcc_dev_user_algo(/* ... event parameters ... */)
{
    uint32_t rtt = /* extract RTT from event */;
    uint32_t rate = /* extract rate from event */;

    doca_pcc_dev_histogram_update(rtt_hist, rtt);
    doca_pcc_dev_histogram_update(rate_hist, rate);
    /* ... rest of algorithm logic ... */
}
```

5.4.4.7.4 7.4. Remote Management via PPCC

Once initialized and registered, histograms can be discovered, queried, and reconfigured dynamically using the Port Programmable Congestion Control (PPCC) register.

Command (cmd_type)	Method	Description & Key Fields
`0x12`	Get	Get histogram IDs for algorithm slot. • `algo_slot` = Algorithm slot number. • Returns: Histogram ID array in `text[]` and count in `value`.
`0x13`	Get	Get histogram description. • `algo_slot` = Histogram ID. • Returns: Description string in `text[]`.
`0x14`	Get	Get histogram configuration and data. • `algo_slot` = Histogram ID. • Returns: Mode in `algo_param_index`, `num_edges` in `value`, enable status in `param_value1`, counter group in `counter_group_id`, and edges in `text[]`.
`0x15`	Set	Set histogram parameters. • `algo_slot` = Histogram ID, `algo_param_index` = Mode. • `value` = 0 (disable), 1 (enable), 2–33 (set edges and enable). Edge values in `text[]`, counter group in `counter_group_id`.

For commands `0x13`, `0x14`, and `0x15`, the `algo_slot` field (offset `0x04`, bits `3:0`) strictly carries the Histogram ID, not the algorithm slot number.

Histogram bin counter values are stored in the histogram's assigned counter group and can be read using standard counter commands (`0xC`, `0xD`) with `cfg_counter_group_en=1` and the appropriate `counter_group_id`.

For detailed PPCC register field descriptions, command examples, and mlxreg usage, refer to section "Port Programmable Congestion Control Register" in the DOCA PCC Application Guide.

5.4.5 DOCA DMA

This guide provides instructions on building and developing applications that require copying memory using Direct Memory Access (DMA).

5.4.5.1 Introduction

⚠ The quality status of DOCA libraries is listed here.

DOCA DMA provides an API to copy data between DOCA buffers using hardware acceleration, supporting both local and remote memory regions.

The library provides an API for executing DMA operations on DOCA buffers, where these buffers reside either in local memory (i.e., within the same host) or host memory accessible by the DPU. See DOCA Core for more information about the memory subsystem.

Using DOCA DMA, complex memory copy operations can be easily executed in an optimized, hardware-accelerated manner.

This document is intended for software developers wishing to accelerate their application's memory I/O operations and access memory that is not local to the host.

5.4.5.2 Prerequisites

This library follows the architecture of a DOCA Core Context, it is recommended read the following sections before:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

5.4.5.3 Library Changes From Previous Releases

5.4.5.3.1 Changes in 2.9.0

N/A

5.4.5.4 Environment

DOCA DMA-based applications can run either on the host machine or on the NVIDIA® BlueField® DPU target.

Copying from Host to DPU and vice versa only works with a DPU configured running in DPU mode as described in BlueField Modes of Operation.

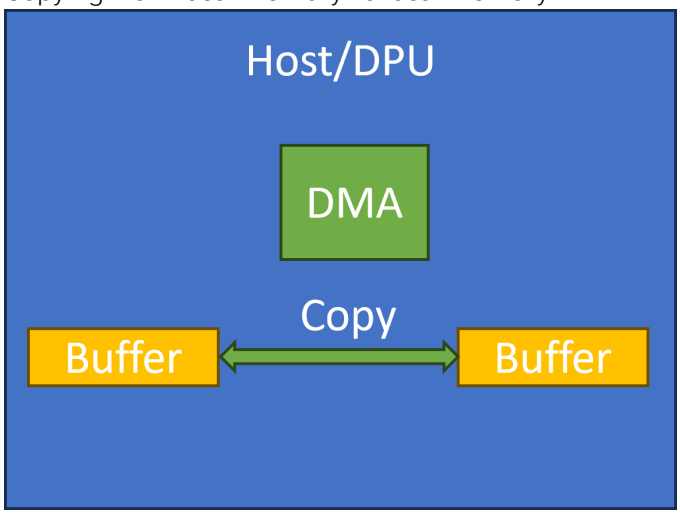
5.4.5.5 Architecture

DOCA DMA is a DOCA Context as defined by DOCA Core. See DOCA Core Context for more information.

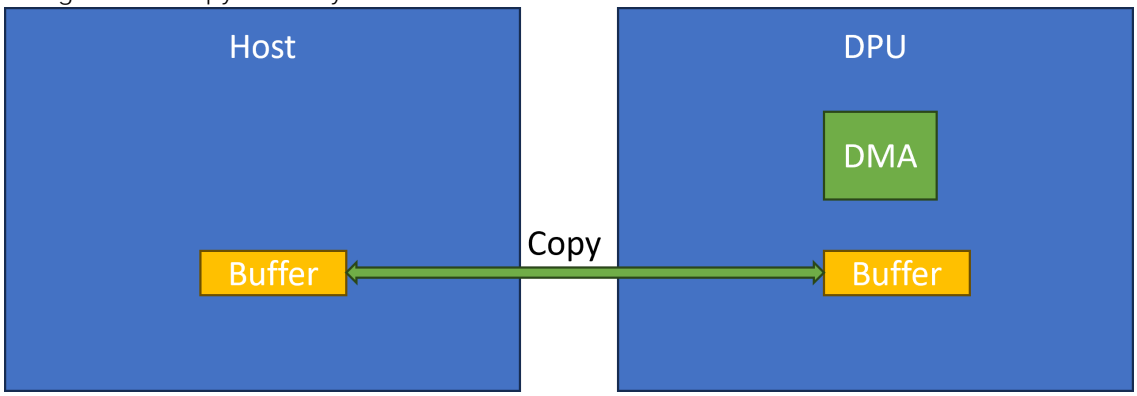
DOCA DMA leverages DOCA Core architecture to expose asynchronous tasks/events that are offloaded to hardware.

DMA can be used to copy data as follows:

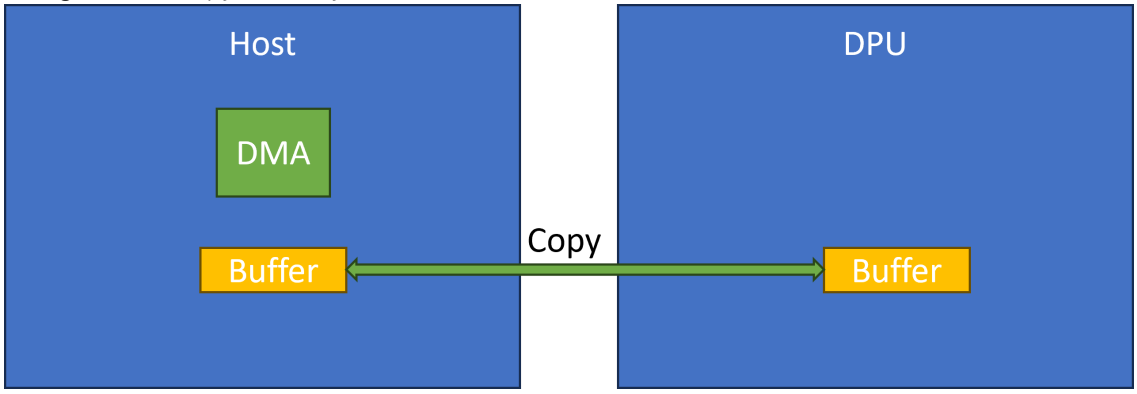
- Copying from local memory to local memory:



- Using DPU to copy memory between host and DPU:



- Using host to copy memory between host and DPU:



5.4.5.5.1 Objects

5.4.5.5.1.1 Device and Device Representor

The DMA library needs a DOCA device to operate. The device is used to access memory and perform the actual copy. See DOCA Core Device Discovery.

For same BlueField DPU, it does not matter which device is used (PF/VF/SF), as all these devices utilize the same hardware component. If there are multiple DPUs, then it is possible to create a DMA instance per DPU, providing each instance with a device from a different DPU.

To access memory that is not local (from the host to the DPU or vice versa), the DPU side of the application must select a device with an appropriate representor. See DOCA Core Device Representor Discovery.

The device must stay valid for as long as the DMA instance is not destroyed.

5.4.5.5.1.2 Memory Buffers

The memory copy task requires two DOCA buffers containing the destination and the source. Depending on the allocation pattern of the buffers, refer to the table in the "Inventory Types" section. To find what kind of memory is supported, refer to the table in section "Buffer Support".

Buffers must not be modified or read during the memory copy operation.

5.4.5.6 Configuration Phase

To start using the library, users must go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context, to allow execution of tasks and retrieval of events.

5.4.5.6.1 Configurations

The context can be configured to match the application use case.

To find if a configuration is supported, or what the min/max value for it is, refer to section "Device Support".

5.4.5.6.1.1 Mandatory Configurations

These configurations are mandatory and must be set by the application before attempting to start the context:

- At least one task/event type must be configured. See configuration of tasks and/or events in sections "Tasks" and "Events" respectively for information.
- A device with appropriate support must be provided upon creation

5.4.5.6.2 Device Support

DOCA DMA requires a device to operate. To select a device, refer to "DOCA Core Device Discovery".

As device capabilities may change (see DOCA Core Device Support), it is recommended to select your device using the following method:

- `doca_dma_cap_task_memcpy_is_supported`

Some devices can allow different capabilities as follows:

- The maximum number of tasks
- The maximum buffer size

5.4.5.6.3 Buffer Support

Tasks support buffers with the following features:

Buffer Type	Source Buffer	Destination Buffer
Local mmap buffer	Yes	Yes
mmap from PCIe export buffer	Yes	Yes
mmap From RDMA export buffer	No	No
Linked list buffer	Yes	Yes

5.4.5.7 Execution Phase

This section describes execution on CPU using DOCA Core Progress Engine.

5.4.5.7.1 Tasks

DOCA DMA exposes asynchronous tasks that leverage the DPU hardware according to the DOCA Core architecture. See DOCA Core Task.

5.4.5.7.1.1 Memory Copy Task

The memory copy task allows copying memory from one location to another. Using buffers as described in Buffer Support.

Task Configuration

Description	API to set the configuration	API to query support
Enable the task	`doca_dma_task_memcpy_set_conf`	`doca_dma_cap_task_memcpy_is_supported`
Number of tasks	`doca_dma_task_memcpy_set_conf`	`doca_dma_cap_get_max_num_tasks`
Maximal buffer size	–	`doca_dma_cap_task_memcpy_get_max_buf_size`

Description	API to set the configuration	API to query support
Maximum buffer list size	–	`doca_dma_cap_task_memcpy_get_max_buf_list_len`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Buffer that points to the memory to be copied	Only the data residing in the data segment is copied
Destination buffer	Buffer that points to where memory is copied	The data is copied to the tail segment extending the data segment

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task is completed successfully:

- The data is copied from the source to the destination
- The destination buffer data segment is extended to include the copied data

Task Completion Failure

If the task fails midway:

- The context may enter stopping state, if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, then the source and destination should not be read/written to
- Source and destination must not overlap
- Other limitations are described in DOCA Core Task

5.4.5.7.2 Events

DOCA DMA exposes asynchronous events to notify on changes that happen unexpectedly, according to DOCA Core architecture.

The only event DMA exposes is common events as described in DOCA Core Event.

5.4.5.8 State Machine

The DOCA DMA library follows the Context state machine as described in DOCA Core Context State Machine.

The following section describes how to move states and what is allowed in each state.

5.4.5.8.1 Idle

In this state it is expected that application:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to section "Configurations"
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop after making sure all tasks have been freed
Stopping	Call progress until all tasks are completed and freed

5.4.5.8.2 Starting

This state cannot be reached.

5.4.5.8.3 Running

In this state it is expected that application:

- Allocates and submits tasks
- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocating a previously configured task
- Submitting a task
- Calling stop

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

5.4.5.8.4 Stopping

In this state it is expected that application:

- Calls progress to complete all inflight tasks (tasks complete with failure)
- Frees any completed tasks

Allowed operations:

- Call progress

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call progress and fatal error occurs
Running	Call stop without freeing all tasks

5.4.5.9 Alternative Datapath Options

DOCA DMA allows data path to be run on the CPU or GPU.

> For the CPU data path, see Execution Phase.

5.4.5.9.1 GPU Datapath

DOCA offers the DOCA GPUNetIO library which provides a programming model for offloading the orchestration of the communication to a GPU CUDA kernel.

The user may run a DMA operation on the GPU data path by configuring the DOCA DMA context used by the application in the following manner:

1. Obtain DOCA CTX by calling **`doca_dma_as_ctx()`**.
2. Set the datapath of the context to GPU by calling **`doca_ctx_set_datapath_on_gpu()`**. For additional information, refer to DOCA Core Alternative Data Path.
3. Finish context configuration and start the context by calling **`doca_ctx_start()`**. For additional information, refer to DOCA Core Context.

After configuring the datapath, the user can obtain a GPU handle for the DOCA RDMA context by calling **`doca_dma_get_gpu_handle()`**. The GPU handle must be passed to a GPU CUDA kernel so the DOCA GPUNetIO CUDA device functions can execute datapath operations. For additional information, refer to section "GPU Functions – RDMA" under "DOCA GPUNetIO" under DOCA GPUNetIO library documentation.

5.4.5.10 DOCA DMA Samples

This section describes DOCA DMA samples based on the DOCA DMA library.

The samples illustrate how to use the DOCA DMA API to do the following:

- Copy contents of a local buffer to another buffer
- Use DPU to copy contents of buffer on the host to a local buffer

> All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

5.4.5.10.1 Running the Samples

1. Refer to the following documents:
   - DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - DOCA Troubleshooting for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

   ```
   cd /opt/mellanox/doca/samples/doca_dma/dma_local_copy
   meson /tmp/build
   ninja -C /tmp/build
   ```

   The binary **doca_dma_local_copy** is created under **/tmp/build/**.
3. Sample (e.g., **doca_dma_local_copy**) usage:

   ```
   Usage: doca_<sample_name> [DOCA Flags] [Program Flags]

   DOCA Flags:
     -h, --help                              Print a help synopsis
     -v, --version                           Print program version information
     -l, --log-level                         Set the (numeric) log level for the program <10=DISABLE,
   20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
     -j, --json <path>                       Parse command line flags from an input json file

   Program Flags:
     -p, --pci_addr <PCI-ADDRESS>            PCI device address
     -t, --text                              Text to DMA copy
     -ns, --num-src-buf                      Number of doca_buf for source buffer
     -nd, --num-dst-buf                      Number of doca_buf for destination buffer
   ```

4. For additional information per sample, use the **-h** option:

   ```
   /tmp/build/<sample_name> -h
   ```

   - The command line option "--num-src-buf" and "--num-dst-buf" are used to show linked-list usage of **doca-buf**.
   - The maximum supported number of elements is 64.
   - And they are only available for **doca_dma_local_copy** and **doca_dma_copy_dpu**, because **doca_dma_copy_host** does not need to construct any **doca_dma_task_memcpy**.

5.4.5.10.2 Samples

These samples are also available on GitHub.

5.4.5.10.2.1 DMA Local Copy

This sample illustrates how to locally copy memory with DMA from one buffer to another on the DPU. This sample should be run on the DPU.

The sample logic includes:

1. Locating DOCA device.
2. Initializing needed DOCA core structures.
3. Populating DOCA memory map with two relevant buffers.
4. Allocating element in DOCA buffer inventory for each buffer.
5. Initializing DOCA DMA memory copy task object.
6. Submitting DMA task.
7. Handling task completion once it is done.
8. Checking task result.
9. Destroying all DMA and DOCA core structures.

Reference:

- **/opt/mellanox/doca/samples/doca_dma/dma_local_copy/dma_local_copy_sample.c**
- **/opt/mellanox/doca/samples/doca_dma/dma_local_copy/dma_local_copy_main.c**
- **/opt/mellanox/doca/samples/doca_dma/dma_local_copy/meson.build**

5.4.5.10.2.2 DMA Copy DPU

This sample should run only after DMA Copy Host is run and the required configuration files (descriptor and buffer) have been copied to the DPU.

This sample illustrates how to copy memory (which contains user defined text) with DMA from the x86 host into the DPU. This sample should be run on the DPU.

The sample logic includes:

1. Locating DOCA device.
2. Initializing needed DOCA core structures.
3. Reading configuration files and saving their content into local buffers.
4. Allocating the local destination buffer in which the host text is to be saved.
5. Populating DOCA memory map with destination buffer.
6. Creating the remote memory map with the export descriptor file.
7. Creating memory map to the remote buffer.
8. Allocating element in DOCA buffer inventory for each buffer.
9. Initializing DOCA DMA memory copy task object.
10. Submitting DMA task.
11. Handling task completion once it is done.
12. Checking DMA task result.
13. If the DMA task ends successfully, printing the text that has been copied to log.
14. Printing to log that the host-side sample can be closed.
15. Destroying all DMA and DOCA core structures.

Reference:

- **/opt/mellanox/doca/samples/doca_dma/dma_copy_dpu/dma_copy_dpu_sample.c**
- **/opt/mellanox/doca/samples/doca_dma/dma_copy_dpu/dma_copy_dpu_main.c**
- **/opt/mellanox/doca/samples/doca_dma/dma_copy_dpu/meson.build**

5.4.5.10.2.3 DMA Copy Host

> ⚠ This sample should be run first. It is user responsibility to transfer the two configuration files (descriptor and buffer) to the DPU and provide their path to the DMA Copy DPU sample.

This sample illustrates how to allow memory copy with DMA from the x86 host into the DPU. This sample should be run on the host.

The sample logic includes:

1. Locating DOCA device.
2. Initializing needed DOCA core structures.
3. Populating DOCA memory map with source buffer.
4. Exporting memory map.
5. Saving export descriptor and local DMA buffer information into files. These files should be transferred to the DPU before running the DPU sample.
6. Waiting until DPU DMA sample has finished.
7. Destroying all DMA and DOCA core structures.

Reference:

- **/opt/mellanox/doca/samples/doca_dma/dma_copy_host/dma_copy_host_sample.c**
- **/opt/mellanox/doca/samples/doca_dma/dma_copy_host/dma_copy_host_main.c**
- **/opt/mellanox/doca/samples/doca_dma/dma_copy_host/meson.build**

5.4.6 DOCA Comch

This guide provides instructions on building and developing applications that require communication channels between the x86 host and the BlueField Arm cores.

5.4.6.1 Introduction

> ⚠ The quality status of DOCA libraries is listed here.

DOCA Comch provides a communication channel between client applications on the host and servers on the BlueField Arm. It introduces features such as high-performance data path over the consumer-producer API, as well as working with DOCA progress engine and other standard DOCA Core objects.

Benefits of using DOCA Comch:

- Security – the communication channel is isolated from the network
- Network independent – the state of the communication channel does not depend on the state and configuration of the network
- Ease of use

DOCA Comch provides two different data path APIs:

- Basic DOCA Comch send/receive for control messages

- High bandwidth, low latency, zero-copy, multi-producer, multi-consumer API

The following table summarizes the differences between the two data path APIs:

Features	Basic Send/ Receive	Fast Path (using doca_comch_consumer/ doca_comch_producer)
Zero-copy	No	Yes
Takes network bandwidth	Yes	No
Isolated from network	Yes	Yes
Max msg size	Fixed	1GB or more (depends on hardware cap)
Multi-threaded	Safe for a single thread	Allows creation of consumer/producers per thread.
Multi-consumer	No	Yes
Multi-producer	Yes – allows multiple clients per server	Yes – allow multiple producers/consumers per connection
Requires `doca_mmap` and `doca_buf`	No	Yes

DOCA Comch does not support event-triggered completions.

5.4.6.2 Prerequisites

This library follows the architecture of a DOCA Core Context, it is recommended to read the following sections before:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem (fast path only)

5.4.6.3 Changes From Previous Release

N/A

5.4.6.4 Environment

DOCA Comch based applications can run either on the host machine or on the NVIDIA BlueField Arm.

Sending messages between the host and BlueField Arm can only be run with a BlueField configured with a mode as described in BlueField Modes of Operation.

For basic DOCA Comch send and receive, the following configuration is required:

- `doca_comch_server` context must run on the BlueField Arm cores

- **doca_comch_client** context must run on the host machine

> Producer and consumer objects can run on both the host and BlueField Arm cores. However, there must be a valid client/server connection already established on the channel.

5.4.6.5 Architecture

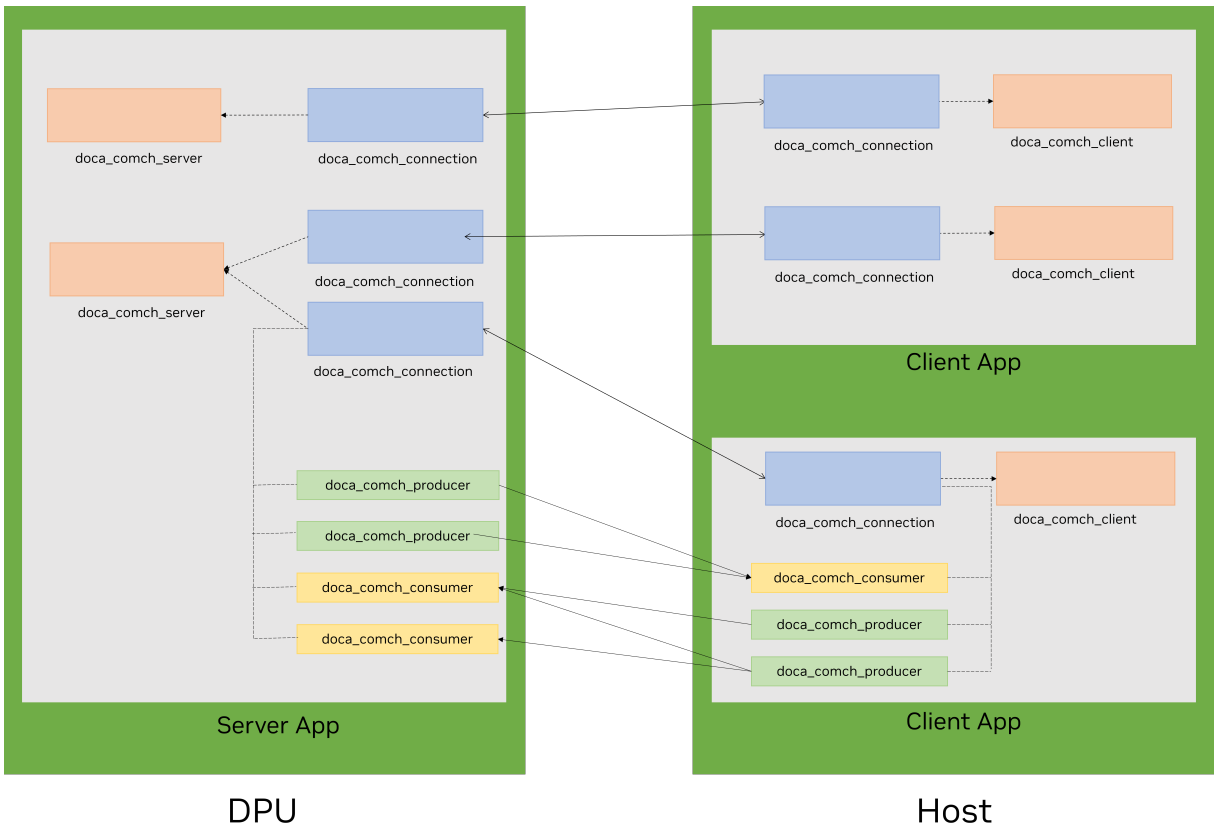
DOCA Comch is comprised of four DOCA Core Contexts. All DOCA Comch contexts leverage DOCA Core architecture to expose asynchronous tasks/events that are offloaded to hardware.

A **doca_comch_server** context runs on the BlueField Arm and listens for incoming connections from the host side. Such host side connections are initiated by a **doca_comch_client** context.

Servers can receive connections from multiple clients in parallel, however, a client can only connect with one server. An established 1-to-1 connection between a client and a server is represented by a **doca_comch_connection**.

Once an established connection exists between a client and a server, the **doca_comch_producer** and **doca_comch_consumer** contexts can be used to run fast path channels.

The following diagram provides examples of the contexts use:



5.4.6.5.1 Objects

	Description	Location	Scope
`doca_comch_server`	Allows applications on the BlueField Arm cores to listen on a specific server name and accept new incoming connection from the host	BlueField Arm only	Per host PCIe function (`doca_dev` + `doca_dev_rep`)
`doca_comch_client`	Allows client applications to connect to a specific server name on the BlueField Arm cores	Host only	Per host PCIe function (`doca_dev`)
`doca_comch_connection`	A connection handle created on the client side or the server side when a new connection is established. This handle is used to send/receive messages or to create `doca_comch_consumer` s and `doca_comch_producer` s.	BlueField Arm and host	Per client server pair
`doca_comch_producer`	A handle for a FIFO-like send queue that provides a zero-copy API to send messages to a specific `doca_comch_consumer` on the same `doca_comch_connection` . Multiple `doca_comch_producer` s can be created per `doca_comch_connection` .	BlueField Arm and host	Per `doca_comch_connection`
`doca_comch_consumer`	A handle for a FIFO-like receive queue that provides a zero-copy API to receive messages from a `doca_comch_producer`	BlueField Arm and host	Per `doca_comch_connection`

5.4.6.5.2 Security Considerations

- DOCA Comch guarantees:
  - The client is connected to the server by providing the exact server name on the client side
  - Only clients on the PF/VF/SF represented by the `doca_dev_rep` provided upon server creation can connect to the server
  - The connection requests and data path are isolated from the network
- DOCA Comch does not provide security at the application level:
  - It is up to the user to implement application-level security and verify the identity of the client application
  - A server handles applications from a single PF/VF/SF. If a server application detects a compromised client application, the server app should consider all clients (from that PF/VF/SF) compromised.

5.4.6.5.3 Initialization Flow

5.4.6.5.3.1 doca_comch_server Initialization Flow

1. A `doca_comch_server` is created on a specific `doca_dev` and a specific `doca_dev_rep` .

2. A **doca_comch_server** must have a unique name per **doca_dev/doca_dev_rep** (i.e., two servers on the same **doca_dev** and **doca_dev_rep** cannot have the same name).
3. Once **doca_ctx_start()** is called, the **doca_comch_server** can start receiving new connection requests.
4. For the **doca_comch_server** to process new connection requests and messages, the user must periodically call **doca_pe_progress()**.
5. When a new connection request arrives, **doca_comch_server** calls the connection request handler function and passes a **doca_comch_connection** object.

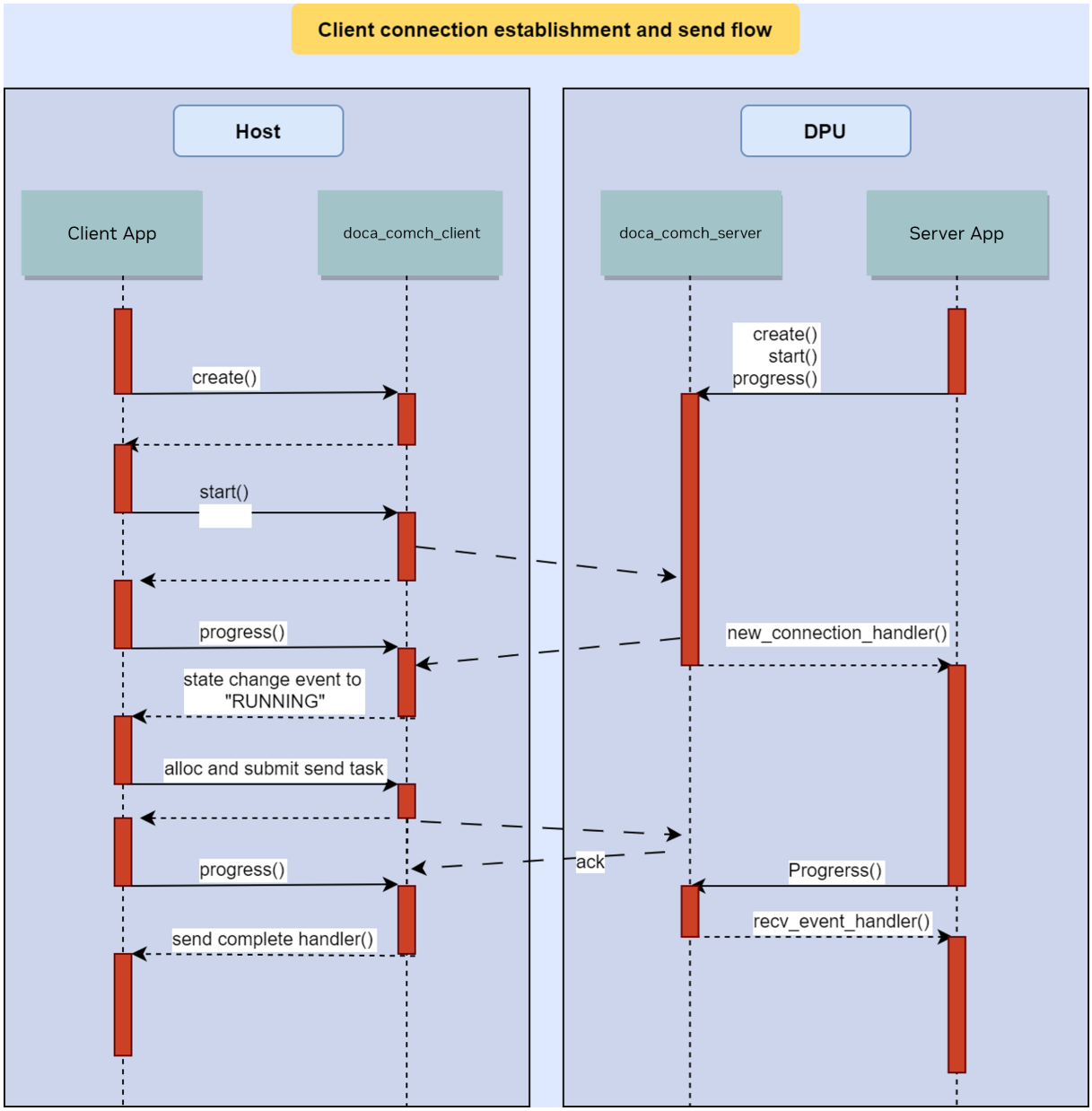
The server can now send and receive messages on the connection represented by **doca_comch_connection**.

5.4.6.5.3.2 doca_comch_client Initialization Flow

1. A **doca_comch_client** is created on a specific **doca_dev** is targeting a specific **doca_comch_server**.
2. Once **doca_ctx_start()** is called, **doca_comch_client** asynchronously tries to connect to the server.
3. To establish the connection and receive messages, the user must periodically call **doca_pe_progress()**.
4. When the connection is established, **doca_comch_client** calls the state change callback indicating state change to "RUNNING".

The client can now send a receive messages.

The following diagram describes the initialization of a basic client/server connection on DOCA Comch:

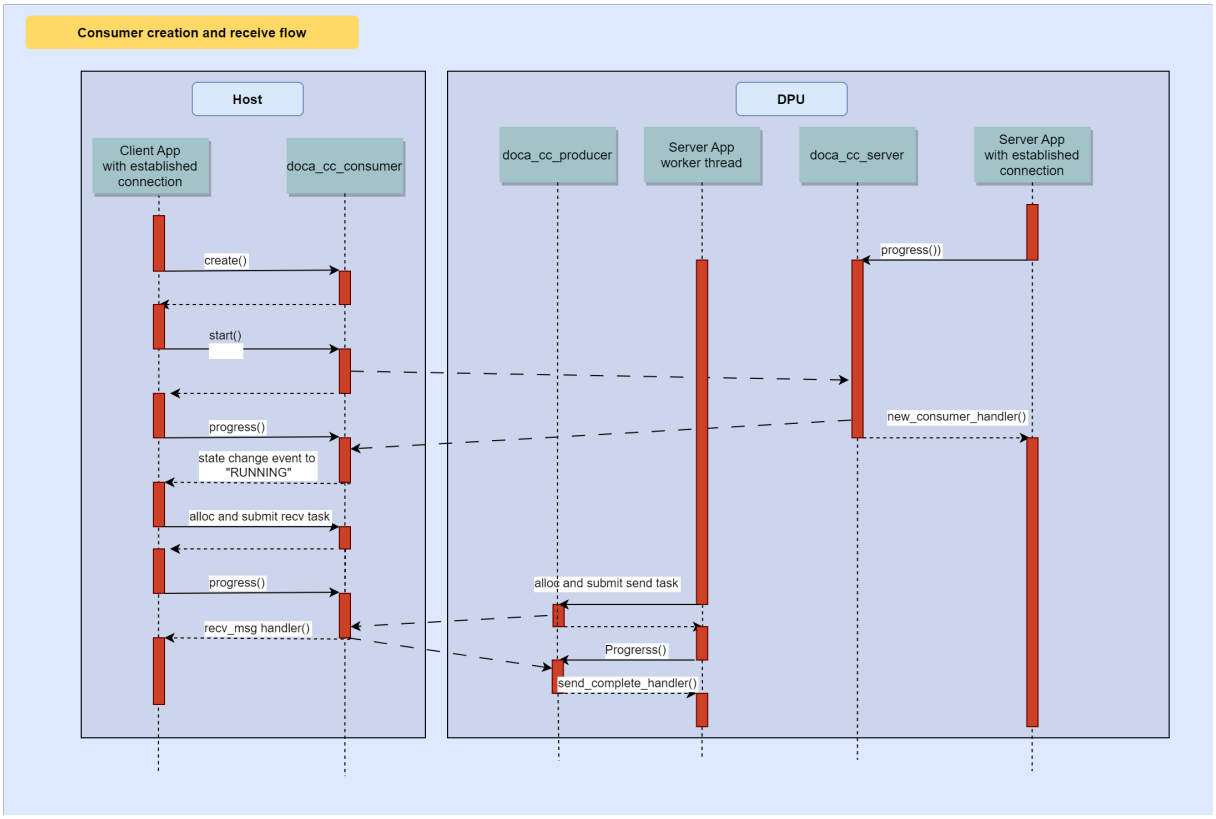


5.4.6.5.3.3 doca_comch_consumer Initialization Flow

1. A **doca_comch_consumer** is created on a specific **doca_comch_connection**.
2. **doca_pe_progress()** must be periodically called on the client/server PE to allow registration of the consumer.
3. After the **doca_comch_consumer** moves to "RUNNING" state:
    a. **doca_comch_consumer** notifies its existence to the peer (invoking a new consumer event).
    b. The application can start posting receive tasks.

c. A **doca_comch_producer** on the peer side can start sending messages to that consumer.

The initialization flow is described in the following diagram:



5.4.6.5.4 Teardown Flow

The teardown flow must be executed in the following order, otherwise errors may occur.

5.4.6.5.4.1 Disconnecting Specific Connection

The proper disconnection process for a specific connection consists of the following steps:

1. Stop all consumers and producers linked to the connection.
2. Server/client:
   a. For server, a connection can be disconnected using **doca_comch_server_disconnect()**. If there are any active producers/consumers linked to the connection, the disconnect would fail. A disconnection notifies the client and initiates teardown on that side too.
   b. For client, since there is only one connection at any given time, the connection can be disconnected by calling **doca_ctx_stop()**. If there are any active producers/consumers, the command would fail. Stopping the client context notifies the server of the disconnection and causes a disconnection of the connection on it.

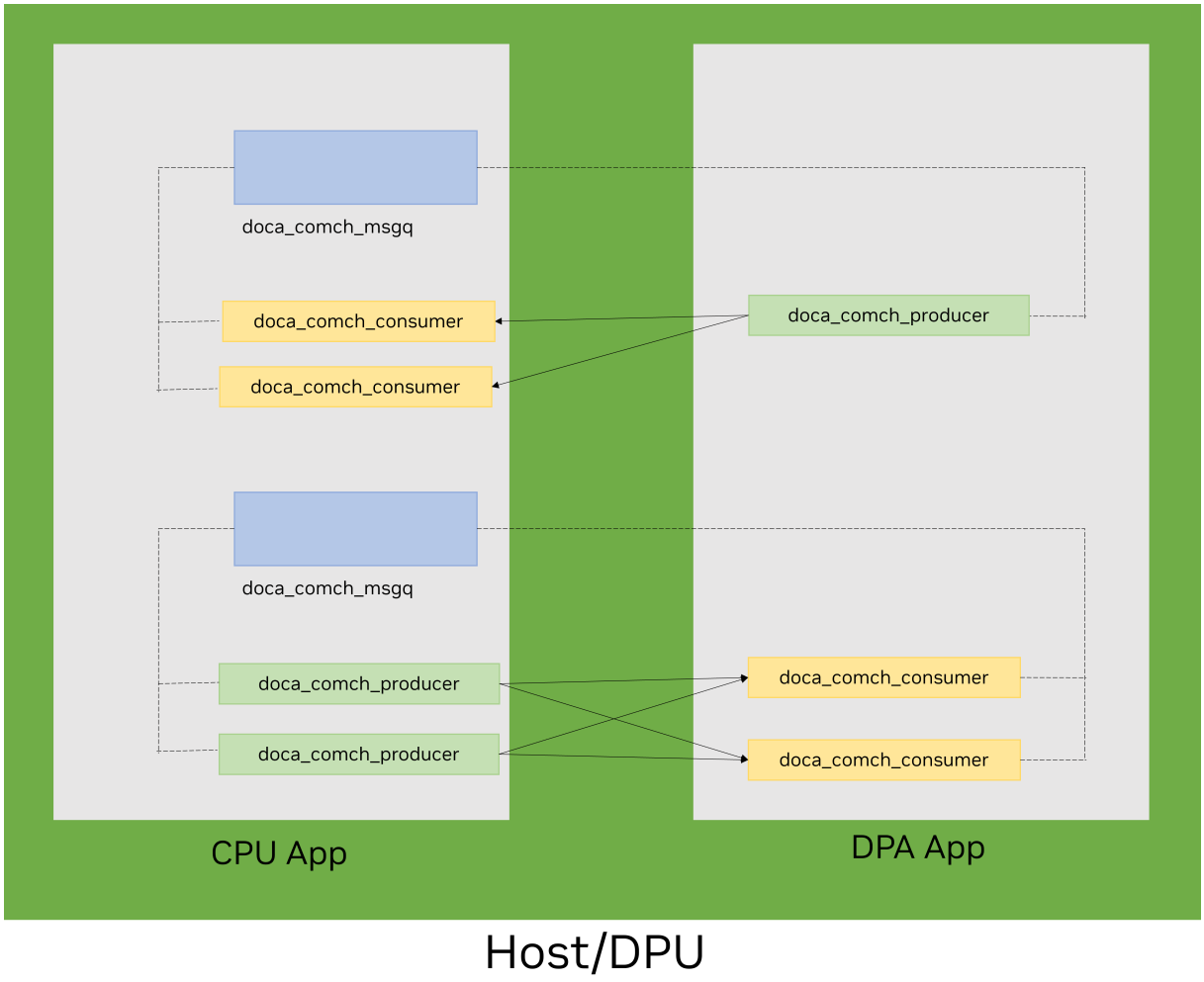
5.4.6.5.4.2 Tearing Down DOCA Comch

The proper teardown for a DOCA Comch context consists of the following:

1. Stop all consumers and producers linked to the context.
2. Call **doca_ctx_stop()** . If there are any active connections, they would all be disconnected. If there are any active consumers/producers, the command would fail. Disconnecting/ stopping the context informs all active peers of the disconnection, and causes teardown (on clients) or disconnection (on server). Calling **doca_ctx_stop()** successfully moves the context to "stopping" state.
3. After moving to stopping state, **doca_pe_progress()** must be called until the context moves to idle state.

5.4.6.5.5 MsgQ (DPA Communication)

DOCA Comch MsgQ leverages the existing consumer/producer model to allow communication between host/BlueField and DPA.



Since communication between the host/BlueField and DPA is local, there is no need to create a server, client, or connection. Instead the user can create a MsgQ and use it to create producers and consumers directly.

When creating a consumer/producer using the MsgQ, it becomes possible to use them in the DPA application as well as the CPU application:

- The CPU application can utilize existing consumer/producer APIs for communication
- The DPA application has a different set of APIs that are usable within a DPA application

5.4.6.5.5.1 Communication Direction

Every instance of a MsgQ can only support a single communication direction as follows:

- Communication from host/BlueField to DPA
  - This direction may be specified using **doca_comch_msgq_set_dpa_consumer**
  - Consumers created from this MsgQ are referred to as DPA consumers, while producers are CPU producers
- Communication from DPA to host/BlueField
  - This direction may be specified using **doca_comch_msgq_set_dpa_producer**
  - Consumers created from this MsgQ are referred to as CPU consumers, while producers are DPA producers

To support bidirectional communication in an application, the user has to create 2 MsgQ instances, as shown in the above diagram.

5.4.6.6 Configuration Phase

To start using the library, users must go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context to allow execution of tasks and retrieval of events.

5.4.6.6.1 Configurations

The context can be configured to match the application use case.

To find out if a certain configuration is supported, or what the min/max value for it is, refer to Device Support.

5.4.6.6.1.1 Mandatory Configurations

These configurations are mandatory and must be set by the application before attempting to start the context:

- For a basic send/receive client or server:
  - A send task callback
  - A receive event callback
  - A device with appropriate support must be provided on creation
  - A valid server name must be provided on creation (for clients this is the server to connect to)
  - A connection event callback (server only)

- For fast path producer or consumer:
  - A device with appropriate support must be provided on creation
  - An established client to server connection must be provided on creation
  - A **doca_mmap** with PCIe read/write permissions of where data should be received must be provided on creation (consumer only)
  - A post receive task callback (consumer only)
  - A send task callback (producer only)
  - A new consumer callback (triggered upon creation/destruction of a remove consumer)
- For MsgQ fast path producer or consumer:
  - A started MsgQ must be provided on creation
  - A DPA instance must be provided (DPA consumer/producer only)
  - A DPA consumer completion context must be connected (DPA consumer only)
  - A DPA completion context must be attached (DPA producer only)
  - A post receive task callback (CPU consumer only)
  - The number of receive operations (DPA consumer only)
  - A send task callback (CPU producer only)
  - The number of send operations (DPA producer only)

5.4.6.6.1.2 Optional Configurations

The following configurations are optional, if they are not set then a default value will be used:

For basic send/receive client:

- **doca_comch_(server|client)_set_max_msg_size** – set the maximum size of message that can be sent. If set, it must be matching between server and client.
- **doca_comch_(server|client)_set_recv_queue_size** – set the size of the queue to receive new messages on

For fast path consumers:

- **doca_comch_consumer_set_imm_data_len** – set the length of immediate data that a consumer can receive.

5.4.6.6.2 Device Support

DOCA Comch requires a device to operate. For instructions on picking a device, see DOCA Core Device Discovery.

As device capabilities are subject to change (see DOCA Core Device Support), it is recommended to select a device using the following methods:

- For basic client and server:
  - **doca_comch_cap_server_is_supported**
  - **doca_comch_cap_client_is_supported**
- For extended fast path functionality:
  - **doca_comch_producer_cap_is_supported**
  - **doca_comch_consumer_cap_is_supported**

Some devices can allow different capabilities as follows:

- The maximum length server name
- The maximum message size
- The maximum receive queue length
- The maximum clients that can connect to a server
- The maximum number of send tasks or post receive tasks
- The maximum buffer length for fast path
- The maximum immediate data supported by a fast path consumer

5.4.6.6.3 Buffer Support

Basic send and receive between a client and server does not use DOCA buffers and so has no restrictions on buffer type.

- For producers, supplied buffers need only be from a local mmap
- For consumers, post receive buffers are required to be from a PCIe export mmap

> ⚠ Chained buffers are not supported in DOCA Comch.

5.4.6.7 Execution Phase

This section describes execution on CPU using DOCA Core Progress Engine. For additional execution environments, refer to section "Alternative Datapath Options".

5.4.6.7.1 Tasks

DOCA Comch exposes asynchronous tasks that leverage the BlueField hardware according to DOCA Core architecture.

5.4.6.7.1.1 Control Channel Send Task

This task allows the sending of messages between connected client and server objects.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Number of tasks	`doca_comch_server_task_send_set_conf` `doca_comch_client_task_send_set_conf`	`doca_comch_cap_get_max_send_tasks`
Maximal message size	`doca_comch_server_set_max_msg_size` `doca_comch_client_set_max_msg_size`	`doca_comch_server_get_max_msg_size` `doca_comch_client_get_max_msg_size`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Peer	Established client/server connection	–

Name	Description	Notes
Message	Data string to send to remote client/ server	The is no requirement for the message to be in DOCA mmap registered memory
Length	Number of bytes in the message	Must not exceed configured max size

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully:

- The message is delivered to the connections remote client/server
- A receive event is triggered on the remote side

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The message is not delivered to the remote side

Task Limitations

- The operation is not atomic
- Once the task has been submitted, then the message should not be updated
- Other limitations are described in DOCA Core Task

5.4.6.7.1.2 Consumer Post Receive Task

This task allows consumer objects to publish buffers which are available for remote producers to write to.

> ⚠ A Post Receive task may have a NULL buffer if it only wishes to receive immediate data.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_comch_consumer_task_post_recv_set_conf`	`doca_comch_consumer_cap_is_supported`
Number of tasks	`doca_comch_consumer_task_post_recv_set_conf`	`doca_comch_consumer_cap_get_max_num_tasks`
Maximal buffer size	–	`doca_comch_consumer_cap_get_max_buf_size`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Buffer	Buffer that the consumer can receive data on	Data is appended to the tail of the buffer Buffers `doca_mmap` must have `DOCA_ACCESS_FLAG_PCI_READ_WRITE` flag set.

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

The task only completes once a producer has written to the advertised buffer (or immediate data, or both), not when the post receive has completed.

Upon successful completion, the buffer contains the data written by the producer and its length is updated appropriately.

Task Completion Failure

Task failure occurs if a buffer has not been successfully posted to receive data.

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- Producers are not aware of the buffer so would not write to it

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the buffer should not be read/written to
- Buffer must come from memory with PCIe read/write access
- Chained buffer lists are not supported
- MsgQ consumer does not support providing `doca_buf`, and can only receive immediate data
- Other limitations are described in DOCA Core Task

5.4.6.7.1.3 Producer Send Task

This task allows producer objects to copy buffers for use by remote consumers.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_comch_producer_task_send_set_conf`	`doca_comch_producer_cap_is_supported`
Number of tasks	`doca_comch_producer_task_send_set_conf`	`doca_comch_producer_cap_get_max_num_tasks`
Maximal buffer Size	–	`doca_comch_producer_cap_get_max_buf_size`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Buffer	Buffer that should be copied to a consumer	Only the data residing in the data segment is copied
Immediate data	Short byte array to add to the post receive completion entry	This is not a zero copy operation but does improve latency for small payloads
Immediate data length	Length of data immediate data pointed to	Maximum length is determined/set by individual consumers
Consumer ID	Identifier for the target consumer to write to	Active consumers and their IDs are advertised through consumer events

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task is completed successfully:

- The data is copied form the buffer to the next free buffer posted by the given consumer
- Consumers process buffers from a given consumer in the order they are sent

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination **doca_buf** objects are not modified
- The destination memory may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the buffer should not be read/written to
- The buffer length should not be greater than consumer post receive buffers (an invalid value is returned otherwise)
- MsgQ producer does not support providing **doca_buf**, and can only send immediate data
- All limitations described in DOCA Core Task

5.4.6.7.2 Events

DOCA Comch exposes asynchronous events to notify about changes that happen out of the blue, according to the DOCA Core architecture. See DOCA Core Event.

Common events as described in DOCA Core Event.

5.4.6.7.2.1 Control Channel Receive Event

This event triggers whenever a remote client/server has sent a message to the local client/server object.

Event Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	`doca_comch_server_event_msg_recv_register` `doca_comch_client_event_msg_recv_register`	–

Event Trigger Condition

The event is triggered when a remote message is received on any currently active connection associated with the client or server.

Event Output

Upon event detection, the registered callback is triggered, passing the following parameters:

- A pointer to the message data

  > The data is only valid in the context of the callback.

- The length in bytes of the message
- The active connection on which the message was received

5.4.6.7.2.2 Connection Status Changed Event (Server Only)

This event provides asynchronous updates on the state of any connections associated with a server.

> A client object can only connect to a single server, so its connection state can be tracked through its `doca_ctx` state and the generic `doca_ctx_set_state_changed_cb` function.

Event Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	`doca_comch_server_event_connection_status_changed_register`	–

Event Trigger Condition

The event is triggered when a new connection is either established or a current connection disconnected on a server.

Event Output

Separate callbacks are registered for connection or disconnection events with the appropriate one triggered based on the specific event.

Both callbacks contain a Boolean indicating if the connection or disconnection was successful.

5.4.6.7.2.3 Consumer Event

This event indicates that a new consumer object has been created or an existing consumer object has been destroyed.

Event Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	`doca_comch_server_event_consumer_register` `doca_comch_client_event_consumer_register`	–

Event Trigger Condition

The event is triggered whenever a new consumer is created or a current consumer destroyed on the remote side of an established DOCA Comch connection.

Event Output

The event hits a separate callback for either the creation or destruction of a consumer.

Callback parameters include:

- The established DOCA Comch connection on which the consumer is connected (on the remote side)
- The ID of the consumer (a unique value per Comch connection)

5.4.6.8 State Machine

The DOCA Comch library follows the Context state machine described in DOCA Core Context State Machine.

The following section describes how to move to the state and what is allowed in each state.

5.4.6.8.1 Idle

In this state it is expected that the application either:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to Configurations
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop after making sure all tasks have been freed
Stopping	Call progress until all tasks are completed and freed

5.4.6.8.2 Starting

In this state it is expected that the application will:

- Call progress to allow transition to next state (e.g., when a connection attempt completes)

Allowed operations:

- Call progress

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

5.4.6.8.3 Running

In this state, it is expected that the application:

- Allocates and submit tasks
- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocate a previously configured task
- Submit an allocated task
- Call stop

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration
Starting	Call progress until context state transitions

5.4.6.8.4 Stopping

In this state, it is expected that the application will:

- Free any completed tasks

Allowed operations:

- Allocate previously configured task
- Submit an allocated task
- Call stop

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call progress and fatal error occurs

Previous State	Transition Action
Running	Call stop without freeing all tasks

5.4.6.9 Alternative Datapath Options

DOCA Comch can be run on as part of DPA data path, using the MsgQ.

5.4.6.9.1 DPA Support

Using the MsgQ it is possible to create consumer/producer on the DPA. They follow the definition described in DOCA Core DPA.

Since these objects can be used in DPA, they have DPA APIs that can be used to perform the data path operations expanded on in the following subsections.

5.4.6.9.1.1 Consumer Ack

The **doca_dpa_dev_comch_consumer_ack** API prepares the DPA consumer to receive a number of immediate messages from CPU producers.

Configuration

Description	API to Set the Configuration	API to Query Support
Queue Size	**doca_comch_consumer_set_dev_max_num_recv**	–

Input

Name	Description	Notes
Number of Messages	A number describing how many additional immediate messages this consumer can receive	Must not exceed the queue size

Completion

Whenever a message is received from the CPU producer a completion element is generatedand can be polled using **doca_dpa_dev_comch_consumer_get_completion** .

Using the generated completion, it is possible to get the following outputs:

Name	Description	Notes
Immediate Message	A pointer to the immediate message that the CPU producer sent	The message lifetime is the same as the completion element lifetime. That is, once the completion is acked using **doca_dpa_dev_comch_consumer_completion_ack** , the pointer is no longer valid. To retain the message past the completion lifetime, the user must copy the contents of the message.
Immediate Message Length	The length in bytes of the immediate message that the CPU producer sent	

Name	Description	Notes
Producer ID	The ID of the CPU producer that sent the message	User can find the IDs of each producer by using `doca_comch_producer_get_id`

Limitations

- The maximal immediate message size is 32 bytes

5.4.6.9.1.2 Producer Post Send Immediate Only

The `doca_dpa_dev_comch_producer_post_send_imm_only` API sends an immediate message to the CPU consumer. Once the message arrives at the CPU consumer side, the CPU consumer receive task completes.

The CPU producer must have posted a receive task prior to this. The user can verify if the consumer can receive the message using `doca_dpa_dev_comch_producer_is_consumer_empty`. Note, however, that this may add overhead.

Configuration

Description	API to Set the Configuration	API to Query Support
Queue Size	`doca_comch_producer_set_dev_max_num_send`	–

Input

Name	Description	Notes
Immediate Message	Short byte array to be sent to the CPU consumer	This is not a zero copy operation but does improve latency for small payloads
Immediate Message Length	Length of the message the immediate message points to	The maximum length is 32 bytes
Consumer ID	Identifier for the target CPU consumer to write to	User can find the IDs of each consumer by using `doca_comch_consumer_get_id`
Completion Requested	Flag indicating whether to generate a completion once the send is completed	This refers to the DPA producer completion which is separate from the completion the CPU consumer receives • 0 – no completion • 1 – otherwise

Completion

Once the message arrives to the CPU consumer, a completion element is generated, indicating that the send is complete (this is separate from the completion the CPU consumer receives) and can be polled using `doca_dpa_dev_get_completion`.

Using the generated completion, it is possible to get the following outputs:

Name	Description	Notes
Producer User Data	Producer user data provided during configuration of the producer	User data previously set using `doca_ctx_set_user_data` when configuring this producer. User data which is returned belongs to the DPA producer this completion has been generated for, and can be used to identify the specific producer.

Limitations

- The maximal immediate message size is 32 bytes

5.4.6.9.1.3 Producer DMA Copy

The `doca_dpa_dev_comch_producer_dma_copy` API performs a DMA copy operation and, once the copy operation is done, sends an immediate message to the CPU consumer. Once the message arrives at the CPU consumer side, the CPU consumer receive task completes.

The CPU producer must have posted a receive task prior to this. The user can verify if the consumer can receive the message using `doca_dpa_dev_comch_producer_is_consumer_empty`. Note, however, that this may add overhead.

Configuration

Description	API to Set the Configuration	API to Query Support
Queue Size	`doca_comch_producer_set_dev_num_recv`	–

Input

Name	Description	Notes
Destination Mmap	Mmap representing the memory to be used as the destination of the copy operation	This mmap must have `LOCAL_READ_WRITE` access enabled
Destination Address	The address to be used as the destination of the copy operation	The address and copy length must be within the range of the destination mmap's memory range
Source Mmap	Mmap representing the memory to be used as the source of the copy operation	This mmap must have `LOCAL_READ` access enabled
Source Address	The address to be used as the source of the copy operation	The address and copy length must be within the range of the source mmap's memory range
Length	The length of the copy operation	Source and destination addresses must not overlap
Immediate Message	Short byte array to be sent to the CPU consumer once the copy operation is done	This is not a zero copy operation but does improve latency for small payloads
Immediate Message Length	Length of the message the immediate message points to	The maximum length is 32 bytes
Consumer ID	Identifier for the target CPU consumer to write to	User can find the IDs of each consumer using `doca_comch_consumer_get_id`

Name	Description	Notes
Completion Requested	Flag indicating whether to generate a completion once the send is completed	This refers to the DPA producer completion which is separate from the completion the CPU consumer receives • 0 – no completion • 1 – otherwise

Completion

Once copy is complete and the message arrives to the CPU consumer, a completion element is generated, indicating that the copy is complete (this is separate from the completion the CPU consumer receives) and can be polled using `doca_dpa_dev_get_completion`.

Using the generated completion, it is possible to get the following outputs:

Name	Description	Notes
Producer User Data	Producer user data provided during configuration of the producer	The user data set using `doca_ctx_set_user_data` when configuring this producer. The user data which is returned belongs to the DPA producer this completion has been generated for, and can be used to identify the specific producer.

Limitations

- The maximal immediate message size is 32 bytes

5.4.6.9.2 GPU Datapath Support

DOCA offers the DOCA GPUNetIO library, which provides a programming model for running DOCA Comch producers and consumers on a GPU CUDA Kernel.

5.4.6.9.2.1 Key Requirements and Limitations

- Client-only support: The GPU datapath is currently supported only for `doca_comch_client` objects and their associated producers/consumers. It is not supported for `doca_comch_server`.
- No mixed datapaths: Once a `doca_comch_client` is configured for the GPU datapath, all producers or consumers created from that client must also use the GPU datapath.
- Progress engine required: All GPU datapath contexts must still be connected to a Progress Engine (`doca_pe_connect_ctx()`). The PE is required to handle connection management events, such as remote Producers or Consumers connecting or disconnecting.

5.4.6.9.2.2 Configuration

Configuring the GPU datapath requires setting the datapath on both the parent client object and the individual Producer/Consumer contexts.

Configure the DOCA Comch Client

1. Obtain the DOCA Context from the client by calling `doca_comch_client_as_ctx()`.
2. Set the context's datapath to GPU by calling `doca_ctx_set_datapath_on_gpu()`. (For details, see DOCA Core Alternative Data Path.)

3. Finish context configuration and start it by calling **doca_ctx_start()**. (For details, see DOCA Core Context.)

Configure the DOCA Comch Producer or Consumer

1. Obtain the DOCA Context from the Producer or Consumer by calling **doca_comch_producer_as_ctx()** or **doca_comch_consumer_as_ctx()**.
2. Set the context's datapath to GPU by calling **doca_ctx_set_datapath_on_gpu()**. (For details, see DOCA Core Alternative Data Path.)
3. Finish context configuration and start the context by calling **doca_ctx_start()**. (For details, see DOCA Core Context.)

5.4.6.9.2.3 Obtaining the GPU Handle

After the Producer or Consumer context is in the "running" state, the application can retrieve a GPU handle by calling **doca_comch_producer_get_gpu_handle()** or **doca_comch_consumer_get_gpu_handle()**.

This handle must be passed to a GPU CUDA kernel, which allows the DOCA GPUNetIO CUDA device functions to execute datapath operations. For more information, refer to the "GPU Functions – Comch" section in the DOCA GPUNetIO library documentation.

5.4.6.10 DOCA Comch Samples

This section describes DOCA Comch samples based on the DOCA Comch library.

The samples illustrate how to use the DOCA Comch API to do the following:

- Set up a client/server between host and BlueField Arm cores and use it to send text messages
- Configure fast path producers and consumers, and send messages between them

> All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

5.4.6.10.1 Running the Samples

1. Refer to the following documents:
   - DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - DOCA Troubleshooting for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

   ```
   cd /opt/mellanox/doca/samples/doca_comch/<sample_name>
   meson /tmp/build
   ninja -C /tmp/build
   ```

   The binary **doca_<sample_name>** is created under **/tmp/build/**.

3. All DOCA Comch samples accept the same input arguments:

Sample	Argument	Description

`doca_comch_ctrl_path_server`
`doca_comch_ctrl_path_client`
`doca_comch_data_path_high_speed_server`
`doca_comch_data_path_high_speed_client`
	`-p` , `--pci-addr`	DOCA Comch device PCIe address

	`-r` , `--rep-pci`	DOCA Comch device representor PCIe address (required only on BlueField Arm)
	`-t` , `--text`	Text to be sent to the other side of channel (overwrites default)

4. For additional information per sample, use the `-h` option:

```
/tmp/build/<sample_name> -h
```

5.4.6.10.2 Samples

These samples are also available on GitHub.

5.4.6.10.2.1 DOCA Comch Control Path Client/Server

`doca_comch_ctrl_path_server` must be run on the BlueField Arm side and started before `doca_comch_ctrl_path_client` is started on the host.

This sample sets up a client server connection between the host and BlueField Arm cores.

The connection is used to pass two messages, the first sent by the client when the connection is established and the second by the server on receipt of the client's message.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the core DOCA structures.
3. Initializing and configuring client/server contexts.
4. Registering tasks and events for sending/receiving messages and tracking connection changes.
5. Allocating and submitting tasks for sending control path messages.
6. Handling event completions for receiving messages.
7. Stopping and destroying client/server objects.

References:

- `/opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_client/comch_ctrl_path_client_main.c`

- `/opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_client/comch_ctrl_path_client_sample.c`
- `/opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_server/comch_ctrl_path_server_main.c`
- `/opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_server/comch_ctrl_path_server_sample.c`
- `/opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_common.c`
- `/opt/mellanox/doca/samples/doca_comch/comch_ctrl_path_common.h`

5.4.6.10.2.2 DOCA Comch Data Path Client/Server

> ⚠ `doca_comch_data_path_high_speed_server` should be run on the BlueField Arm cores and should be started before `doca_comch_data_path_high_speed_client` is started on the host.

This sample sets up a client server connection between host and BlueField Arm.

The connection is used to create a producer and consumer on both sides and pass a message across the two fastpath connections.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the core DOCA structures.
3. Initializing and configuring client/server contexts.
4. Initializing and configuring producer/consumer contexts on top of an established connection.
5. Submitting post receive tasks for population by producers.
6. Submitting send tasks from producers to write to consumers.
7. Stopping and destroying producer/consumer objects.
8. Stopping and destroying client/server objects.

References:

- `/opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speed_client/comch_data_path_high_speed_client_main.c`
- `/opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speedclient/comch_data_path_high_speed_client_sample.c`
- `/opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speedserver/comch_data_path_high_speed_server_main.c`
- `/opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speedserver/comch_data_path_high_speed_server_sample.c`
- `/opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speed_common.c`
- `/opt/mellanox/doca/samples/doca_comch/comch_data_path_high_speed_common.h`

5.4.7 DOCA UROM

This guide provides an overview and configuration instructions for DOCA Unified Resources and Offload Manager (UROM) API.

5.4.7.1 Introduction

> ⚠ The quality status of DOCA libraries is listed here.

The DOCA Unified Resource and Offload Manager (UROM) offers a framework for offloading a portion of parallel computing tasks, such as those related to HPC or AI workloads and frameworks, from the host to the NVIDIA DPUs. This framework includes the UROM service which is responsible for resource discovery, coordination between the host and DPU, and the management of UROM workers that execute parallel computing tasks.

When an application utilizes the UROM framework for offloading, it consists of two main components: the host part and the UROM worker on the DPU. The host part is responsible for interacting with the DOCA UROM API and operates as part of the application with the aim of offloading tasks to the DPU. This component establishes a connection with the UROM service and initiates an offload request. In response to the offload request, the UROM service provides network identifiers for the workers, which are spawned by the UROM service. If the UROM service is running as a Kubernetes POD, the workers are spawned within the POD. Each worker is responsible for executing either a single offload or multiple offloads, depending on the requirements of the host application.

5.4.7.2 Prerequisites

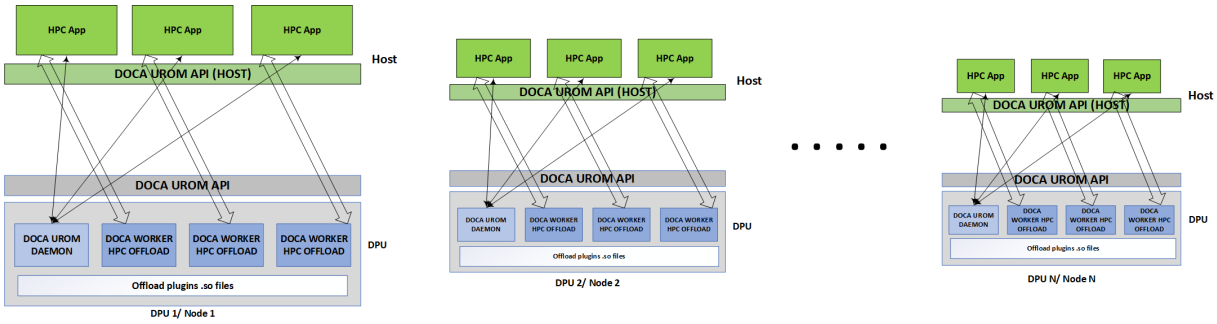
UCX is required for the communication channel between the host and DPU parts of DOCA UROM based on TCP socket transport. This is a mechanism to transfer commands from the host to the UROM service on the DPU and receive responses from the DPU.

By default, UCX scans all available devices on the machine and selects the best ones based on performance characteristics. The environment variable `UCX_NET_DEVICES=<dev1>,<dev2>,...` would restrict UCX to using only the specified devices. For example, `UCX_NET_DEVICES=eth2` uses the Ethernet device **`eth2`** for TCP socket transport.

For more information about UCX, refer to DOCA UCX Programming Guide.

5.4.7.3 Architecture

5.4.7.3.1 UROM Deployment



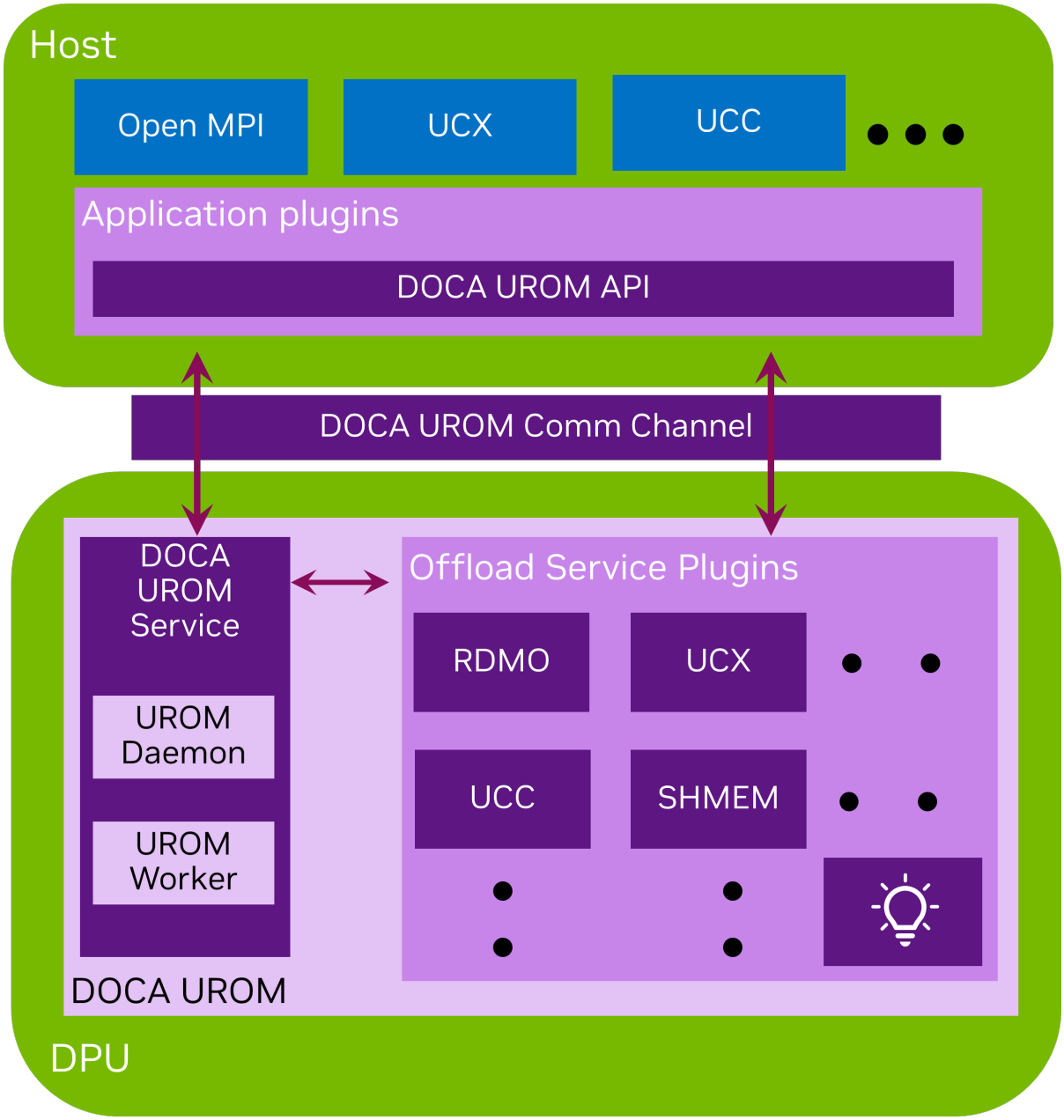
The diagram illustrates a standard UROM deployment where each DPU is required to host both a service process instance and a group of worker processes.

The typical usage of UROM services involves the following steps:

1. Every process in the parallel application discovers the UROM service.
2. UROM handles authentication and provides service details.
3. The host application receives the available offloading plugins on the local DPU through UROM service.
4. The host application picks the desired plugin info and triggers UROM worker plugin instances on the DPU through the UROM service.
5. The application delegates specific tasks to the UROM workers.
6. UROM workers execute these tasks and return the results.

5.4.7.3.2 UROM Framework

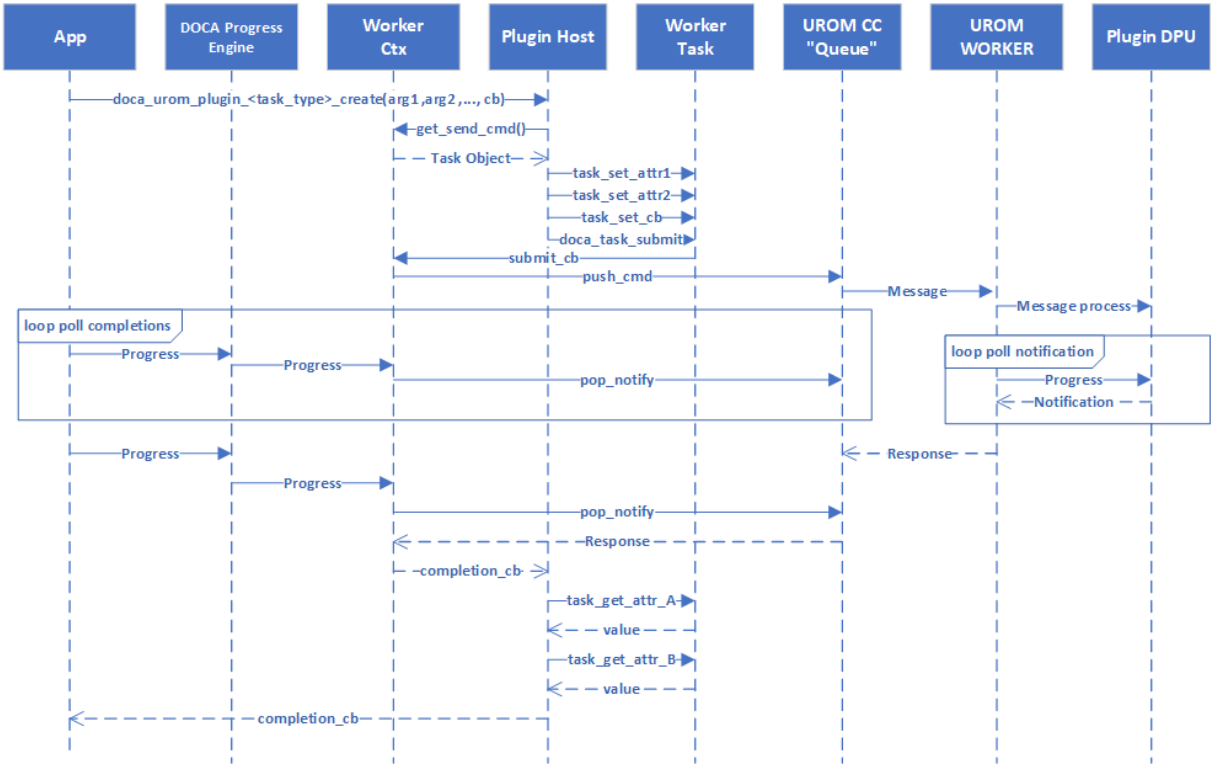
This diagram shows a high-level overview of the DOCA UROM framework.



A UROM offload plugin is where developers of AI/HPC offloads implement their own offloading logic while using DOCA UROM as the transport layer and resource manager. Each plugin defines commands to execute logic on the DPU and notifications that are returned to the host application. Each type of supported offload corresponds to a distinct type of DOCA UROM plugin. For example, a developer may need a UCC plugin to offload UCC functionality to the DPU. Each plugin implements a DPU-side plugin API and exposes a corresponding host-side interface.

A UROM daemon loads the plugin DPU version (`.so` file) in runtime as part of the discovery of local plugins.

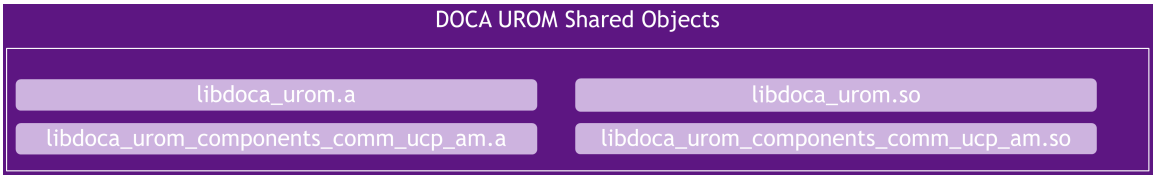
5.4.7.3.2.1 Plugin Task Offloading Flow



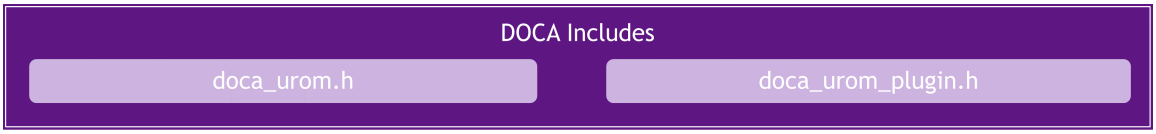
5.4.7.3.3 UROM Installation

DOCA UROM is an integral part of the DOCA SDK installation package. Depending on your system architecture and enabled offload plugins, UROM is comprised by several components, which can be categorized into two main parts: those on the host and those on the DPU.

- DOCA UROM library components:
  - **libdoca_urom** shared object – contains the DOCA UROM API
  - **libdoca_urom_components_comm_ucp_am** – includes the UROM communication channel interface API



- DOCA UROM headers:

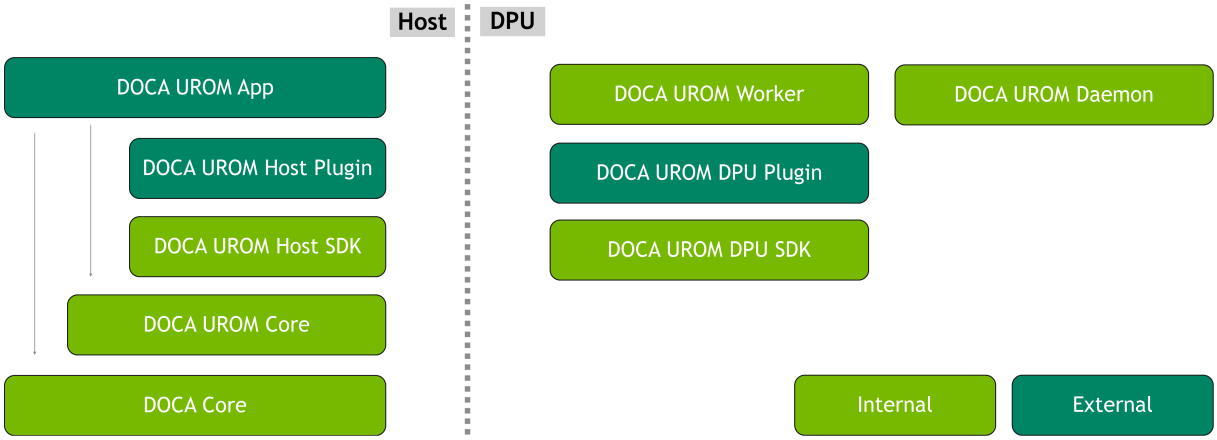


The header files include definitions for DOCA UROM as described in the following:

- DOCA UROM host interface (**doca_urom.h**) – this header includes three essential components: contexts, tasks, and plugins.
  - Service context (**doca_urom_service**) – this context serves as an abstraction of the UROM service process. Tasks posted within this context include the authentication, spawning, and termination of workers on the DPU.
  - Worker context (**doca_urom_worker**) – this context abstracts the DPU UROM worker, which operates on behalf of host application plugins (offload). Tasks posted within this context involve relaying commands from the host application to the worker on behalf of a specific offload plugin, such as offloaded functionality for communication operations.
  - Domain context (**doca_urom_domain**) – this context encapsulates a group of workers belonging to the same host application. This concept is similar to the MPI (message passing interface) communicator in the MPI programming model or PyTorch's process groups. Plugins are not required to use the UROM Domain.
- DOCA UROM plugin interface (**doca_urom_plugin.h**) – this header includes the main structure and definitions that the user can use to build both the host and DPU components of their own offloading plugins
  - UROM plugin interface structure (**urom_plugin_iface**) – this interface includes a set of operations to be executed by the UROM worker
  - UROM worker command structure (**urom_worker_cmd**) – this structure defines the worker instance command format
  - UROM worker notification structure (**urom_worker_notify**) – this structure defines the worker instance notification format

The following diagram shows various software components of DOCA UROM:

- DOCA Core – involves DOCA device discovery, DOCA progress engine, DOCA context, etc.
- DOCA UROM Core – includes the UROM library functionality
- DOCA UROM Host SDK – UROM API for the host application to use
- DOCA UROM DPU SDK – UROM API for the NVIDIA® BlueField® networking platform (DPU or SuperNIC) to use
- DOCA UROM Host Plugin – user plugin host version
- DOCA UROM DPU Plugin – user plugin DPU version
- DOCA UROM App – user UROM host application
- DOCA UROM Worker – the offload functionality component that executes the offloading logic
- DOCA UROM Daemon – is responsible for resource discovery, coordination between the host and DPU, managing the workers on BlueField



5.4.7.4 API

More information is available on DOCA UROM API in the DOCA API References.

The pkg-config (***.pc** file) for the UROM library is **doca-urom**.

The following sections provide additional details about the library API.

5.4.7.4.1 DOCA_UROM_SERVICE_FILE

This environment variable sets the path to the UROM service file. When creating the UROM service object (see **doca_urom_service_create**), UROM performs a look-up using this file, the hostname where an application is running, and the PCIe address of the associated DOCA device to identify the network address, and network devices associated with the UROM service.

This file contains one entry per line describing the location of each UROM service that may be used by UROM. The format of each line must be as follows:

```
<app_hostname> <service_type> <dev_hostname> <dev_pci_addr> <net,devs>
```

Example:

```
app_host1 dpu dpu_host1 03:00.0 dev1:1,dev2:1
```

Fields are described in the following table:

Field	Description
app_hostname	Network hostname (or IP address) for the node that this line applies to

Field	Description
service_type	The UROM service type. Valid type is **dpu** (used for all DOCA devices).
dev_hostname	Network hostname (or IP address) for the associated DOCA device
dev_pci_addr	PCIe address of the associated DOCA device. This must match the PCIe address provided by DOCA.
net,devs	Comma-separated list of network devices shared between the host and DOCA device

5.4.7.4.2 doca_urom_service

An opaque structure that represents a DOCA UROM service.

```
struct doca_urom_service;
```

5.4.7.4.3 doca_urom_service_plugin_info

DOCA UROM plugin info structure. UROM generates this structure for each plugin on the local DPU where the UROM service is running and the service returns an array of available plugins to the host application to pick which plugins to use.

```
struct doca_urom_service_plugin_info {
    uint64_t id;
    uint64_t version;
    char plugin_name[DOCA_UROM_PLUGIN_NAME_MAX_LEN];
};
```

- **id** – Unique ID to send commands to the plugin, UROM generates this ID
- **version** – Plugin DPU version to verify that the plugin host interface has the same version
- **plugin_name** – The **.so** plugin file name without "**.so**". The name is used to find the desired plugin.

5.4.7.4.4 doca_urom_service_get_workers_by_gid_task

An opaque structure representing a DOCA service gets workers by group ID task.

```
struct doca_urom_service_get_workers_by_gid_task;
```

5.4.7.4.5 doca_urom_service_create

Before performing any UROM service operation (spawn worker, destroy worker, etc.), it is essential to create a **doca_urom_service** object. A service object is created in state **DOCA_CTX_STATE_IDLE**. After creation, the user may configure the service using setter methods (e.g., **doca_urom_service_set_dev()**).

Before use, a service object must be transitioned to state **DOCA_CTX_STATE_RUNNING** using the **doca_ctx_start()** interface. A typical invocation looks like **doca_ctx_start(doca_urom_service_as_ctx(service_ctx))**.

```
doca_error_t doca_urom_service_create(struct doca_urom_service **service_ctx);
```

- **service_ctx [in/out]** – **doca_urom_service** object to be created
- Returns – **DOCA_SUCCESS** on success, error code otherwise

Multiple application processes could create different service objects that represent/connect to the same worker on the DPU.

5.4.7.4.6 doca_urom_service_destroy

Destroy a **doca_urom_service** object.

```
doca_error_t doca_urom_service_destroy(struct doca_urom_service *service_ctx);
```

- **service_ctx[in]** – **doca_urom_service** object to be destroyed. It is created by **doca_urom_service_create()**.
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.7 doca_urom_service_set_max_comm_msg_size

Set the maximum size for a message in the UROM communication channel. The default message size is 4096B.

It is important to ensure that the combined size of the plugins' commands and notifications and the UROM structure's size do not exceed this maximum size.

Once the service state is running, users cannot update the maximum size for the message.

```
doca_error_t doca_urom_service_set_max_comm_msg_size(struct doca_urom_service *service_ctx, size_t msg_size);
```

- **service_ctx[in]** – a pointer to **doca_urom_service** object to set new message size
- **msg_size[in]** – new message size to set
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.8 doca_urom_service_as_ctx

Convert a **doca_urom_service** object into a DOCA object.

```
struct doca_ctx *doca_urom_service_as_ctx(struct doca_urom_service *service_ctx);
```

- **service_ctx[in]** – a pointer to **doca_urom_service** object
- Returns – a pointer to the **doca_ctx** object on success, **NULL** otherwise

5.4.7.4.9 doca_urom_service_get_plugins_list

Retrieve the list of supported plugins on the UROM service.

```
doca_error_t doca_urom_service_get_plugins_list(struct doca_urom_service *service_ctx, const struct
doca_urom_service_plugin_info **plugins, size_t *plugins_count);
```

- **service_ctx[in]** – a pointer to **doca_urom_service** object
- **plugins[out]** – an array of pointers to **doca_urom_service_plugin_info** object
- **plugins_count[out]** – number of plugins
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.10 doca_urom_service_get_cpuset

Get the allowed CPU set for the UROM service on BlueField, which can be used when spawning workers to set processor affinity.

```
doca_error_t doca_urom_service_get_cpuset(struct doca_urom_service *service_ctx, doca_cpu_set_t *cpuset);
```

- **service_ctx[in]** – a pointer to **doca_urom_service** object
- **cpuset[out]** – set of allowed CPUs
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.11 doca_urom_service_get_workers_by_gid_task_allocate_init

Allocate a get-workers-by-GID service task and set task attributes.

```
doca_error_t doca_urom_service_get_workers_by_gid_task_allocate_init(struct doca_urom_service *service_ctx,
                                                                     uint32_t gid,
doca_urom_service_get_workers_by_gid_task_completion_cb_t cb,
                                                                     struct
doca_urom_service_get_workers_by_gid_task **task);
```

- **service_ctx[in]** – a pointer to **doca_urom_service** object
- **gid[in]** – group ID to set
- **cb[in]** – user task completion callback
- **task[out]** – a new get-workers-by-GID service task
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.12 doca_urom_service_get_workers_by_gid_task_release

Release a get-workers-by-GID service task and task resources.

```
doca_error_t doca_urom_service_get_workers_by_gid_task_release(struct doca_urom_service_get_workers_by_gid_task
*task);
```

- **task[in]** – service task to release
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.13 doca_urom_service_get_workers_by_gid_task_as_task

Convert a **doca_urom_service_get_workers_by_gid_task** object into a DOCA task object.

After creating a service task and configuring it using setter methods (e.g., **doca_urom_service_get_workers_by_gid_task_set_gid()**) or as part of task allocation, the user should submit the task by calling **doca_task_submit** .

A typical invocation looks like **doca_task_submit(doca_urom_service_get_workers_by_gid_task_as_task(task))** .

```
struct doca_task *doca_urom_service_get_workers_by_gid_task_as_task(struct
doca_urom_service_get_workers_by_gid_task *task);
```

- **task[in]** – get-workers-by-GID service task
- Returns – a pointer to the **doca_task** object on success, **NULL** otherwise

5.4.7.4.14 doca_urom_service_get_workers_by_gid_task_get_workers_count

Get the number of workers returned for the requested GID.

```
size_t doca_urom_service_get_workers_by_gid_task_get_workers_count(struct doca_urom_service_get_workers_by_gid_task
*task);
```

- **task[in]** – get-workers-by-GID service task
- Returns – workers ID's array size

5.4.7.4.15 doca_urom_service_get_workers_by_gid_task_get_worker_ids

Get service get workers task IDs array.

```
const uint64_t *doca_urom_service_get_workers_by_gid_task_get_worker_ids(struct
doca_urom_service_get_workers_by_gid_task *task);
```

- **task[in]** – get-workers-by-GID service task
- Returns – workers ID's array, **NULL** otherwise

5.4.7.4.16 doca_urom_worker

An opaque structure representing a DOCA UROM worker context.

```
struct doca_urom_worker;
```

5.4.7.4.17 doca_urom_worker_cmd_task

An opaque structure representing a DOCA UROM worker command task context.

```
struct doca_urom_worker_cmd_task;
```

5.4.7.4.18 doca_urom_worker_cmd_task_completion_cb_t

A worker command task completion callback type. It is called once the worker task is completed.

```
typedef void (*doca_urom_worker_cmd_task_completion_cb_t)(struct doca_urom_worker_cmd_task *task,
                                                          union doca_data task_user_data,
                                                          union doca_data ctx_user_data);
```

- **task[in]** – a pointer to worker command task
- **task_user_data[in]** – user task data
- **ctx_user_data[in]** – user worker context data

5.4.7.4.19 doca_urom_worker_create

This method creates a UROM worker context.

A worker is created in a **DOCA_CTX_STATE_IDLE** state. After creation, a user may configure the worker using setter methods (e.g., **doca_urom_worker_set_service()**). Before use, a worker must be transitioned to state **DOCA_CTX_STATE_RUNNING** using the **doca_ctx_start()** interface. A typical invocation looks like **doca_ctx_start(doca_urom_worker_as_ctx(worker_ctx))** .

```
doca_error_t doca_urom_worker_create(struct doca_urom_worker **worker_ctx);
```

- **worker_ctx [in/out]** – **doca_urom_worker** object to be created
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.20 doca_urom_worker_destroy

Destroys a UROM worker context.

```
doca_error_t doca_urom_worker_destroy(struct doca_urom_worker *worker_ctx);
```

- **worker_ctx [in]** – **doca_urom_worker** object to be destroyed. It is created by **doca_urom_worker_create()** .
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.21 doca_urom_worker_set_service

Attaches a UROM service to the worker context. The worker is launched on the DOCA device managed by the provided service context.

```
doca_error_t doca_urom_worker_set_service(struct doca_urom_worker *worker_ctx, struct doca_urom_service
*service_ctx);
```

- **service_ctx [in]** – service context to set
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.22 doca_urom_worker_set_id

This method sets the worker context ID to be used to identify the worker. Worker IDs enable an application to establish multiple connections to the same worker process running on a DOCA device.

Worker ID must be unique to a UROM service.

- If **DOCA_UROM_WORKER_ID_ANY** is specified, the service assigns a unique ID for the newly created worker.
- If a specific ID is used, the service looks for an existing worker with matching ID. If one exists, the service establishes a new connection to the existing worker. If a matching worker does not exist, a new worker is created with the specified worker ID.

```
doca_error_t doca_urom_worker_set_id(struct doca_urom_worker *worker_ctx, uint64_t worker_id);
```

- **worker_ctx [in]** – **doca_urom_worker** object
- **worker_id [in]** – worker ID
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.23 doca_urom_worker_set_gid

Set worker group ID. This ID must be set before starting the worker context.

Through service get workers by GID task, the application can have the list of workers' IDs which are running on DOCA device and that belong to the same group ID.

```
doca_error_t doca_urom_worker_set_gid(struct doca_urom_worker *worker_ctx, uint32_t gid);
```

- **worker_ctx [in]** – **doca_urom_worker** object
- **gid [in]** – worker group ID
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.24 doca_urom_worker_set_plugins

Adds a plugin mask for the supported plugins by the UROM worker on the DPU. The application can use up to 62 plugins.

```
doca_error_t doca_urom_worker_set_plugins(struct doca_urom_worker *worker_ctx, uint64_t plugins);
```

- **worker_ctx[in]** – **doca_urom_worker** object
- **plugins[in]** – an ORing set of worker plugin IDs
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.25 doca_urom_worker_set_env

Set worker environment variables when spawning worker on DPU side by DOCA UROM service. They must be set before starting the worker context.

This call fails if the worker already spawned on the DPU.

```
doca_error_t doca_urom_worker_set_env(struct doca_urom_worker *worker_ctx, char *const env[], size_t count);
```

- **worker_ctx [in]** – **doca_urom_worker** object
- **env [in]** – an array of environment variables
- **count [in]** – array size
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.26 doca_urom_worker_as_ctx

Convert a **doca_urom_worker** object into a DOCA object.

```
struct doca_ctx *doca_urom_worker_as_ctx(struct doca_urom_worker *worker_ctx);
```

- **worker_ctx[in]** – a pointer to **doca_urom_worker** object
- Returns – a pointer to the **doca_ctx** object on success, **NULL** otherwise

5.4.7.4.27 doca_urom_worker_cmd_task_allocate_init

Allocate worker command task and set task attributes.

```
doca_error_t doca_urom_worker_cmd_task_allocate_init(struct doca_urom_worker *worker_ctx, uint64_t plugin, struct
doca_urom_worker_cmd_task **task);
```

- **worker_ctx [in]** – a pointer to **doca_urom_worker** object
- **plugin [in]** – task plugin ID
- **task [out]** – set worker command new task
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.28 doca_urom_worker_cmd_task_release

Release worker command task.

```
doca_error_t doca_urom_worker_cmd_task_release(struct doca_urom_worker_cmd_task *task);
```

- **task[in]** – worker task to release
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.29 doca_urom_worker_cmd_task_set_plugin

Set worker command task plugin ID. The plugin ID is created by the UROM service and the plugin host interface should hold it to create UROM worker command tasks.

```
void doca_urom_worker_cmd_task_set_plugin(struct doca_urom_worker_cmd_task *task, uint64_t plugin);
```

- **task [in]** – worker task

- **plugin [in]** – task plugin to set

5.4.7.4.30 doca_urom_worker_cmd_task_set_cb

Set worker command task completion callback.

```
void doca_urom_worker_cmd_task_set_cb(struct doca_urom_worker_cmd_task *task,
doca_urom_worker_cmd_task_completion_cb_t cb);
```

- **task[in]** – worker task
- **plugin[in]** – task callback to set

5.4.7.4.31 doca_urom_worker_cmd_task_get_payload

Get worker command task payload. The plugin interface populates this buffer by plugin command structure. The payload size is the maximum message size in the DOCA UROM communication channel (the user can configure the size by calling **doca_urom_service_set_max_comm_msg_size()**). To update the payload buffer, the user should call **doca_buf_set_data()**.

```
struct doca_buf *doca_urom_worker_cmd_task_get_payload(struct doca_urom_worker_cmd_task *task);
```

- **task [in]** – worker task
- Returns – a **doca_buf** that represents the task's payload

5.4.7.4.32 doca_urom_worker_cmd_task_get_response

Get worker command task response. To get the response's buffer, the user should call **doca_buf_get_data()**.

```
struct doca_buf *doca_urom_worker_cmd_task_get_response(struct doca_urom_worker_cmd_task *task);
```

- **task [in]** – worker task
- Returns – a **doca_buf** that represents the task's response

5.4.7.4.33 doca_urom_worker_cmd_task_get_user_data

Get worker command user data to populate. The data refers to the reserved data inside the task that the user can get when calling the completion callback. The maximum data size is 32 bytes.

```
void *doca_urom_worker_cmd_task_get_user_data(struct doca_urom_worker_cmd_task *task);
```

- **task [in]** – worker task
- Returns – a pointer to user data memory

5.4.7.4.34 doca_urom_worker_cmd_task_as_task

Convert a **doca_urom_worker_cmd_task** object into a DOCA task object.

After creating a worker command task and configuring it using setter methods (e.g., **doca_urom_worker_cmd_task_set_plugin()**) or as part of task allocation, the user should submit the task by calling **doca_task_submit** .

A typical invocation looks like **doca_task_submit(doca_urom_worker_cmd_task_as_task(task))** .

```
struct doca_task *doca_urom_worker_cmd_task_as_task(struct doca_urom_worker_cmd_task *task);
```

- **task[in]** – worker command task
- Returns – a pointer to the **doca_task** object on success, **NULL** otherwise

5.4.7.4.35 doca_urom_domain

An opaque structure representing a DOCA UROM domain context.

```
struct doca_urom_domain;
```

5.4.7.4.36 doca_urom_domain_allgather_cb_t

A callback for a non-blocking all-gather operation.

```
typedef doca_error_t (*doca_urom_domain_allgather_cb_t)(void *sbuf, void *rbuf, size_t msglen, void *coll_info,
void **req);
```

- **sbuf [in]** – local buffer to send to other processes
- **rbuf [in]** – global buffer to include other process's source buffer
- **msglen [in]** – source buffer length
- **coll_info [in]** – collection info
- **req [in]** – allgather request data
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.37 doca_urom_domain_req_test_cb_t

A callback to test the status of a non-blocking allgather request.

```
typedef doca_error_t (*doca_urom_domain_req_test_cb_t)(void *req);
```

- **req [in]** – allgather request data to check status
- Returns – **DOCA_SUCCESS** on success, **DOCA_ERROR_IN_PROGRESS** otherwise

5.4.7.4.38 doca_urom_domain_req_free_cb_t

A callback to free a non-blocking allgather request.

```
typedef doca_error_t (*doca_urom_domain_req_free_cb_t)(void *req);
```

- **req [in]** – allgather request data to release.
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.39 doca_urom_domain_oob_coll

Out-of-band communication descriptor for domain creation.

```
struct doca_urom_domain_oob_coll {
    doca_urom_domain_allgather_cb_t allgather;
    doca_urom_domain_req_test_cb_t req_test;
    doca_urom_domain_req_free_cb_t req_free;
    void *coll_info;
    uint32_t n_oob_indexes;
    uint32_t oob_index;
};
```

- **allgather** – non-blocking allgather callback
- **req_test** – request test callback
- **req_free** – request free callback
- **coll_info** – context or metadata required by the OOB collective
- **n_oob_indexes** – number of endpoints participating in the OOB operation (e.g., number of client processes representing domain workers)
- **oob_index** – an integer value that represents the position of the calling processes in the given OOB operation. The data specified by **src_buf** is placed at the offset "**oob_index** *size" in the **recv_buf**.

  > ⚠ **oob_index** must be unique at every calling process and should be in the range [0: **n_oob_indexes**).

5.4.7.4.40 doca_urom_domain_create

Creates a UROM domain context. A domain is created in state **DOCA_CTX_STATE_IDLE**. After creation, a user may configure the domain using setter methods (e.g., **doca_urom_domain_set_workers()**). Before use, a domain must be transitioned to state **DOCA_CTX_STATE_RUNNING** using the **doca_ctx_start()** interface. A typical invocation looks like **doca_ctx_start(doca_urom_domain_as_ctx(worker_ctx))**.

```
doca_error_t doca_urom_domain_create(struct doca_urom_domain **domain_ctx);
```

- **domain_ctx [in/out]** – **doca_urom_domain** object to be created
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.41 doca_urom_domain_destroy

Destroys a UROM domain context.

```
doca_error_t doca_urom_domain_destroy(struct doca_urom_domain *domain_ctx);
```

- **domain_ctx [in]** – **doca_urom_domain** object to be destroyed; it is created by **doca_urom_domain_create()**
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.42 doca_urom_domain_set_workers

Sets the list of workers in the domain.

```
doca_error_t doca_urom_domain_set_workers(struct doca_urom_domain *domain_ctx, uint64_t *domain_worker_ids, struct
doca_urom_worker **workers, size_t workers_cnt);
```

- **domain_ctx [in]** – **doca_urom_domain** object
- **domain_worker_ids [in]** – list of domain worker IDs
- **workers [in]** – an array of UROM worker contexts that should be part of the domain
- **workers_cnt [in]** – the number of workers in the given array
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.43 doca_urom_domain_add_buffer

Attaches local buffer attributes to the domain. It should be called after calling **doca_urom_domain_set_buffers_count()** .

The local buffer will be shared with all workers belonging to the domain.

```
doca_error_t doca_urom_domain_add_buffer(struct doca_urom_domain *domain_ctx, void *buffer, size_t buf_len, void
 *memh, size_t memh_len, void *mkey, size_t mkey_len);
```

- **domain_ctx [in]** – **doca_urom_domain** object
- **buffer [in]** – buffer ready for remote access which is given to the domain
- **buf_len [in]** – buffer length
- **memh [in]** – memory handle for the exported buffer. (should be packed)
- **memh_len [in]** – memory handle size
- **mkey [in]** – memory key for the exported buffer. (should be packed)
- **mkey_len [in]** – memory key size
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.44 doca_urom_domain_set_oob

Sets OOB communication info to be used for domain initialization.

```
doca_error_t doca_urom_domain_set_oob(struct doca_urom_domain *domain_ctx, struct doca_urom_domain_oob_coll *oob);
```

- **domain_ctx [in]** – **doca_urom_domain** object
- **oob [in]** – OOB communication info to set
- Returns – **DOCA_SUCCESS** on success, error code otherwise

5.4.7.4.45 doca_urom_domain_as_ctx

Convert a **doca_urom_domain** object into a DOCA object.

```
struct doca_ctx *doca_urom_domain_as_ctx(struct doca_urom_domain *domain_ctx);
```

- **domain_ctx[in]** – a pointer to **doca_urom_domain** object
- Returns – a pointer to the **doca_ctx** object on success, **NULL** otherwise

5.4.7.5 Execution Model

DOCA UROM uses the DOCA Core Progress Engine as an execution model for service and worker contexts and tasks progress. For more details about it please refer to this guide.

5.4.7.6 UROM Building Blocks

This section explains the general concepts behind the fundamental building blocks to use when creating a DOCA UROM application and offloading functionality.

5.4.7.6.1 Program Flow

5.4.7.6.1.1 DPU

Launching DOCA UROM Service

DOCA UROM service should be run before running the application on the host to offload commands to BlueField. For more information, refer to the DOCA UROM Service Guide.

5.4.7.6.1.2 Host

Initializing UROM Service Context

1. Create service context: Establish a service context within the control plane alongside the progress engine.
2. Set service context attributes: Specific attributes of the service context are configured. The required attribute is **doca_dev** .
3. Start the service context: The service context is initiated by invoking the **doca_ctx_start** function.
    a. Discover BlueField availability: The UROM library identifies the available BlueField device.
    b. Connect to UROM service: The library establishes a connection to the UROM service. The connection process is synchronized, meaning that the host process and the BlueField service process are blocked until the connection is established.
    c. Perform lookup using UROM service file: A lookup operation is executed using the UROM service file. The path to this file should be specified in the **DOCA_UROM_SERVICE_FILE** environment variable. More information can be found in **doca_urom.h** .
4. Switch context state to **DOCA_CTX_STATE_RUNNING** : The context state transitions to **DOCA_CTX_STATE_RUNNING** at this point.
5. Service context waits for worker bootstrap requests: The service context is now in a state where it awaits and handles worker bootstrap requests.

```
/* Create DOCA UROM service instance */
doca_urom_service_create(&service);
```

```
/* Connect service context to DOCA progress engine */
doca_pe_connect_ctx(pe, doca_urom_service_as_ctx(service));

/* Set service attributes */
doca_urom_service_set_max_workers(service, nb_workers)
doca_urom_service_set_dev(service, dev);

/* Start service context */
doca_ctx_start(doca_urom_service_as_ctx(service));

/* Handling workers bootstrap requests */
do {
    doca_pe_progress(pe);
} while (!are_all_workers_started);
```

Picking UROM Worker Offload Functionality

Once the service context state is running, the application can call **doca_urom_service_get_plugins_list()** to get the available plugins on the local BlueField device where the UROM service is running.

The UROM service generates an identifier for each plugin and the application is responsible for forwarding this ID to the host plugin interface for sending commands and receiving notifications by calling **urom_<plugin_name>_init(<plugin_id>, <plugin_version>)**.

```
const char *plugin_name = "worker_rdmo";
struct doca_urom_service *service;
const struct doca_urom_service_plugin_info *plugins, *rdmo_info;

/* Create and Start UROM service context. */
..

/* Get worker plugins list. */
doca_urom_service_get_plugins_list(*service, &plugins, &plugins_count);

/* Check if RDMO plugin exists. */
for (i = 0; i < plugins_count; i++) {
        if (strcmp(plugin_name, plugins[i].plugin_name) == 0) {
            rdmo_info = &plugins[i];
            break;
        }
}

/* Attach RDMO plugin ID and DPU plugin version for compatibility check */
urom_rdmo_init(rdmo_info->id, rdmo_info->version);
```

Initializing UROM Worker Context

1. Create a service context and connect the worker context to DOCA Progress Engine (PE).
2. Set worker context attributes (in the example below worker plugin is RDMO).
3. Start worker context, submitting internally spawns worker requests on the service context.
4. Worker context state changes to **DOCA_CTX_STATE_STARTING** (this process is asynchronous).
5. Wait until the worker context state changes to **DOCA_CTX_STATE_RUNNING**:
    a. When calling **doca_pe_progress**, check for a response from the service context that the spawning worker on BlueField is done.
    b. If the worker is spawned on BlueField, connect to it and change the status to running.

```
const struct doca_urom_service_plugin_info *rdmo_info;

/* Create DOCA UROM worker instance */
doca_urom_worker_create(&worker);

/* Connect worker context to DOCA progress engine */
doca_pe_connect_ctx(pe, doca_urom_worker_as_ctx(worker));

/* Set worker attributes */
doca_urom_worker_set_service(worker, service);
doca_urom_worker_set_id(worker, worker_id);
doca_urom_worker_set_max_inflight_tasks(worker, nb_tasks);
doca_urom_worker_set_plugins(worker, rdmo_info->id);
doca_urom_worker_set_cpuset(worker, cpuset);
```

```
/* Start UROM worker context */
doca_ctx_start(doca_urom_worker_as_ctx(worker));

/* Progress until worker state changes to running or error happened */
do {
    doca_pe_progress(pe);
    result = doca_ctx_get_state(doca_urom_worker_as_ctx(worker), &state);
} while (state == DOCA_CTX_STATE_STARTING);
```

Offloading Plugin Task

Once the worker context state is **DOCA_CTX_STATE_RUNNING**, the worker is ready to execute offload tasks. The example below is for offloading an RDMO command.

1. Prepare RDMO task arguments (e.g., completion callback).
2. Call the task function from the plugin host interface.
3. Poll for completion by calling **doca_pe_progress**.
4. Get completion notification through the user callback.

```
    int ret;
    struct doca_urom_worker *worker;
    struct rdmo_result res = {0};
    union doca_data cookie = {0};
    size_t server_worker_addr_len;
    ucp_address_t *server_worker_addr;

    cookie.ptr = &res;
    res.result = DOCA_SUCCESS;

    ucp_worker_create(*ucp_context, &worker_params, server_ucp_worker);
    ucp_worker_get_address(*server_ucp_worker, &server_worker_addr, &server_worker_addr_len);

    /* Create and submit RDMO client init task */
    urom_rdmo_task_client_init(worker, cookie, 0, server_worker_addr, server_worker_addr_len,
urom_rdmo_client_init_finished);

    /* Wait for completion */
    do {
        ret = doca_pe_progress(pe);
        ucp_worker_progress(*server_ucp_worker);
    } while (ret == 0 && res.result == DOCA_SUCCESS);

    /* Check task result */
    if (res.result != DOCA_SUCCESS)
        DOCA_LOG_ERR("Client init task finished with error");
```

Initializing UROM Domain Context

1. Create a domain context on the control plane PE.
2. Set domain context attributes.
3. Start the domain context by calling **doca_ctx_start**.
    a. Exchange memory descriptors between all workers.
4. Wait until the domain context state is running.

```
/* Create DOCA UROM domain instance */
doca_urom_domain_create(&domain);

/* Connect domain context to DOCA progress engine */
doca_pe_connect_ctx(pe, doca_urom_domain_as_ctx(domain));;

/* Set domain attributes */
doca_urom_domain_set_oob(domain, oob);
doca_urom_domain_set_workers(domain, worker_ids, workers, nb_workers);
doca_urom_domain_set_buffers_count(domain, nb_buffers);
for each buffer:
    doca_urom_domain_add_buffer(domain);

/* Start domain context */
doca_ctx_start(doca_urom_domain_as_ctx(domain));

/* Loop till domain state changes to running */
do {
    doca_pe_progress(pe);
    result = doca_ctx_get_state(doca_urom_domain_as_ctx(domain), &state);
} while (state == DOCA_CTX_STATE_STARTING && result == DOCA_SUCCESS);
```

Destroying UROM Domain Context

1. Request the domain context to stop by calling **doca_ctx_stop** .
2. Clean up resources by destroying the domain context.

```
/* Request domain context stop */
doca_ctx_stop(doca_urom_domain_as_ctx(domain));

/* Destroy domain context */
doca_urom_domain_destroy(domain);
```

Destroying UROM Worker Context

1. Request the worker context stop by calling **doca_ctx_stop** and posting the destroy command on the service context.
2. Wait until completion for the destroy command is received.
    a. Change worker state to idle.
3. Clean up resources.

```
/* Stop worker context */
doca_ctx_stop(doca_urom_worker_as_ctx(worker));

/* Progress till receiving a completion */
do {
    doca_pe_progress(pe);
    doca_ctx_get_state(doca_urom_worker_as_ctx(worker), &state);
} while (state != DOCA_CTX_STATE_IDLE);

/* Destroy worker context */
doca_urom_worker_destroy(worker);
```

Destroying UROM Service Context

1. Wait for the completion of the UROM worker context commands.
2. Once all UROM workers have been successfully destroyed, initiate service context stop by invoking **doca_ctx_stop** .
3. Disconnect from the UROM service.
4. Perform resource cleanup.

```
/* Handling workers teardown requests*/
do {
    doca_pe_progress(pe);
} while (!are_all_workers_exited);

/* Stop service context */
doca_ctx_stop(doca_urom_service_as_ctx(service));

/* Destroy service context */
doca_urom_service_destroy(service);
```

5.4.7.6.2 Plugin Development

5.4.7.6.2.1 Developing Offload Plugin on DPU

1. Implement **struct urom_plugin_iface** methods.
    a. The **open()** method initializes the plugin connection state and may create an endpoint to perform communication with other processes/workers.

```
static doca_error_t urom_worker_rdmo_open(struct urom_worker_ctx *ctx)
{
    ucp_context_h ucp_context;
    ucp_worker_h ucp_worker;
```

```
    struct urom_worker_rdmo *rdmo_worker;

    rdmo_worker = calloc(1, sizeof(*rdmo_worker));
    if (rdmo_worker == NULL)
        return DOCA_ERROR_NO_MEMORY;

    ctx->plugin_ctx = rdmo_worker;

    /* UCX transport layer initialization */
    .
    .
    .

    /* Create UCX worker Endpoint */
    ucp_worker_create(ucp_context, &worker_params, &ucp_worker);
    ucp_worker_get_address(ucp_worker, &rdmo_worker->ucp_data.worker_address, &rdmo_worker-
>ucp_data.ucp_addrlen);

    /* Resources  initialization */
    rdmo_worker->clients = kh_init(client);
    rdmo_worker->eps = kh_init(ep);

    /* Init completions list, UROM worker checks completed requests by calling progress() method */
    ucs_list_head_init(&rdmo_worker->completed_reqs);

    return DOCA_SUCCESS;
}
```

b. The **addr()** method returns the address of the plugin endpoint generated during **open()** if it exists (e.g., UCX endpoint to communicate with other UROM workers).

c. The **worker_cmd()** method is used to parse and start work on incoming commands to the plugin.

```
static doca_error_t urom_worker_rdmo_worker_cmd(struct urom_worker_ctx *ctx, ucs_list_link_t
*cmd_list)
{
    struct urom_worker_rdmo_cmd *rdmo_cmd;
    struct urom_worker_cmd_desc *cmd_desc;
    struct urom_worker_rdmo *rdmo_worker = (struct urom_worker_rdmo *)ctx->plugin_ctx;

    while (!ucs_list_is_empty(cmd_list)) {
        /* Get new RDMO command from the list */
        cmd_desc = ucs_list_extract_head(cmd_list, struct urom_worker_cmd_desc, entry);

        /* Unpack and deserialize RDMO command */
        urom_worker_rdmo_cmd_unpack(&cmd_desc->worker_cmd, cmd_desc->worker_cmd.len, &cmd);

        rdmo_cmd = (struct urom_worker_rdmo_cmd *)cmd->plugin_cmd;
        /* Handle command according to it's type */
        switch (rdmo_cmd->type) {
        case UROM_WORKER_CMD_RDMO_CLIENT_INIT:
            /* Handle RDMO client init command */
            status = urom_worker_rdmo_client_init_cmd(rdmo_worker, cmd_desc);
            break;
        case UROM_WORKER_CMD_RDMO_RQ_CREATE:
            /* Handle RDMO RQ create command */
            status = urom_worker_rdmo_rq_create_cmd(rdmo_worker, cmd_desc);
            break;
         .
         .
         .
        default:
            DOCA_LOG_INFO("Invalid RDMO command type: %lu", rdmo_cmd->type);
            status = DOCA_ERROR_INVALID_VALUE;
            break;
        }
        free(cmd_desc);
        if (status != DOCA_SUCCESS)
            return status;
    }
    return status;
}
```

d. The **progress()** method is used to give CPU time to the plugin code to advance asynchronous tasks.

```
static doca_error_t urom_worker_rdmo_progress(struct urom_worker_ctx *ctx, ucs_list_link_t
*notif_list)
{
    struct urom_worker_notif_desc *nd;
    struct urom_worker_rdmo *rdmo_worker = (struct urom_worker_rdmo *)ctx->plugin_ctx;

    /* RDMO UCP worker progress */
    ucp_worker_progress(rdmo_worker->ucp_data.ucp_worker);

    /* Check if completion list is empty */
    if (ucs_list_is_empty(&rdmo_worker->completed_reqs))
        return DOCA_ERROR_EMPTY;

    /* Pop completed commands from the list */
    while (!ucs_list_is_empty(&rdmo_worker->completed_reqs)) {
```

```
        nd = ucs_list_extract_head(&rdmo_worker->completed_reqs, struct urom_worker_notif_desc,
entry);
        ucs_list_add_tail(notif_list, &nd->entry);
    }
    return DOCA_SUCCESS;
}
```

e. The **notif_pack()** method is used to serialize notifications before they are sent back to the host.

2. Implement and expose the following symbols:
   a. **doca_error_t urom_plugin_get_version(uint64_t *version);**
      Returns a compile-time constant value stored within the **.so** file and is used to verify that the host and DPU plugin versions are compatible.
   b. **doca_error_t urom_plugin_get_iface(struct urom_plugin_iface *iface);**
      Get the **urom_plugin_iface** struct with methods implemented by the plugin.
3. Compile the user plugin as an **.so** file and place it where the UROM service can access it. For more details, refer to section "Plugin Discovery and Reporting" under the NVIDIA DOCA UROM Service Guide.

5.4.7.6.2.2 Creating Plugin Host Task

1. Allocate and init worker command task.
2. Populate payload buffer by task command.
3. Pack and serialize the command.
4. Set user data.
5. Submit the task.

```
doca_error_t urom_rdmo_task_client_init(struct doca_urom_worker *worker_ctx, union doca_data cookie,
uint32_t id, void *addr, uint64_t addr_len, urom_rdmo_client_init_finished cb)
{
    doca_error_t result;
    size_t pack_len = 0;
    struct doca_buf *payload;
    struct doca_urom_worker_cmd_task *task;
    struct doca_rdmo_task_data *task_data;
    struct urom_worker_rdmo_cmd *rdmo_cmd;

    /* Allocate task */
    doca_urom_worker_cmd_task_allocate_init(worker_ctx, rdmo_id, &task);

    /* Get payload buffer */
    payload = doca_urom_worker_cmd_task_get_payload(task);
    doca_buf_get_data(payload, (void **)&rdmo_cmd);
    doca_buf_get_data_len(payload, &pack_len);

    /* Populate commands attributes */
    rdmo_cmd->type = UROM_WORKER_CMD_RDMO_CLIENT_INIT;
    rdmo_cmd->client_init.id = id;
    rdmo_cmd->client_init.addr = addr;
    rdmo_cmd->client_init.addr_len = addr_len;

    /* Pack and serialize the command */
    urom_worker_rdmo_cmd_pack(rdmo_cmd, &pack_len, (void *)rdmo_cmd);

    /* Update payload data size */
    doca_buf_set_data(payload, rdmo_cmd, pack_len);

    /* Set user data */
    task_data = (struct doca_rdmo_task_data *)doca_urom_worker_cmd_task_get_user_data(task);
    task_data->client_init_cb = cb;
    task_data->cookie = cookie;

    /* Set task plugin callback */
    doca_urom_worker_cmd_task_set_cb(task, urom_rdmo_client_init_completed);

    /* Submit task */
    doca_task_submit(doca_urom_worker_cmd_task_as_task(task));

    return DOCA_SUCCESS;
}
```

5.4.7.7 DOCA UROM Samples

This section provides DOCA UROM library sample implementations on top of BlueField.

The samples illustrate how to use the DOCA UROM API to do the following:

- Define and create a UROM plugin host and DPU versions for offloading HPC/AI tasks
- Build host applications that use the plugin to execute jobs on BlueField by the DOCA UROM service and workers

> All the DOCA samples described in this section are governed under the BSD-3 software license agreement. These samples are also available on GitHub.

5.4.7.7.1 Sample Prerequisite

Sample	Type	Prerequisite
Sandbox	Plugin	A plugin which offloads the UCX tagged send/receive API
Graph	Plugin	The plugin uses UCX data structures and UCX endpoint
UROM Ping Pong	Program	The sample uses the Open MPI package as a launcher framework to launch two processes in parallel

5.4.7.7.2 Running the Sample

1. Refer to the following documents:
   - DOCA Installation Guide for Linux for details on how to install BlueField-related software
   - DOCA Troubleshooting for any issue you may encounter with the installation, compilation, or execution of DOCA samples
2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

   ```
   cd /opt/mellanox/doca/samples/doca_urom/<sample_name>
   meson /tmp/build
   ninja -C /tmp/build
   ```

   > The binary **doca_<sample_name>** is created under **/tmp/build/** .

3. UROM Sample arguments:

Sample	Argument	Description
UROM multi-workers bootstrap	**-d, --device <IB device name>**	IB device name
UROM Ping Pong	**-d, --device <IB device name>**	IB device name
	-m, --message	Specify ping pong message

4. For additional information per sample, use the **-h** option:

```
/tmp/build/doca_<sample_name> -h
```

5.4.7.7.3 UROM Plugin Samples

DOCA UROM plugin samples have two components. The first one is the host component which is linked with UROM host programs. The second is the DPU component which is compiled as an **.so** file and is loaded at runtime by the DOCA UROM service (daemon, workers).

To build a given plugin:

```
cd /opt/mellanox/doca/samples/doca_urom/plugins/worker_<plugin_name>
meson /tmp/build
ninja -C /tmp/build
```

The binary **worker_<sample_name>.so** file is created under **/tmp/build/** .

5.4.7.7.3.1 Graph

This plugin provides a simple example for creating a UROM plugin interface. It exposes only a single command loopback, sending a specific value in the command, and expects to receive the same value in the notification from UROM worker.

References:

- **/opt/mellanox/doca/samples/doca_urom/plugins/worker_graph/meson.build**
- **/opt/mellanox/doca/samples/doca_urom/plugins/worker_graph/urom_graph.h**
- **/opt/mellanox/doca/samples/doca_urom/plugins/worker_graph/worker_graph.c**
- **/opt/mellanox/doca/samples/doca_urom/plugins/worker_graph/worker_graph.h**

5.4.7.7.3.2 Sandbox

This plugin provides a set of commands for using the offloaded ping pong communication operation.

References:

- **/opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox/meson.build**
- **/opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox/urom_sandbox.h**
- **/opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox/worker_sandbox.c**
- **/opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox/worker_sandbox.h**

5.4.7.7.4 UROM Program Samples

DOCA UROM program samples can run only on the host side and require at least one DOCA UROM service instance to be running on BlueField.

The environment variable should be set **DOCA_UROM_SERVICE_FILE** to the path to the UROM service file.

5.4.7.7.4.1 UROM Multi-worker Bootstrap

This sample illustrates how to properly initialize DOCA UROM interfaces and use the API to spawn multiple workers on the same application process.

The sample initiates four threads as UROM workers to execute concurrently, alongside the main thread operating as a UROM service. It divides the workers into two groups based on their IDs, with odd-numbered workers in one group and even-numbered workers in the other.

Each worker executes the data loopback command by using the Graph plugin, sends a specific value, and expects to receive the same value in the notification.

The sample logic includes:

1. Opening DOCA IB device.
2. Initializing needed DOCA core structures.
3. Creating and starting UROM service context.
4. Initiating the Graph plugin host interface by attaching the generated plugin ID.
5. Launching 4 threads and for each of them:
    a. Creating and starting UROM worker context.
    b. Once the worker context switches to running, sending the loopback graph command to wait until receiving a notification.
    c. Verifying the received data.
    d. Waiting until an interrupt signal is received.
6. The main thread checking for pending jobs of spawning workers (4 jobs, one per thread).
7. Waiting until an interrupt signal is received.
8. The main thread checking for pending jobs of destroying workers (4 jobs, one per thread) for exiting.
9. Cleaning up and exiting.

References:

- **/opt/mellanox/doca/samples/doca_urom/urom_multi_workers_bootstrap/urom_multi_workers_bootstrap_sample.c**
- **/opt/mellanox/doca/samples/doca_urom/urom_multi_workers_bootstrap/urom_multi_workers_bootstrap_main.c**
- **/opt/mellanox/doca/samples/doca_urom/urom_multi_workers_bootstrap/meson.build**
- **/opt/mellanox/doca/samples/doca_urom/urom_common.c**
- **/opt/mellanox/doca/samples/doca_urom/urom_common.h**

5.4.7.7.4.2 UROM Ping Pong

This sample illustrates how to properly initialize the DOCA UROM interfaces and use its API to create two different workers and run ping pong between them by using Sandbox plugin-based UCX.

The sample is using Open MPI to launch two different processes, one process as server and the second one as client, the flow is decided according to process rank.

The sample logic per process includes:

1. Initializing MPI.

2. Opening DOCA IB device.
3. Creating and starting UROM service context.
4. Initiating the Sandbox plugin host interface by attaching the generated plugin id.
5. Creating and starting UROM worker context.
6. Creating and starting domain context.
7. Through domain context, the sample processes exchange the worker's details to communicate between them on the BlueField side for ping pong flow.
8. Starting ping pong flow between the processes, each process offloading the commands to its worker on the BlueField side.
9. Verifying that ping pong is finished successfully.
10. Destroying the domain context.
11. Destroying the worker context.
12. Destroying the service context.

References:

- **/opt/mellanox/doca/samples/doca_urom/urom_ping_pong/urom_ping_pong_sample.c**
- **/opt/mellanox/doca/samples/doca_urom/urom_ping_pong/urom_ping_pong_main.c**
- **/opt/mellanox/doca/samples/doca_urom/urom_ping_pong/meson.build**
- **/opt/mellanox/doca/samples/doca_urom/urom_common.c**
- **/opt/mellanox/doca/samples/doca_urom/urom_common.h**

5.4.8 DOCA RDMA

This guide provides an overview and configuration instructions for the DOCA RDMA API.

5.4.8.1 1. Introduction

The quality status of DOCA libraries is listed here.

DOCA RDMA (Remote Direct Memory Access) enables applications to directly access the memory of remote machines without involving their CPUs or operating systems. By avoiding CPU interruptions, RDMA significantly reduces context switching for I/O operations, yielding:

- Lower latency
- Higher bandwidth
- Improved application performance

The DOCA RDMA library offers an API to execute a variety of RDMA operations, providing software developers with the tools to optimize their applications using high-performance memory access.

In production environments, ensure RDMA operations are performed over a secure channel. Due to the direct memory access nature of RDMA, unauthorized access can pose significant security risks. Implement appropriate security measures to protect your application and network.

5.4.8.2 2. Prerequisites

This library follows the architecture of a DOCA Core Context, it is recommended read the following sections before proceeding:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

5.4.8.3 3. Changes From Previous Release

N/A

5.4.8.4 4. Environment

DOCA RDMA-based applications can run either on the host machine or on the NVIDIA® BlueField® networking platform (DPU or SuperNIC).

5.4.8.5 5. Architecture

DOCA RDMA is a DOCA Context as defined by DOCA Core. See NVIDIA DOCA Core Context for more information.

DOCA RDMA consists of two connected sides, passing data between one another. This includes the option for one side to access the remote side's memory if the granted permissions allow it.

The connection between the two sides can either be based on InfiniBand (IB) or based on Ethernet using RoCE.

DOCA RDMA leverages the Core architecture to expose asynchronous tasks/events that are offloaded to hardware.

The supported operations that may be executed between the two sides, using DOCA RDMA, are:

- Receive
- Send
- Send with immediate
- Write
- Write with immediate
- Read
- Atomic compare and swap
- Atomic fetch and add
- Get remote DOCA Sync Event
- Set remote DOCA Sync Event
- Add remote DOCA Sync Event

5.4.8.5.1 5.1. Objects

5.4.8.5.1.1 5.1.1. Device

The RDMA library requires a DOCA device to operate. This device is used to utilize the connection between the peers in RDMA, access memory, and perform the different operations.

The device must stay valid until the RDMA instance is destroyed.

5.4.8.5.1.2 5.1.2. Memory Map

Executing any DOCA RDMA operation in which data is passed between the peers requires creating a memory map (mmap) on each side.

- The mmap's permissions must include the relevant RDMA permission, according to the required RDMA operations. Tasks fail in case of insufficient permissions.

  Refer to section "Permissions" for more information.

- To allow the peer to execute RDMA operations, the mmap must be exported, using **doca_mmap_export_rdma()**, and passed to the peer (i.e., the side requesting the RDMA operation) where the remote mmap is created and used to access the memory.

5.4.8.5.1.3 5.1.3. Buffer Inventory and Buffers

Executing any DOCA RDMA operation, in which data is passed between the peers, requires using buffers, and thus requires a buffer inventory as well.

Each operation calls for a different set-up for the buffers in use, this is explicitly explained in the "Tasks" section.

5.4.8.6 6. Configuration Phase

To start using the library you need to first go through a configuration phase as described in DOCA Core Context Configuration Phase

This section describes how to configure and start the context, to allow execution of tasks and retrieval of events.

5.4.8.6.1 6.1. Configurations

The context can be configured to match the application use case.

5.4.8.6.1.1 6.1.1. Mandatory Configurations

These configurations are mandatory and must be set by the application before attempting to start the context:

6.1.1.1. Task Configurations

At least one task/event type must be configured. See configuration of Tasks and/or Events.

6.1.1.2. Permissions

Different tasks require different permission to be set for both the RDMA and the mmap in use.

The following table summarizes the necessary RDMA and mmap permissions for each RDMA operation:

DOCA RDMA Task Types	Minimal Permissions				Should Export MMAP?
	The Side Submitting the Task		The Peer		
	RDMA	MMAP	RDMA	MMAP	
Read Get Remote Sync Event	–	Local read write	RDMA read	Local read write | RDMA read	Yes
Write Write with Immediate Set Remote Sync Event	–	Local read write	RDMA write	Local read write | RDMA write	Yes
Atomic Compare and Swap Atomic Fetch and Add Add Remote Sync Event	–	Local read write	RDMA atomic	Local read write | RDMA atomic	Yes
Send Send with Immediate	–	Local read write	–	Local read write	No
Receive	Depending on the received task	Local read write	Not relevant		

We don't have a way to export this macro.

5.4.8.6.1.2 6.1.2. Optional Configurations

If these configurations are not set, a default value is used.

Users may edit the default properties of the RDMA instance using the `doca_rdma_set_<property>()`. The user may also query the default/set properties using `doca_rdma_cap_get_<property>(struct doca_rdma *, ...)` functions.

> The number of tasks that can be submitted in bulk is dependent on the properties `max_send_buf_list_len` and `send_queue_size`.

Refer to section "Library Capability" for querying valid property values when configuring the library context.

> ⚠ While it is possible to select "dynamically connected" (DC) as transport type using `doca_rdma_set_transport_type()`, current support for this transport type is in alpha level only.

5.4.8.6.2 6.2. Device Support

DOCA RDMA requires a device to operate. For picking a device, see DOCA Core Device Discovery.

As device capabilities may change in the future, it is recommended to query each `doca_devinfo` for its capabilities relevant to RDMA operations, using `doca_rdma_cap_*(struct doca_devinfo *, ...)` functions, and check whether the device is suitable for the required RDMA task types, using `doca_rdma_task_<task_type>_is_supported()`.

BlueField-2 and higher devices are supported:

- On the host, any `doca_dev` is supported
- On the BlueField Platform, applications must provide the library with SFs as a `doca_dev`. See OpenvSwitch Acceleration - OVS in DOCA and BlueField DPU Scalable Function to see how to create SFs and connect them to the appropriate ports.

  > ℹ An exception to this is when running RDMA on the DPA datapath, which currently only supports PFs.

5.4.8.6.3 6.3. Buffer Support

The DOCA RDMA library utilizes different buffer types, depending on the task and the buffer's purpose:

- Local mmap buffer
- Mmap from RDMA export buffer
- Mmap from PCIe export buffers

  > ℹ This type of buffer can be used in an equivalent manner to local mmap buffers.

- Linked list buffer

For task-specific information, refer to section "Tasks".

5.4.8.6.4 6.4. Multiple Connections

Multiple parallel connections can be established between peers, enabling an RDMA instance to simultaneously connect to different peers. A unique connection object is provided as to identify and use each established connection.

The maximum number of connection supported by a RDMA instance may be configured using **doca_rdma_set_max_num_connections()**.

To terminate a connection, the connection object must be provided as input to **doca_rdma_connection_disconnect()**.

> ⚠ All established connections are automatically terminated when the context is stopped

> ⚠ All connection methods allow multiple connections. However, the specifics way for providing the connection object depends on the connection methods. See section "Establishing RDMA Connections" for more information

5.4.8.6.5 6.5. Establishing RDMA Connections

To establish the communication between the peers and allow the execution of different DOCA RDMA tasks, the RDMA instances must be connected.

> ⚠ This should be executed after **doca_ctx_start()** is called.

> ℹ Refer to section "State Machine" for more information.

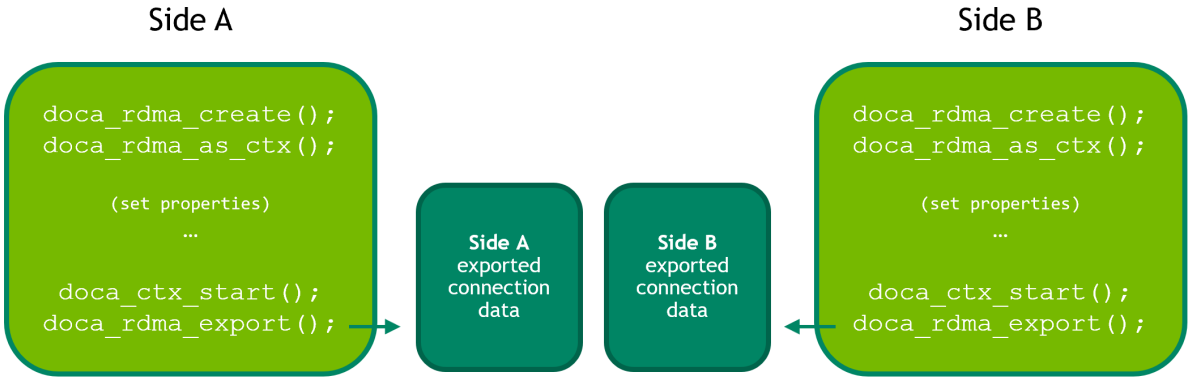
There are three methods for establishing RDMA connections as detailed in the following subsections.

5.4.8.6.5.1 6.5.1. Exporting and Connecting RDMA

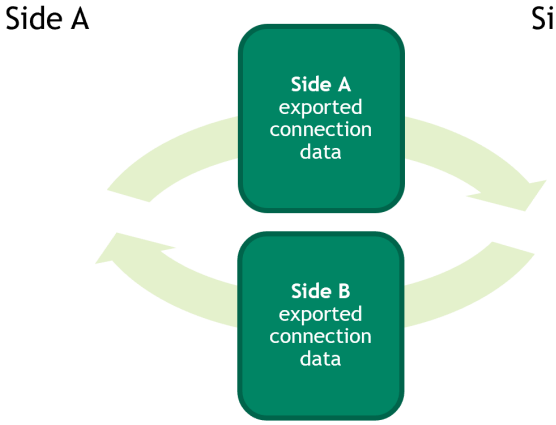
Connecting the RDMA instances can be done by exporting each RDMA instance to the remote side to a blob by using **doca_rdma_export()**, transferring the blob to the opposite side, out-of-band (OOB), and providing it as input to the **doca_rdma_connect()** function on that side.

All in all, the configuration flow should be as presented in the following image:

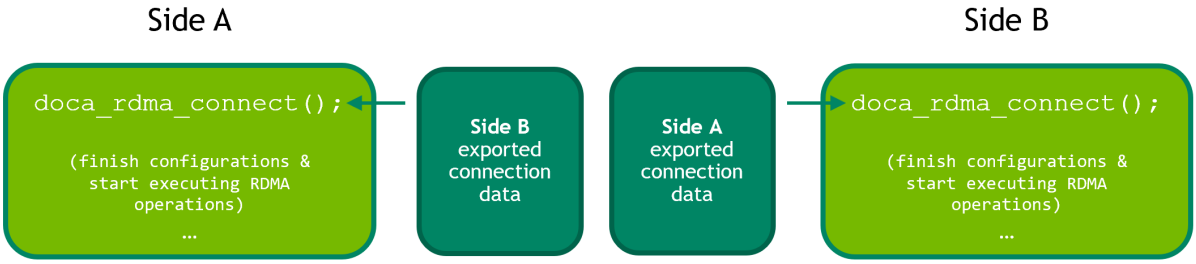
Step 1: Initiate the RDMA instance, and when ready, export it



Step 2: Transfer the exported connection data out-of-band



Step 3: Connect the two RDMA instances



The exported data contains sensitive information. Make sure to pass this data through a secure channel!

The **doca_rdma_export()** returns a unique connection object identifying this connection. This object must be provided as input to **doca_rdma_connect()** (to perform a connection) or to **doca_rdma_connection_disconnect()** (to perform a disconnection), and to all task allocations exchanged over this connection.

An RDMA instance can perform an asymmetric disconnection process using **doca_rdma_connection_disconnect()**.

The RDMA instance which did not initiate the disconnection process will not receive a notification.

5.4.8.6.5.2 6.5.2. Connecting Using RDMA CM Connection Flow

The RDMA CM (communication manager) flow uses the server/client scheme where one of the RDMA instances acts as a server for the second RDMA instance (client). The process for both RDMA instances is non-blocking, event driven, and governed by the progress engine (PE). The connection process is reported to both instances by callbacks which should be set with **doca_rdma_set_connection_state_callbacks()**.

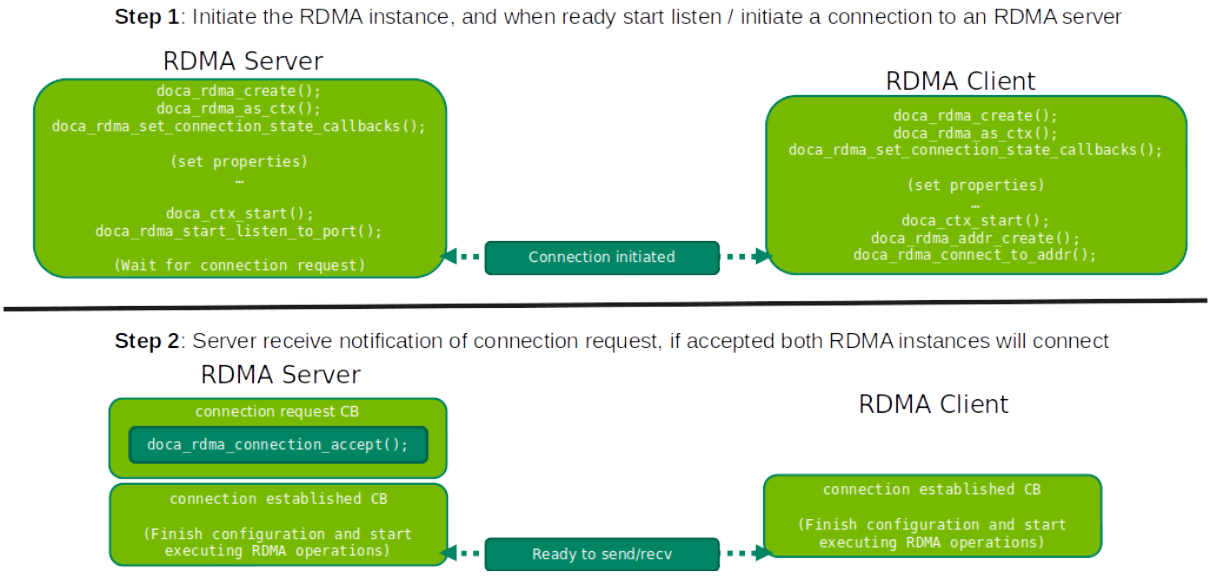
There are four state callbacks:

- Connection request callback – This function is called by **doca_pe_progress()** when a connection request is received by an RDMA instance acting as server
- Connection established callback – This function is called by **doca_pe_progress()** when a connection is successfully established between server/client RDMA instances
- Connection failure callback – This function is called by **doca_pe_progress()** when a connection fails to be established between server/client RDMA instances
- Connection disconnection callback – This function is called by **doca_pe_progress()** when a connection disconnects either server/client RDMA instances

A typical connection flow would be as follows:

1. Prior to initiating a connection, the RDMA instance acting as server (i.e., RDMA server) must start active listening for a connection from a remote RDMA peer (using RDMA CM) to a specific port using **doca_rdma_start_listen_to_port()**. An RDMA server can stop listening for a connection from a remote RDMA peer (using RDMA CM) by using **doca_rdma_stop_listen_to_port()**.
2. The RDMA CM instance acting as client (i.e., RDMA client) can now perform an RDMA connection to the RDMA server. As first step it must create an address object by using **doca_rdma_addr_create()**. The parameters to this function correspond to the RDMA server details required to perform a connection. This object can be destroyed by using **doca_rdma_addr_destroy()**, and retrieve it from a connection with **doca_rdma_connection_get_addr()**.

3. The RDMA client can set the connection user data to include in each connection using **doca_rdma_connection_set_user_data()**, and retrieve it from a connection using **doca_rdma_connection_get_user_data()**.
4. The RDMA client can now perform a connection to the RDMA server using **doca_rdma_connect_to_addr()**. Depending on the network topology and configuration, the connection can be established with IPv4, IPv6, or GID.
5. The RDMA server receives a notification with a connection request through the previously set connection request callback function. The RDMA server can decide to accept the connection with **doca_rdma_connection_accept()** or reject the connection with **doca_rdma_connection_reject()**.
    - If the RDMA server rejects the connection or the connection cannot be successfully established, the RDMA server and RDMA client receive a notification through the connection failure callback function.
    - If the RDMA server accepts the connection and the connection can be successfully established, the RDMA server and RDMA client receive a notification through the connection established callback function.
6. After the RDMA operation is complete, either side can perform the disconnection process using **doca_rdma_connection_disconnect()**. The RDMA instance that did not initiate the disconnection process receives a notification through the disconnection callback function.



The connection process involves resolving the RDMA server connection address. This process is limited by the **connection_request_timeout** property which can be set using **doca_rdma_set_connection_request_timeout()** and retrieved using **doca_rdma_get_connection_request_timeout()**.

⚠ The RDMA server "connection request" callback and the RDMA client "connection established" callback functions provide a connection object unique to the connection. This object must be provided as input to **doca_rdma_connection_accept()**, **doca_rdma_connection_reject()**, and/or **doca_rdma_connection_disconnect()** as well as to all task allocations exchanged over this connection. The same procedure is followed for each separate connection in order to have a unique connection object per connection.

⚠ If a connection fails to be established between server/client RDMA instances, the connection object is valid only in the connection failure callback, and is implicitly destroyed immediately after return.

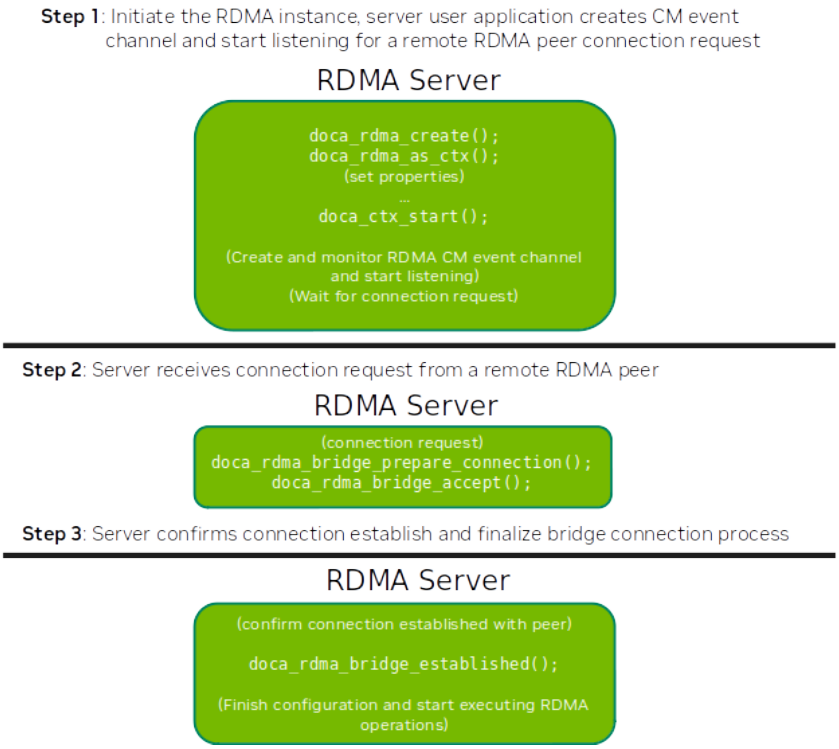
⚠ The **doca_rdma_connection_accept()** method accepts private data as an input parameter that gets transparently passed to the remote side as part of the communication request.

5.4.8.6.5.3 6.5.3. Using Bridge Functions to Accept CM Connection

DOCA RDMA offers connection functionality for user RDMA CM applications acting as a server that maintains a CM event channel and performs the listen process itself (i.e., not using DOCA RDMA connection flow functions).

The functionality must be executed as follows:

1. Server user application, using RDMA CM, must create an RDMA CM event channel, start active listening for a connection from a remote RDMA peer, and monitor the created CM event channel. These functions are performed without the use of the DOCA RDMA connection flow functions explained in section "Connecting Using RDMA CM Connection Flow".
2. Once the server user application received a connection request from a remote RDMA peer acting as client (using RDMA CM), it can call **doca_rdma_bridge_prepare_connection()**. This method acts as a bridge to prepare and perform the doca connection to a connection request, from an application that performs the listen process by itself. The previously explained **doca_rdma_connection_accept()** cannot be used for this connection step as the user application needs to provide the RDMA CM id to prepare the connection.
3. Once the connection is ready, the server can call **doca_rdma_bridge_accept()**. This method acts as a bridge to accept a connection request from an application that performs the listen process by itself.
4. After the server side calls **doca_rdma_bridge_accept()** and confirms the client connection is successfully established, it should call **doca_rdma_bridge_established()** to finish the connection process from the server side. Only after a connection is established can DOCA RDMA tasks be allocated and submitted.



The **doca_rdma_bridge_prepare_connection()** method returns a unique connection object that identifies the connection. This object must be provided as input to **doca_rdma_bridge_accept()** and **doca_rdma_bridge_established()** to finish the connection process from the server side as well as to all task allocations exchanged over this connection. The same procedure is required for each separate connection so as to have a unique connection object per connection.

The **doca_rdma_bridge_accept()** method accepts private data as input parameter that is transparently passed to the remote side as part of the communication request.

5.4.8.7 7. Execution Phase

This section describes execution on CPU using DOCA Core PE. For additional execution environments refer to section "Alternative Datapath Options".

5.4.8.7.1 7.1. Tasks

DOCA RDMA exposes asynchronous tasks that leverage the DPU hardware according to the DOCA Core architecture. See DOCA Core Task.

⚠ Most DOCA RDMA operations are not atomic and therefore it is imperative that the application handle synchronization appropriately. Moreover, successful completion of a write task, with or without immediate, does not guarantee data has been fully written to the remote address.

⚠ All buffers used in DOCA RDMA tasks must remain valid until the task result is retrieved.

5.4.8.7.1.1 7.1.1. Receive Task

This task should be submitted prior to an expected submission of a send/send with immediate/ write with immediate task on the remote side.

7.1.1.1. Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_receive_set_conf`	`doca_rdma_cap_task_receive_is_supported`
Number of tasks	`doca_rdma_task_receive_set_conf`	–
Destination buffer list length	`doca_rdma_task_receive_set_dst_buf_list_len`	`doca_rdma_cap_task_receive_get_max_dst_buf_list_len`

7.1.1.2. Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Destination buffer	Buffer pointing to a local memory address. The data is written to the buffer upon successful completion of the task.	• Linked list buffers are supported • The given destination buffer/list of buffers (given in `dst_buf`) must have a total length sufficient for the expected message size or the task would fail • The destination buffer is not mandatory and may be NULL when the requested DOCA RDMA task on the remote side is "write with immediate" or when the remote side is sending an empty message, with or without immediate (these tasks are presented later on in the "Tasks" section) • For the DOCA RDMA receive task, the length of each buffer is considered as the length from the end of the data section until the end of the buffer, as this is the available memory that can be written to in each buffer. The data length is increased in each buffer if data is written to it once the task is successfully completed.

7.1.1.3. Task Output

Common output as described in DOCA Core Task.

Name	Description	Notes
Result length	The length of data received by the task	Valid only on successful completion of the task

Name	Description	Notes
Result opcode	The opcode of the operation executed by the peer and received by the task	Valid only after task completion, irrespective of success
Result immediate data	The immediate data received by the task	• Valid only on successful completion of the task • Valid only when an immediate value was received (i.e. when the result opcode is `DOCA_RDMA_OPCODE_RECV_SEND_WITH_IMM` or `DOCA_RDMA_OPCODE_RECV_WRITE_WITH_IMM`) – may be retrieved using `doca_rdma_task_receive_get_result_opcode()`)
RDMA connection	The RDMA connection used by the task	Valid only on successful completion callback of the task

7.1.1.4. Task Completion Success

After the task completes successfully, the following happens:

- The received data is copied to the tail segment extending the original data segment
- The data length is increased by the received data length

7.1.1.5. Task Completion Failure

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user
- If a non-fatal error occurs, the task status is updated. Some buffers may be updated and some may remain unchanged.

7.1.1.6. Task Limitations

- The operation is not atomic and therefore it is imperative that the application handle synchronization appropriately
- The destination buffer must remain valid until task is completed
- The total length of the message must not exceed the `max_message_size` device capability
- The buffer list length must not exceed the `dst_buf_list_len` property of the DOCA RDMA receive task
- Other limitations are described in DOCA Core Task

5.4.8.7.1.2 7.1.2. Send Task

This task should be submitted to transfer a message to the remote side, and while the remote side is expecting a message and had submitted a receive task beforehand.

7.1.2.1. Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_send_set_conf`	`doca_rdma_cap_task_send_is_supported`
Number of tasks	`doca_rdma_task_send_set_conf`	–

Description	API to Set the Configuration	API to Query Support
Source buffer list length	`doca_rdma_set_max_send_buf_list_len`	`doca_rdma_cap_get_max_send_buf_list_len`

We don't have a way to export this macro.

7.1.2.2. Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Buffer pointing to a local memory address and holds the data to be sent to the remote peer	• Linked list buffers are supported • The total length of the given source buffer/list of buffers (in `src_buf`) may not exceed the expected message size on the remote side or the task fails • The source buffer is not mandatory and may be NULL when wishing to send an empty message • For the DOCA RDMA send task, the length of each buffer is considered as its data length
RDMA connection	The RDMA connection to use on the task	Connection object provided in the connection establishment process as described in the "Establishing RDMA Connections" section

7.1.2.3. Task Output

Common output as described in DOCA Core Task.

7.1.2.4. Task Completion Success

After the task completes successfully, the following happens:

- On successful completion of the task, the data in the source buffer will be sent to the remote side.
- It doesn't indicate that the data is received by the remote side.

7.1.2.5. Task Completion Failure

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user
- If a non-fatal error occurs, the task status is updated

7.1.2.6. Task Limitations

- The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.
- The source buffer must remain valid until the task completes
- The total length of the message must not exceed the `max_message_size` device capability
- The buffer list length must not exceed the `max_send_buf_list_len` property of the DOCA RDMA instance
- Other limitations are described in DOCA Core Task

5.4.8.7.1.3 7.1.3. Send With Immediate Task

This task should be submitted to transfer a message to the remote side with immediate data (a 32-bit value sent to the remote side, out-of-band), and while the remote side is expecting a message and had submitted a receive task beforehand.

7.1.3.1. Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_send_imm_set_conf`	`doca_rdma_cap_task_send_imm_is_supported`
Number of tasks	`doca_rdma_task_send_imm_set_conf`	–
Source buffer list length	`doca_rdma_set_max_send_buf_list_len`	`doca_rdma_cap_get_max_send_buf_list_len`

We don't have a way to export this macro.

7.1.3.2. Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Buffer pointing to a local memory address and holding the data to be sent to the remote peer	• Linked list buffers are supported. • The total length of the given source buffer/list of buffers (in `src_buf`) may not exceed the expected message size on the remote side or the task fails. • The source buffer is not mandatory and may be NULL when wishing to send an empty message (may be relevant when wishing to keep a connection alive) • For the DOCA RDMA send task, the length of each buffer is considered as its data length
Immediate data	32-bit value sent to the remote side, out-of-band	• The `immediate_data` field should be in Big-Endian format. This value is received by the remote side only once a receive task is completed successfully.
RDMA connection	The RDMA connection to use on the task	Connection object provided in the connection establishment process as described in the "Establishing RDMA Connections" section

7.1.3.3. Task Output

Common output as described in DOCA Core Task.

7.1.3.4. Task Completion Success

After the task completes successfully, the following happens:

- The data in the source buffer is sent to the remote side
- It does not indicate that the data is received by the remote side

7.1.3.5. Task Completion Failure

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user
- If a non-fatal error occurs, the task status is updated

7.1.3.6. Task Limitations

- The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.
- The source buffer must remain valid until the task completes
- The total length of the message must not exceed the `max_message_size` device capability
- The buffer list length must not exceed the `max_send_buf_list_len` property of the DOCA RDMA instance
- Other limitations are described in DOCA Core Task

5.4.8.7.1.4 7.1.4. Read Task

This task should be submitted when wishing to read data from remote memory (i.e., the memory on the remote side of the connection).

7.1.4.1. Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_read_set_conf`	`doca_rdma_cap_task_read_is_supported`
Number of tasks	`doca_rdma_task_read_set_conf`	–
Destination buffer list length	`doca_rdma_set_max_send_buf_list_len`	`doca_rdma_cap_get_max_send_buf_list_len`

We don't have a way to export this macro.

7.1.4.2. Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Points to a remote memory address and holds the data to be read	• Linked list buffers are not supported • The source buffer (`src_buf`) is not mandatory and may be NULL when wishing to read zero bytes (may be relevant when wishing to keep a connection alive) • The data is read only from the data section of the source buffer • The length of the source buffer is considered its data length. The length of data read from the source buffer depends on its data length yet cannot exceed the total length of the given destination buffer/list of buffers. That is, the actual length read depends on the minimal length between the source and destination.
Destination buffer	Points to a local memory address. The data is written to the buffer upon successful completion of the task	• Linked list buffers are supported • The length of each destination buffer is considered as the length from the end of the data section until the end of the buffer, as this is the available memory that can be written to in each buffer • May be NULL if the source buffer has been set to NULL

Name	Description	Notes
RDMA connection	The RDMA connection to use on the task	Connection object provided in the connection establishment process as described in the "Establishing RDMA Connections" section

7.1.4.3. Task Output

Common output as described in DOCA Core Task.

Name	Description	Notes
Result length	The length of data read by the task	Valid only on successful completion of the task

7.1.4.4. Task Completion Success

After the task completes successfully, the following happens:

- The read data is appended after the data section in the destination buffer, as it was prior to the task submission
- The data length is increased by the read data length

7.1.4.5. Task Completion Failure

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user
- If a non-fatal error occurs, the task status is updated. Some destination buffers may be updated, and some may remain unchanged.

7.1.4.6. Task Limitations

- The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.
- The task buffers must remain valid until task is completed
- The given source buffer length must not exceed the **max_message_size** device capability
- The destination buffer list length must not exceed the **max_send_buf_list_len** property of the DOCA RDMA instance
- Other limitations are described in DOCA Core Task

5.4.8.7.1.5 7.1.5. Write Task

This task should be submitted when wishing to write data to remote memory (i.e., the memory on the remote side of the connection).

7.1.5.1. Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	**doca_rdma_task_write_set_conf**	**doca_rdma_cap_task_write_is_supported**
Number of tasks	**doca_rdma_task_write_set_conf**	–

Description	API to Set the Configuration	API to Query Support
Source buffer list length	`doca_rdma_set_max_send_buf_list_len`	`doca_rdma_cap_get_max_send_buf_list_len`

We don't have a way to export this macro.

7.1.5.2. Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Buffer pointing to a local memory address and holding the data to be written to the remote peer.	• Linked list buffers are supported • The source buffer should point to a local memory address from which the data should be read. The data is read only from the data section of the source buffer. • The source buffer (`src_buf`) is not mandatory and may be NULL when wishing to write zero bytes (may be relevant when wishing to keep a connection alive) • The length of the buffer is considered as its data length
Destination buffer	Points to a remote memory address. The data is written to the buffer upon successful completion of the task.	• Linked list buffers are not supported • The destination buffer (`dst_buf`) should point to a remote memory address • The length of the buffer is considered as its data length • The length of the destination buffer is considered as the length from the end of the data section until the end of the buffer, as this is the available memory that can be written to • The length of data written to the destination buffer depends on the total length of the given source buffer/list of buffers • May be NULL if the source buffer was set to NULL
RDMA connection	The RDMA connection to use on the task	Connection object provided in the connection establishment process as described in the "Establishing RDMA Connections" section

7.1.5.3. Task Output

Common output as described in DOCA Core Task.

7.1.5.4. Task Completion Success

After the task completes successfully, the following happens:

- The written data is appended after the data section in the destination buffer, as it was prior to the task submission.
- The data length is increased by the written data length

7.1.5.5. Task Completion Failure

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user

- If a non-fatal error occurs, the task status is updated. Some destination buffers may be updated, and some may remain unchanged.

7.1.5.6. Task Limitations

- The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.
- The task buffers must remain valid until task is completed
- The total length of the given source buffer/list of buffers must be not exceed the `max_message_size` device capability
- The source buffer list length must not exceed the `max_send_buf_list_len` property of the DOCA RDMA instance
- Other limitations are described in DOCA Core Task

5.4.8.7.1.6 7.1.6. Write With Immediate Task

This task should be submitted when wishing to write data to remote memory (i.e., the memory on the remote side of the connection).

7.1.6.1. Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_write_imm_set_conf`	`doca_rdma_cap_task_write_imm_is_supported`
Number of tasks	`doca_rdma_task_write_imm_set_conf`	–
Source buffer list length	`doca_rdma_set_max_send_buf_list_len`	`doca_rdma_cap_get_max_send_buf_list_len`

We don't have a way to export this macro.

7.1.6.2. Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Buffer pointing to a local memory address and holding the data to be written to the remote peer	• Linked list buffers are supported • The source buffer should point to a local memory address from which the data should be read. The data is read only from the data section of the source buffer. • The source buffer (`src_buf`) is not mandatory and may be NULL when wishing to write zero bytes • The length of the buffer is considered as its data length

Name	Description	Notes
Destination buffer	Points to a remote memory address. The data is written to the buffer upon successful completion of the task.	• Linked list buffers are not supported • The destination buffer (**dst_buf**) should point to a remote memory address • The length of the buffer is considered as its data length • The length of the destination buffer is considered as the length from the end of the data section until the end of the buffer, as this is the available memory that can be written to • The length of data written to the destination buffer depends on the total length of the given source buffer/list of buffers • May be NULL if the source buffer was set to NULL
Immediate data	32-bit value sent to the remote side, out-of-band	• Should be in a Big-Endian format • Value is received by the remote side only once a receive task completes successfully
RDMA connection	The RDMA connection to use on the task	Connection object provided in the connection establishment process as described in the "Establishing RDMA Connections" section

7.1.6.3. Task Output

Common output as described in DOCA Core Task.

7.1.6.4. Task Completion Success

A write with immediate task succeeds only if the remote side is expecting the immediate and had submitted a receive task beforehand.

After the task completes successfully, the following happens:

- The written data is appended after the data section in the destination buffer, as it was prior to the task submission
- The data length is increased by the written data length.

7.1.6.5. Task Completion Failure

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user
- If a non-fatal error occurs, the task status is updated. Some destination buffers may be updated, and some may remain unchanged.

7.1.6.6. Task Limitations

- The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.
- The tasks buffers must remain valid until task is completed
- The total length of the given source buffer/list of buffers must be not exceed the **max_message_size** device capability
- The source buffer list length must not exceed the **max_send_buf_list_len** property of the DOCA RDMA instance
- Other limitations are described in DOCA Core Task

5.4.8.7.1.7 7.1.7. Atomic Compare and Swap Task

This task should be submitted when wishing to execute an 8-byte atomic read-modify-write operation on the remote memory (i.e., the memory on the remote side of the connection), in which the remote value is retrieved and updated if it is equal to a given value.

7.1.7.1. Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_atomic_cmp_swp_set_conf`	`doca_rdma_cap_task_atomic_cmp_swp_is_supported`
Number of tasks	`doca_rdma_task_atomic_cmp_swp_set_conf`	–

7.1.7.2. Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Destination buffer	Buffer pointing to a remote memory address	• Linked list buffers are not supported • The destination buffer's data section must begin in a memory address aligned to 8-bytes • Only the first 8-bytes following the data address are considered for atomic operations
Compare data	64-bit value to be compared with the value in the destination buffer	
Swap data	64-bit value to be swapped with the value in the destination buffer	• The value in the destination buffer is only swapped if the compared data value is equal to the value in the destination buffer. Otherwise, the destination buffer remains unchanged.
Result buffer	Buffer pointing to a local memory address. The original value of the destination buffer (before executing the atomic operation) is written to the buffer upon success.	• Linked list buffers are not supported • The result is written to the first 8-bytes following the data address
RDMA connection	The RDMA connection to use on the task	Connection object provided in the connection establishment process as described in the "Establishing RDMA Connections" section

7.1.7.3. Task Output

Common output as described in DOCA Core Task.

7.1.7.4. Task Completion Success

After the task completes successfully, the following happens:

- If the compared values are equal, the value in the destination is swapped with the 64-bit value in the task's swap data field (`swap_data`)

- If the compared values are not equal, the value in the destination value remains unchanged
- The original value of the destination buffer (before executing the atomic operation) is written to the result buffer

7.1.7.5. Task Completion Failure

If the task fails midway:

- The context is stopped, and the task should be freed by the user

7.1.7.6. Task Limitations

- Task buffers must remain valid until task is completed
- Other limitations are described in DOCA Core Task

5.4.8.7.1.8 7.1.8. Atomic Fetch and Add Task

This task should be submitted when wishing to execute an 8-byte atomic read-modify-write operation on the remote memory (i.e., the memory on the remote side of the connection), in which the remote value is retrieved and increased by a given value.

7.1.8.1. Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_atomic_fetch_add_set_conf`	`doca_rdma_cap_task_atomic_fetch_add_is_supported`
Number of tasks	`doca_rdma_task_atomic_fetch_add_set_conf`	–

7.1.8.2. Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Destination buffer	Buffer that points to a remote memory address	• Linked list buffers are not supported • The destination buffer's data section must begin in a memory address aligned to 8-bytes • Only the first 8-bytes following the data address are considered for atomic operations
Add data	64-bit value to be added to the value in the destination buffer	
Result buffer	Buffer pointing to a local memory address. The original value of the destination buffer (before executing the atomic operation) is written to the buffer upon success.	• Linked list buffers are not supported • The result is written to the first 8-bytes following the data address
RDMA connection	The RDMA connection to use on the task	Connection object provided in the connection establishment process as described in the "Establishing RDMA Connections" section

7.1.8.3. Task Output

Common output as described in DOCA Core Task.

7.1.8.4. Task Completion Success

After the task completes successfully, the following happens:

- The value in the destination is increased by the 64-bit value in the task's add data field
- The original value of the destination buffer (before executing the atomic operation) is written to the result buffer

7.1.8.5. Task Completion Failure

If the task fails midway:

- The context is stopped, and the task should be freed by the user

7.1.8.6. Task Limitations

- Task buffers must remain valid until task is completed
- Other limitations are described in DOCA Core Task

5.4.8.7.1.9 7.1.9. Get Remote Sync Event Task

This task should be submitted when wishing to get the value of a remote sync event.

7.1.9.1. Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_remote_net_sync_event_get_set_conf`	`doca_rdma_cap_task_remote_net_sync_event_get_is_supported`
Number of tasks	`doca_rdma_task_remote_net_sync_event_get_set_conf`	–
Destination buffer list length	`doca_rdma_set_max_send_buf_list_len`	`doca_rdma_cap_get_max_send_buf_list_len`

We don't have a way to export this macro.

7.1.9.2. Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Sync Event	The remote DOCA Sync Event to get its value	
Destination buffer	Points to a local memory address. The Sync Event value is written to the buffer upon successful completion of the task.	• Linked list buffers are supported • The length of the each buffer is considered as the length from the end of the data section until the end of the buffer, as this is the available memory that can be written to in each buffer

Name	Description	Notes
RDMA connection	The RDMA connection to use on the task	Connection object provided in the connection establishment process as described in the "Establishing RDMA Connections" section

7.1.9.3. Task Output

Common output as described in DOCA Core Task.

Name	Description	Notes
Result length	The length of data received by the task	Valid only on successful completion of the task

7.1.9.4. Task Completion Success

After the task completes successfully, the following happens:

- The remote Sync Event value is appended after the data section in the destination buffer, as it was prior to the task submission
- The data length is increased by the retrieved data length

7.1.9.5. Task Completion Failure

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user
- If a non-fatal error occurs, the task status is updated. Some destination buffers may be updated, and some may remain unchanged.

7.1.9.6. Task Limitations

- The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.
- The destination buffer must remain valid until the task is completed
- The destination buffer list length must not exceed the **max_send_buf_list_len** property of the DOCA RDMA instance
- Other limitations are described in DOCA Core Task

5.4.8.7.1.10 7.1.10. Set Remote Sync Event Task

This task should be submitted when wishing to set a remote sync event to a given value.

7.1.10.1. Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	**doca_rdma_task_remote_net_sync_event_notify_set_set_conf**	**doca_rdma_cap_task_remote_net_sync_event_notify_set_is_supported**
Number of tasks	**doca_rdma_task_remote_net_sync_event_notify_set_set_conf**	–
Source buffer list length	**doca_rdma_set_max_send_buf_list_len**	**doca_rdma_cap_get_max_send_buf_list_len**

We don't have a way to export this macro.

7.1.10.2. Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Points to a local memory address from which the Sync Event should be retrieved	• Linked list buffers are supported • The data is retrieved only from the buffer data section, until 8-bytes • The length of the source buffer is considered its data length. The length of data retrieved from the source buffer will not exceed the Sync Event value length (8-bytes). Thus, the actual length retrieved depends on the minimal length between the source buffer and Sync Event value length.
Sync Event	The remote DOCA Sync Event to set its value	
RDMA connection	The RDMA connection to use on the task	Connection object provided in the connection establishment process as described in the "Establishing RDMA Connections" section

7.1.10.3. Task Output

Common output as described in DOCA Core Task.

7.1.10.4. Task Completion Success

After the task completes successfully, the following happens:

- The remote sync event value is set to the data in the source buffer

7.1.10.5. Task Completion Failure

If the task fails midway:

- If a fatal error occurs, the context is stopped, and the task should be freed by the user
- If a non-fatal error occurs, the task status is updated, and the Sync Event value is undefined

7.1.10.6. Task Limitations

- The operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.
- The source buffer must remain valid until the task completes
- The source buffer list length must not exceed the **max_send_buf_list_len** property of the DOCA RDMA instance
- Other limitations are described in DOCA Core Task

5.4.8.7.1.11 7.1.11. Add Remote Sync Event Task

This task should be submitted when wishing to atomically increase a remote sync event by a given value.

7.1.11.1. Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_rdma_task_remote_net_sync_event_notify_add_set_conf`	`doca_rdma_cap_task_remote_net_sync_event_notify_add_is_supported`
Number of tasks	`doca_rdma_task_remote_net_sync_event_notify_add_set_conf`	–

7.1.11.2. Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Sync event	A remote Sync Event	
Add data	64-bit value that is added to the Sync Event value	
Result buffer	Buffer pointing to a local memory address. The original Sync Event value of the destination buffer (before executing the atomic operation) is written to the buffer upon success.	• Linked list buffers are not supported • The result is written to the first 8-bytes following the data address
RDMA connection	The RDMA connection to use on the task	Connection object provided in the connection establishment process as described in the "Establishing RDMA Connections" section

7.1.11.3. Task Output

Common output as described in DOCA Core Task.

7.1.11.4. Task Completion Success

After the task completes successfully, the following happens:

- The value of the remote sync event is increased by the 64-bit value in the task's add data field
- The original value of the remote sync event (before executing the operation) is written to the result buffer

7.1.11.5. Task Completion Failure

If the task fails midway:

- The context is stopped, and the task should be freed by the user

7.1.11.6. Task Limitations

- Result buffer must remain valid until task is completed
- Other limitations are described in DOCA Core Task

5.4.8.7.2 7.2. Events

DOCA RDMA exposes asynchronous events to notify about changes that happen unexpectedly, according to DOCA Core architecture.

The only event DOCA RDMA exposes is common events as described in DOCA Core Event.

5.4.8.8 8. State Machine

The DOCA RDMA library follows the Context state machine as described in DOCA Core Context State Machine.

The following section describes how to move states and what is allowed in each state.

5.4.8.8.1 8.1.1. Idle

In this state, it is expected that application either:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to section "Configurations"
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
N/A	Create the context
Running	Call stop after making sure all tasks have been freed
Stopping	Call progress until all tasks are completed and freed

5.4.8.8.2 8.1.2. Starting

This state cannot be reached.

5.4.8.8.3 8.1.3. Running

In this state, it is expected that application:

1. Connects the RDMA instances on both peers. Refer to section "Establishing RDMA Connections" for more information.
2. Performs an RDMA instance disconnection process. Refer to section "Establishing RDMA Connections" for more information.
3. Performs a new connection of the RDMA instances on both peers after an RDMA instance disconnection process. Refer to section "Establishing RDMA Connections" for more information.

4. Accepts and indicates an established RDMA connection if the listening and CM channel monitoring was done by the user application. Refer to section "Connecting Using RDMA CM Connection Flow" for more information.
5. Allocates and submits tasks.
6. Calls progress to complete tasks and/or receive events.

Allowed operations:

- Performing a connection between 2 peers
- Allocating previously configured task
- Submitting an allocated task
- Calling stop

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

5.4.8.8.4 8.1.4. Stopping

In this state, it is expected that application:

1. Calls progress to complete all inflight tasks (tasks complete with failure)
2. Frees any completed tasks
3. Performs an RDMA instance disconnection process. Refer to section "Establishing RDMA Connections" for more information

Allowed operations:

- Call progress

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call progress and fatal error occurs
Running	Call stop without freeing all tasks

5.4.8.9 9. Alternative Datapath Options

DOCA RDMA allows data path to be run on DPA or GPU.

5.4.8.9.1 9.1. DPA Datapath

DOCA offers the DOCA DPA library which provides a programming model for offloading communication-centric user code to run on the DPA processor on the BlueField DPU. For additional information on the DOCA DPA library.

The user can choose to run an RDMA operation on the DPA datapath by configuring the DOCA RDMA context used by the application in the following manner:

1. Obtain DOCA CTX by calling **doca_rdma_as_ctx()**.
2. Set the datapath of the context to DPA by calling **doca_ctx_set_datapath_on_dpa()**. For additional information, refer to DOCA Core Alternative Data Path.
3. Finish context configuration and start the context by calling **doca_ctx_start()**. For additional information, refer to DOCA Context.

After configuring the datapath, the user can obtain a DPA handle for the DOCA RDMA context by calling **doca_rdma_get_dpa_handle()**.

The DPA handle can be used by the DOCA DPA library for datapath operations. For additional information, refer to DOCA DPA Communication Model.

5.4.8.9.2 9.2. GPU Datapath

DOCA offers the DOCA GPUNetIO library which provides a programming model for offloading the orchestration of the communication to a GPU CUDA kernel. For additional information on the DOCA GPUNetIO library.

The user can choose to run an RDMA operation on the GPU datapath by configuring the DOCA RDMA context used by the application in the following manner:

1. Obtain DOCA CTX by calling **doca_rdma_as_ctx()**.
2. Set the datapath of the context to GPU by calling **doca_ctx_set_datapath_on_gpu()**. For additional information, refer to DOCA Core Alternative Data Path.
3. Finish context configuration and start the context by calling **doca_ctx_start()**. For additional information, refer to DOCA Core Context.

After configuring the datapath, the user can obtain a GPU handle for the DOCA RDMA context by calling **doca_rdma_get_gpu_handle()**.

The GPU handle must be passed to a GPU CUDA kernel so the DOCA GPUNetIO CUDA device functions can execute datapath operations. For additional information, refer to DOCA GPUNetIO device functions.

5.4.8.10 10. DOCA RDMA Samples

These samples illustrate how to use the DOCA RDMA API to execute DOCA RDMA operations.

> All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

5.4.8.10.1 10.1. Running the Samples

1. Refer to the following documents:
   - DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - DOCA Troubleshooting for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

```
cd /opt/mellanox/doca/samples/doca_rdma/<sample_name>
meson /tmp/build
ninja -C /tmp/build
```

> The binary **doca_<sample_name>** is created under **/tmp/build/**.

3. Sample usage:
   - Common arguments

Argument	Description
-d, **--device**	IB device name (optional). If not provided, a random IB device is assigned.
-ld, **--local-descriptor-path**	Local descriptor file path that includes the local connection information to be copied to the remote program
-re, **--remote-descriptor-path**	Remote descriptor file path that includes the remote connection information to be copied from the remote program
-m, **--mmap-descriptor-path**	Remote descriptor file path that includes the remote mmap connection information to be copied from the remote program
-g, **--gid-index**	GID index for DOCA RDMA (optional)
-tt, **---transport-type**	Transport type for DOCA RDMA (RC or DC; optional); only useful for single out-of-band RDMA connection currently
-cm, **--use-rdma-cm**	Whether to use RDMA CM or OOB to setup connection
-lp, **--listen-port**	Server listen port number
-sa, **--server-addr**	RDMA CM server device address
-sat, **--server-addr-type**	RDMA CM server device address type: IPv4, IPv6, or GID

   - Sample-specific arguments

Sample	Argument	Description
RDMA Read Responder	**-r**, **--read-string**	String to read (optional). If not provided, "Hi DOCA RDMA!" is defined.
RDMA Send RDMA Send Immediate RDMA Multi-conn Send	**-s**, **--send-string**	

Sample	Argument	Description
RDMA Write Requester RDMA Write Immediate Requester	`-w`, `--write-string`	
RDMA Multi-conn Send RDMA Multi-conn Receive	`-nc`, `--num-connections`	Number of connections for DOCA RDMA (optional); max connections number must be ≤8 in this sample

4. For additional information per sample, use the `-h` option:

```
/tmp/build/<sample_name> -h
```

5.4.8.10.2 10.2. Configuring Scalable Functions for DOCA RDMA Samples

When running DOCA RDMA samples on DPU-DPU setups with multiple SFs configured for a port, ensure the SF attribute `trust` is set to `off`.

By default, two SFs are present after logging into a DPU:

- `en3f0pf0sf0` corresponds to port `p0`.
- `en3f1pf1sf0` corresponds to port `p1`.

Steps:

1. Run the following command on both DPUs:

```
/opt/mellanox/iproute2/sbin/mlxdevm port add pci/0000:03:00.0 flavour pcisf pfnum 0 sfnum 1 controller 0
```

2. Find the SF index:

```
/opt/mellanox/iproute2/sbin/mlxdevm port show -jp
```

3. Set the SF attribute `trust` to `off`:

```
/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/SF-INDEX trust off
```

4. Configure the SF:
   - Set the hardware address:

```
/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/SF-INDEX hw_addr MAC-ADDR-USER-SPECIFIED
```

   - Activate the SF:

```
/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/SF-INDEX state active
```

5. Find SF-NEXT-SEQ:

```
devlink dev show
```

6. Unbind and bind the SF:

```
echo mlx5_core.sf.SF-NEXT-SEQ > /sys/bus/auxiliary/drivers/mlx5_core.sf_cfg/unbind
echo mlx5_core.sf.SF-NEXT-SEQ > /sys/bus/auxiliary/drivers/mlx5_core.sf/bind
```

7. Verify the SF status:

```
/opt/mellanox/iproute2/sbin/mlxdevm port show -jp
```

8. Add the new SF's representor to **ovs-vswitch**.
9. Assign a new IP address to the SF.
10. Run the DOCA RDMA samples using the new SF on port **p0**.

5.4.8.10.3 10.3. Samples

> These samples are also available on GitHub.

Each sample presents a connection between two peers, transferring data from one to another, using a different RDMA operation in each sample. For more information on the available RDMA operations, refer to section "Tasks".

Each sample is comprised of two executables, each running on a peer.

The samples can run on either DPU or host, as long as the chosen peers have a connection between them.

> Prior to running the samples, ensure that the chosen devices, selected by the device name and the GID index, are set correctly and have a connection between one another. In each sample, it is the user's responsibility to copy the descriptors between the peers.

Most of the samples follow the following main basic steps:

1. Allocating resources:
    a. Locating and opening a device. The chosen device is one that supports the tasks relevant for the sample. If the sample requires no task, any device may be chosen.
    b. Creating a local MMAP and configuring it (including setting the MMAP memory range and relevant permissions)
    c. Creating a DOCA PE
    d. Creating an RDMA instance and configuring it (including setting the relevant permissions)
    e. Connecting the RDMA context to the PE
2. Sample-specific configurations:
    a. Configuring the tasks relevant to the sample, if any. Including:
        i. Setting the number of tasks for each task type.
        ii. Setting callback functions for each task type, with the following logic:
            1. Successful completion callback:
                a. Verifying the data received from the remote, if any, is valid.
                b. Printing the transferred data.

c. Freeing the task and task-specific resources (such as source/destination buffers).
d. If an error occurs in steps a. and b., update the error that was encountered.

> ⚠ If the context is not in an idle state, only the first error in the flow is saved.

e. Decreasing the number of remaining tasks and stopping the context once it reaches 0.

2. Failed completion callback:
a. Update the error that was encountered.

> ⚠ If the context is not in an idle state, only the first error in the flow is saved.

b. Freeing the task and task-specific resources (such as source/destination buffers).
c. Decreasing the number of remaining tasks and stopping the context once it reaches 0.

b. Setting a state change callback function, with the following logic:

- Once the context moves to Starting state (can only be reached from Idle), export and connect the RDMA and, in some samples, export the local mmap or the sync event.

> ⚠ During this step, the user is responsible for copying the descriptors between the two peers.

> ⚠ The descriptors are to be read and used only by the peer, using the relevant DOCA functions (the descriptors contain encoded data).

- Once the context moves to Running state (can only be reached from Starting state in RDMA samples):
  - In some samples, only print a log and wait for the peer, or synchronize events
  - In other samples, prepare and submit a task:
    a. If needed, create an mmap from the received exported mmap descriptor, passed from the peer.
    b. Request the required buffers from the buffer inventory.
    c. Allocate and initiate the required task, together with setting the number of remaining tasks parameter as the task's user data.
    d. Submit the task.
- Once the context moves to Stopping state, print a relevant log.
- Once the context moves to Idle state:
  1. Print a relevant log.
  2. Send update that the main loop may be stopped.

3. Setting the program's resources as the context user data to be used in callbacks.
4. Creating a buffer inventory and starting it.
5. Starting the context.

   > After starting the context, the state change callback function is called by the PE which executes the relevant steps.

   > In a successful run, each section is executed in the order they are presented in section 2.b.

6. Progressing the PE until the context returns to Idle state and the main loop may be stooped, either because of a run in which all tasks have been completed, or due to a fatal error.
7. Cleaning up the resources.

5.4.8.10.3.1 10.3.1. RDMA Read

10.3.1.1. RDMA Read Requester

This sample illustrates how to read from a remote peer (the responder) using DOCA RDMA.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample are set to local read and write.
2. A read task is configured for this sample.
3. In this sample, data is read from the peer, verified to be valid, and printed in the successful task completion callback.
4. The local mmap is not exported as the peer does not intend to access it.
5. To read from the peer, a remote mmap is created from the peer's exported mmap.

Reference:

- **/opt/mellanox/doca/samples/doca_rdma/rdma_read_requester/rdma_read_requester_sample.c**
- **/opt/mellanox/doca/samples/doca_rdma/rdma_read_requester/rdma_read_requester_main.c**
- **/opt/mellanox/doca/samples/doca_rdma/rdma_read_requester/meson.build**

10.3.1.2. RDMA Read Responder

This sample illustrates how to set up a remote peer for a DOCA RDMA read request.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for both the local mmap and the RDMA instance in this sample allow for RDMA read.
2. No tasks are configured for this sample, and thus no tasks are prepared and submitted, nor are there task completion callbacks.
3. The local mmap is exported to the remote memory to allow it to be used by the peer for RDMA read.
4. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- **/opt/mellanox/doca/samples/doca_rdma/rdma_read_responder/rdma_read_responder_sample.c**
- **/opt/mellanox/doca/samples/doca_rdma/rdma_read_responder/rdma_read_responder_main.c**
- **/opt/mellanox/doca/samples/doca_rdma/rdma_read_responder/meson.build**

5.4.8.10.3.2 10.3.2. RDMA Write

10.3.2.1. RDMA Write Requester

This sample illustrates how to write to a remote peer (the responder) using DOCA RDMA.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample are set to local read and write.
2. A write task is configured for this sample.
3. In this sample, data is written to the peer and printed in the successful task completion callback.
4. The local mmap is not exported as the peer does not intend to access it.
5. To write to the peer, a remote mmap is created from the peer's exported mmap.

Reference:

- **/opt/mellanox/doca/samples/doca_rdma/rdma_write_requester/rdma_write_requester_sample.c**
- **/opt/mellanox/doca/samples/doca_rdma/rdma_write_requester/rdma_write_requester_main.c**
- **/opt/mellanox/doca/samples/doca_rdma/rdma_write_requester/meson.build**

10.3.2.2. RDMA Write Responder

This sample illustrates how to set up a remote peer for a DOCA RDMA write request.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for both the local mmap and the RDMA instance in this sample allow for RDMA write.
2. No tasks are configured for this sample, and thus no tasks are prepared and submitted, nor are there task completion callbacks. In this sample, the data written to the memory of the responder is printed once the context state is changed to Running, using the state change callback. This is done only after receiving input from the user, indicating that the requester had finished writing.
3. The local mmap is exported to the remote memory to allow it to be used by the peer for RDMA write.
4. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- **/opt/mellanox/doca/samples/doca_rdma/rdma_write_responder/rdma_write_responder_sample.c**
- **/opt/mellanox/doca/samples/doca_rdma/rdma_write_responder/rdma_write_responder_main.c**
- **/opt/mellanox/doca/samples/doca_rdma/rdma_write_responder/meson.build**

5.4.8.10.3.3 10.3.3. RDMA Write Immediate

10.3.3.1. RDMA Write Immediate Requester

This sample illustrates how to write to a remote peer (the responder) using DOCA RDMA along with a 32-bit immediate value which is sent OOB.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample are set to local read and write.
2. A write with immediate task is configured for this sample.
3. In this sample, data is written to the peer and printed in the successful task completion callback.
4. The local mmap is not exported as the peer does not intend to access it.
5. To write to the peer, a remote mmap is created from the peer's exported mmap.

Reference:

- **/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_requester/rdma_write_immediate_requester_sample.c**
- **/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_requester/rdma_write_immediate_requester_main.c**
- **/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_requester/meson.build**

10.3.3.2. RDMA Write Immediate Responder

This sample illustrates how the set up a remote peer for a DOCA RDMA write request whilst receiving a 32-bit immediate value from the peer's OOB.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for both the local mmap and the RDMA instance in this sample allow for RDMA write.
2. A receive task is configured for this sample to retrieve the immediate value. Failing to submit a receive task prior to the write with immediate task results in a fatal failure.
3. In this sample, the successful task completion callback also includes:
    a. Checking the result opcode, to verify that the receive task has completed after receiving a write with immediate request.
    b. Verifying the data written to the memory of the responder is valid and printing it, along with the immediate data received.
4. The local mmap is exported to the remote memory, to allow it to be used by the peer for RDMA write.
5. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- **/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_responder/rdma_write_immediate_responder_sample.c**
- **/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_responder/rdma_write_immediate_responder_main.c**

- `/opt/mellanox/doca/samples/doca_rdma/rdma_write_immediate_responder/meson.build`

5.4.8.10.3.4 10.3.4. RDMA Send and Receive

10.3.4.1. RDMA Send

This sample illustrates how to send a message to a remote peer using DOCA RDMA.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample are set to local read and write.
2. A send task is configured for this sample.
3. In this sample, the data sent is printed during the task preparation, not in the successful task completion callback.
4. The local mmap is not exported as the peer does not intend to access it.
5. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_send/rdma_send_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_send/rdma_send_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_send/meson.build`

10.3.4.2. RDMA Receive

This sample illustrates how the remote peer can receive a message sent by the peer (the sender).

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample are set to local read and write.
2. A receive task is configured for this sample to retrieve the sent data. Failing to submit a receive task prior to the send task results in a fatal failure.
3. In this sample, data is received from the peer verified to be valid and printed in the successful task completion callback.
4. The local mmap is not exported as the peer does not intend to access it.
5. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_receive/rdma_receive_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_receive/rdma_receive_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_receive/meson.build`

5.4.8.10.3.5 10.3.5. RDMA Send and Receive with Immediate

10.3.5.1. RDMA Send with Immediate

This sample illustrates how to send a message to a remote peer using DOCA RDMA along with a 32-bit immediate value which is sent OOB.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample are set to local read and write.
2. A send with immediate task is configured for this sample.
3. In this sample, the data sent is printed during the task preparation, not in the successful task completion callback.
4. The local mmap is not exported as the peer does not intend to access it.
5. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- **/opt/mellanox/doca/samples/doca_rdma/rdma_send_immediate/rdma_send_immediate_sample.c**
- **/opt/mellanox/doca/samples/doca_rdma/rdma_send_immediate/rdma_send_immediate_main.c**
- **/opt/mellanox/doca/samples/doca_rdma/rdma_send_immediate/meson.build**

10.3.5.2. RDMA Receive with Immediate

This sample illustrates how the remote peer can receive a message sent by the peer (the sender) while also receiving a 32-bit immediate value from the peer's OOB.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample are set to local read and write.
2. A receive task is configured for this sample to retrieve the sent data and the immediate value. Failing to submit a receive task prior to the send with immediate task results in a fatal failure.
3. In this sample, the successful task completion callback also includes:
    a. Checking the result opcode, to verify that the receive task has completed after receiving a sent message with an immediate.
    b. Verifying the data received from the peer is valid and printing it along with the immediate data received.
4. In this sample, data is received from the peer verified to be valid and printed in the successful task completion callback.
5. The local mmap is not exported as the peer does not intend to access it.
6. No remote mmap is created as there is no intention to access the remote memory in this sample.

Reference:

- **/opt/mellanox/doca/samples/doca_rdma/rdma_receive_immediate/rdma_receive_immediate_sample.c**
- **/opt/mellanox/doca/samples/doca_rdma/rdma_receive_immediate/rdma_receive_immediate_main.c**
- **/opt/mellanox/doca/samples/doca_rdma/rdma_receive_immediate/meson.build**

5.4.8.10.3.6 10.3.6. RDMA Remote Sync Event

This sample illustrates how to synchronize between local sync event and a remote sync event DOCA RDMA.

10.3.6.1. RDMA Remote Sync Event Requester

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample are set to local read and write.
2. A "remote net sync event notify set" task is configured for this sample.
    - For this task, the successful task completion callback has the following logic:
        i. Printing an info log saying the task was successfully completed and a specific successful completion log for the task.
        ii. Decreasing the number of remaining tasks. Once 0 is reached:
            1. Freeing the task and task-specific resources.
            2. Stopping the context.
    - For this task, the failed task completion callback stops the context even when the number of remaining tasks is different than 0 (since the synchronization between the peers would fail).
3. A "remote net sync event get" task is configured for this sample.
    - For this task, the successful task completion callback also includes:
        i. Resubmitting the task, until a value greater than or equal to the expected value is retrieved.
        ii. Once such value is retrieved, submitting a "remote net sync event notify set" task to signal sample completion, including:
            1. Updating the successful completion message accordingly.
            2. Increasing the number of submitted tasks.
            3. If an error was encountered, and the "remote net sync event notify set" task was not submitted, the task and task resources are freed.
    - For this task, the failed task completion callback also includes freeing the "remote net sync event notify set" task and task resources.
4. The local mmap is not exported as the peer does not intend to access it.
5. No remote mmap is created as there is no intention to access the remote memory in this sample.
6. To synchronize events with the peer, a sync event remote net is created from the peer's exported sync event.
7. Both tasks are prepared and submitted in the state change callback, once the context moves from starting to running.
8. The user data of the "remote net sync event get" task points to the "remote net sync event notify set" task.

Reference:

- **`/opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_requester/rdma_sync_event_requester_sample.c`**
- **`/opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_requester/rdma_sync_event_requester_main.c`**
- **`/opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_requester/meson.build`**

10.3.6.2. RDMA Remote Sync Event Responder

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample are set to local read and write.
2. This sample includes creating a local sync event and exporting it to the remote memory to allow the peer to create a remote handle.

3. No tasks are configured for this sample, and thus no tasks are prepared and submitted, nor are there task completion callbacks. In this sample, the following steps are executed once the context moves from starting to running, using the state change callback:
    a. Waiting for the sync event to be signaled from the remote side.
    b. Notifying the sync event from the local side.
    c. Waiting for completion notification from the remote side.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_responder/rdma_sync_event_responder_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_responder/rdma_sync_event_responder_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_sync_event_responder/meson.build`

5.4.8.10.3.7 10.3.7. RDMA Multi-connections Send and Receive

The following samples illustrates how to multiple connections can be performed to demonstrate an exchange of messages between peers.

Please note, using DOCA RDMA CM flow (see section "Connecting Using RDMA CM Connection Flow"):

1. One sample would act as Server while the other would act as a client
2. Multiple instances of the sample acting as client need to be independently executed to simulate each client peer to reach the amount of desired RDMA connections.

10.3.7.1. RDMA Multi-conn Send

This sample shows how multiple connections can be established and demonstrates:

- How multiple peers (clients) can send a message to a remote single peer (server) using DOCA RDMA CM flow (see section "Connecting Using RDMA CM Connection Flow").
- How a single peer (server) can send a message to multiple remote peers (clients) using DOCA RDMA CM flow (see section "Connecting Using RDMA CM Connection Flow").
- How multiple remote peers can send a message to their connected peers using the DOCA RDMA exporting and connection flow (see section "Exporting and Connecting RDMA").

In DOCA RDMA CM flow multiple instances of this sample need to be independently executed to simulate each client peer to reach the amount of desired RDMA connections.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample are set to local read and write.
2. A send task is configured for this sample.
3. In this sample, the data sent is printed during the task preparation, not in the successful task completion callback.
4. The local mmap is not exported as the peer does not intend to access it.
5. No remote mmap is created as there is no intention to access the remote memory in this sample.
6. The number of connections option can be set to

a. the number of remote peers expected by the single peer (server sender) when using DOCA RDMA CM flow (see section "Connecting Using RDMA CM Connection Flow").
b. the number of peers to connect to its remote peers using the DOCA RDMA exporting and connection flow (see section "Exporting and Connecting RDMA").

> In DOCA RDMA CM flow, the number of connections cannot be used to simulate each client peer.

Reference:

- **/opt/mellanox/doca/samples/doca_rdma/rdma_multi_conn_send/rdma_multi_conn_send_sample.c**
- **/opt/mellanox/doca/samples/doca_rdma/rdma_multi_conn_send/rdma_multi_conn_send_main.c**
- **/opt/mellanox/doca/samples/doca_rdma/rdma_multi_conn_send/meson.build**

10.3.7.2. RDMA Multi-conn Receive

This sample shows how multiple connections can be performed and demonstrates:

- How multiple remote peers (clients) can receive a message sent by one single peer (the sender server) using DOCA RDMA CM flow (see section "Connecting Using RDMA CM Connection Flow").
- How a single remote peer (server) can receive a message from multiple peers (the sender clients) using DOCA RDMA CM flow (see section "Connecting Using RDMA CM Connection Flow").
- How multiple remote peers can receive a message sent by their connected peers using the DOCA RDMA exporting and connection flow (see section "Exporting and Connecting RDMA").

> In DOCA RDMA CM flow multiple instances of this sample need to be independently executed to simulate each client peer to reach the amount of desired RDMA connections.

The sample logic is as presented in the General Sample Steps, with attention to the following:

1. The permissions for the local mmap in this sample are set to local read and write.
2. A receive task is configured for this sample to retrieve the sent data.

   > Failing to submit a receive task prior to the send task results in a fatal failure.

3. In this sample, data is received from the peer verified to be valid and printed in the successful task completion callback.
4. The local mmap is not exported as the peer does not intend to access it.
5. No remote mmap is created as there is no intention to access the remote memory in this sample.
6. The number of connections can be set to:
   a. The number of peers expected by the single remote peer (server) when using DOCA RDMA CM flow (see section "Connecting Using RDMA CM Connection Flow").

b. The number of remote peers connected to their sender peers using the DOCA RDMA exporting and connection flow (see section "Exporting and Connecting RDMA").

> In DOCA RDMA CM flow the number of connections cannot be used to simulate each client peer.

Reference:

- `/opt/mellanox/doca/samples/doca_rdma/rdma_multi_conn_receive/rdma_multi_conn_receive_sample.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_multi_conn_receive/rdma_multi_conn_receive_main.c`
- `/opt/mellanox/doca/samples/doca_rdma/rdma_multi_conn_receive/meson.build`

5.4.9 DOCA RDMA Verbs

This guide provides an overview and configuration instructions for the DOCA Verbs API.

5.4.9.1 1. Introduction

DOCA RDMA Verbs is a Verbs-like API that exposes the low-level features of Remote Direct Memory Access (RDMA). By enabling direct access to remote machine memory without CPU interruptions, DOCA RDMA Verbs reduces context switching for I/O operations, resulting in lower latency and higher bandwidth compared to traditional network communication methods.

> DOCA RDMA Verbs supports both InfiniBand and Ethernet using RoCE.

Key features:

- Vendor agnostic: Supports multiple RDMA protocols.
- Verbs compatibility: Built on established Verbs interfaces for easy migration from existing Verbs-based applications.
- Low-level control: Provides direct access to hardware capabilities for optimal performance tuning.
- Generic design: Offers a unified API interface for various RDMA protocols.

DOCA RDMA Verbs operates on Verbs objects and is designed to ensure compatibility while extending capabilities for new RDMA protocols and hardware features. The library includes both native DOCA Verbs APIs and bridge APIs for seamless integration with existing IB Verbs applications.

For detailed API documentation and implementation, refer to the `doca_verbs.h` and `doca_verbs_bridge.h` header files.

5.4.9.2 2. Prerequisites

This library follows the architecture of DOCA Core and Verbs, it is recommended to read the following sections before proceeding:

- DOCA Core Device
- Libibverbs external guides

5.4.9.3 3. Environment

DOCA RDMA Verbs-based applications can run on either the host machine or the NVIDIA® BlueField® networking platform (DPU or SuperNIC).
BF2 doesn't support DOCA verbs.

5.4.9.4 4. Architecture

The DOCA Verbs API is centered around Verbs objects, designed to utilize the RDMA NIC's capabilities for high-performance data transfer. Each object plays a specific role in enabling and managing RDMA operations.

The API is categorized into two main function types:

- Control Path APIs: For initializing and configuring RDMA resources based on user requirements.
- Data Path APIs: For performing RDMA operations like sending, receiving, reading, and writing data. These functions are bridge functions relying on IB Verbs structures (e.g., `ibv_send_wr`, `ibv_wc`).

5.4.9.4.1 4.1. Objects

5.4.9.4.1.1 4.1.1. DOCA RDMA Verbs Device Attributes

> For DOCA Device documentation, read DOCA Core Device.

A DOCA Device represents a hardware device that supports RDMA operations through the DOCA RDMA Verbs library, serving as the foundation for creating Verbs contexts, protection domains, and other necessary resources.

DOCA RDMA Verbs Device Attributes provide detailed information about the device's capabilities and limitations, including:

- Maximum number of supported Queue Pairs (QPs), Completion Queues (CQs), Memory Regions (MRs), Protection Domains (PDs), Address Handles (AHs), and Shared Receive Queues (SRQs)
- Support for different QP types (RC, UC)
- Support for atomic operations
- Support for specialized data paths like DPA or GPU acceleration

These attributes are typically queried during application initialization to validate resource requirements against hardware capabilities.

The `doca_verbs_query_device` function is used to retrieve device attributes:

```
doca_error_t doca_verbs_query_device(struct doca_verbs_context *context, struct doca_verbs_device_attr
**verbs_device_attr)
```

5.4.9.4.1.2 4.1.2. DOCA RDMA Verbs Context

A DOCA RDMA Verbs Context represents a logical connection to an RDMA device, providing access to the RDMA NIC and serving as the root for other Verbs-related objects. It enables centralized cleanup and resource management for objects created under the Verbs library.

DOCA RDMA Verbs context can be created or imported (more information on IB Verbs Bridge section):

- Create context from **devinfo** or **ibv_device**:

```
doca_error_t doca_verbs_context_create(struct doca_devinfo *devinfo, uint32_t flags, struct
doca_verbs_context **verbs_context)
doca_error_t doca_verbs_bridge_verbs_context_create(struct ibv_device *ibv_dev, uint32_t flags, struct
doca_verbs_context **verbs_context)
```

  DOCA manages the resources and is independent of the IBV resources.
- The context can also be imported from an already existing IB Verbs context using the RDMA bridge:

```
doca_error_t doca_verbs_bridge_verbs_context_import(struct ibv_context *ibv_ctx, uint32_t flags, struct
doca_verbs_context **verbs_context)
```

  The user is responsible for cleanup operations, as the library does not manage the lifecycle of imported contexts.

  > ⚠ When importing using the bridge method, the IB context must remain open until the Verbs context is closed.

5.4.9.4.1.3 4.1.3. DOCA RDMA Verbs Protection Domain (PD)

A DOCA RDMA Verbs Protection Domain (PD) defines a logical security boundary within an RDMA device. It acts as a *sandbox* for grouping RDMA resources and ensures hardware-enforced isolation between applications or components sharing the same RDMA device.

The PD guarantees that all RDMA objects created within the same domain can trust and safely interact with each other, while any access from outside the domain is explicitly prohibited. This isolation creates a secure environment for RDMA operations.

Key features:

- Memory Access Control: Restricts RDMA operations to authorized memory regions.
- Hardware-Enforced Security: Utilizes hardware mechanisms to prevent unauthorized access.
- Application Isolation: Enforces separation between different applications using the same RDMA device.
- Safe Resource Grouping: Organizes related RDMA resources (e.g., memory regions, queue pairs) within a trusted context.

4.1.3.1. Interoperability with IB Verbs

- You can import an existing **ibv_pd** instance into DOCA RDMA Verbs using:

```
doca_error_t doca_verbs_bridge_verbs_pd_import(struct ibv_pd *pd, struct doca_verbs_pd **verbs_pd)
```

- For use cases such as registering memory with **ibv_reg_mr()**, retrieve the underlying IB Verbs PD:

```
struct ibv_pd *doca_verbs_bridge_verbs_pd_get_ibv_pd(const struct doca_verbs_pd *verbs_pd)
```

4.1.3.2. PD Creation and Usage

- Create a DOCA RDMA Verbs PD

```
doca_error_t doca_verbs_pd_create(struct doca_verbs_context *verbs_context, struct doca_verbs_pd
**verbs_pd);
```

- When initializing a queue pair (QP), associate it with the PD:

```
doca_error_t doca_verbs_qp_init_attr_set_pd(struct doca_verbs_qp_init_attr *verbs_qp_init_attr, struct
doca_verbs_pd *pd);
```

5.4.9.4.1.4 4.1.4. Memory Region (MR)

A Memory Region (MR) is a block of memory registered with the RDMA NIC to enable direct memory access through RDMA operations. Registering a memory region allows the NIC to perform zero-copy data transfers while enforcing access permissions and hardware-level protections.

4.1.4.1. Access Permissions

When registering an MR, the following access permissions can be enabled:

- Local Write – Allows local RDMA operations to write to the memory.
- Remote Read – Allows remote RDMA peers to read from the memory.
- Remote Write – Allows remote RDMA peers to write to the memory.
- Atomic Operations – Enables support for remote atomic operations (e.g., compare-and-swap, fetch-and-add).

4.1.4.2. Memory Keys

Each MR is associated with two keys used to authorize RDMA access:

- Local Key (lkey) – Used by the local RDMA NIC for internal access to the memory.
- Remote Key (rkey) – Shared with remote peers to grant them access, based on the permissions set during registration.

4.1.4.3. Registration

Memory regions are registered using the **ibv_reg_mr()** function from the RDMA Core API. This call:

- Allocates the necessary hardware resources for the memory region.
- Returns the local and remote keys (lkey and rkey).
- Enforces the specified access permissions for RDMA operations.

5.4.9.4.1.5 4.1.5. DOCA RDMA Verbs Queue Pair

A DOCA RDMA Verbs Queue Pair (QP) represents a connection between two RDMA-capable nodes. Each QP consists of a Send Queue and a Receive Queue, which process Work Requests (WRs) for RDMA operations.

4.1.5.1. QP Types

- RC (Reliable Connection) – Guarantees reliable, ordered delivery of messages.
- UC (Unreliable Connection) – Provides best-effort delivery without guaranteed ordering or reliability.

4.1.5.2. QP Initialization

Before creating a QP, users must first configure the desired attributes using a **doca_verbs_qp_init_attr** object. This object specifies:

- Send and receive queue sizes
- Associated Completion Queues (CQs)
- Protection Domain (PD)
- Optional data-path configuration

The configured initialization object is then used to instantiate the QP.

```
doca_error_t doca_verbs_qp_init_attr_create(struct doca_verbs_qp_init_attr **verbs_qp_init_attr);
doca_error_t doca_verbs_qp_create(struct doca_verbs_context *verbs_context,
                                  struct doca_verbs_qp_init_attr *verbs_qp_init_attr,
                                  struct doca_verbs_qp **verbs_qp);
```

4.1.5.3. QP Modification

After creation, QPs may require state transitions or updates to their configuration. This is done using a **doca_verbs_qp_attr** object.

Common use cases include transitioning a QP from INIT → RTR (Ready to Receive) → RTS (Ready to Send) or updating addressing and connection parameters.

```
doca_error_t doca_verbs_qp_attr_create(struct doca_verbs_qp_attr **verbs_qp_attr);
doca_error_t doca_verbs_qp_modify(struct doca_verbs_qp *verbs_qp,
                                  struct doca_verbs_qp_attr *verbs_qp_attr,
                                  int attr_mask);
```

4.1.5.4. QP Query

To inspect the current configuration or state of an existing QP, use the **doca_verbs_qp_query()** function. This retrieves attributes and initialization parameters according to the specified mask.

```
doca_error_t doca_verbs_qp_query(struct doca_verbs_qp *verbs_qp,
                                 struct doca_verbs_qp_attr *verbs_qp_attr,
                                 struct doca_verbs_qp_init_attr *verbs_qp_init_attr,
                                 int *attr_mask);
```

4.1.5.5. External Data Path Mode

DOCA RDMA Verbs supports an external data path mode, allowing the data path to be implemented by the user or a third-party library (e.g., **doca_dpa** or **doca_gpunetio**). To enable this mode:

```
doca_error_t doca_verbs_qp_init_attr_set_external_datapath_en(
    struct doca_verbs_qp_init_attr *verbs_qp_init_attr,
    uint8_t external_datapath_en);
```

Refer to the Bridge Execution Phase and Alternative Data Path Options sections for more details on integrating external data-path logic.

5.4.9.4.1.6 4.1.6. DOCA RDMA Verbs Completion Queue

A DOCA RDMA Verbs Completion Queue (CQ) is a data structure that collects notifications for completed RDMA operations. CQs are essential for tracking the status of Send and Receive Work Requests (WRs) submitted to a Queue Pair (QP).

4.1.6.1. Associating CQs with a QP

During QP initialization, the application must specify the CQs for send and receive operations:

```
doca_error_t doca_verbs_qp_init_attr_set_send_cq(
    struct doca_verbs_qp_init_attr *verbs_qp_init_attr,
    struct doca_verbs_cq *send_cq);

doca_error_t doca_verbs_qp_init_attr_set_receive_cq(
    struct doca_verbs_qp_init_attr *verbs_qp_init_attr,
    struct doca_verbs_cq *receive_cq);
```

4.1.6.2. CQ Initialization

A **doca_verbs_cq_attr** object defines the configuration for the CQ, including:

- Queue size
- Entry size
- Completion channel binding
- Data-path mode settings

To create and configure a CQ:

```
doca_error_t doca_verbs_cq_attr_create(struct doca_verbs_cq_attr **verbs_cq_attr);
doca_error_t doca_verbs_cq_create(struct doca_verbs_context *verbs_context,
                                  struct doca_verbs_cq_attr *verbs_cq_attr,
                                  struct doca_verbs_cq **verbs_cq);
```

4.1.6.3. External Data Path Mode

DOCA RDMA Verbs supports an external data path mode for advanced integration with external processing frameworks (e.g., **doca_dpa**, **doca_gpunetio**). In this mode, CQ data management and notification handling are offloaded outside of DOCA RDMA Verbs.

To enable this mode:

```
doca_error_t doca_verbs_cq_attr_set_external_datapath_en(
    struct doca_verbs_cq_attr *cq_attr,
    uint8_t external_datapath_en);
```

Refer to the Bridge Execution Phase and Alternative Data Path Options sections for more details.

5.4.9.4.1.7 4.1.7. DOCA RDMA Verbs Address Handle Attribute

A DOCA RDMA Verbs Address Handle Attribute (AH) is a data structure that encapsulates the addressing information required to establish communication with remote Queue Pairs (QPs). It is typically used during QP state transitions—such as INIT → RTR (Ready to Receive)—to define the destination addressing parameters.

4.1.7.1. Key Attributes

The AH structure may include the following fields depending on the address type and transport:

- Address Type – Specifies the type of addressing:
  - IPv4
  - IPv6
  - InfiniBand (with or without GRH)
- GID (Global Identifier) – Used for global routing in RoCE or IB with GRH.
- LID (Local Identifier) – Used for local routing in InfiniBand.
- Hop Limit – Specifies the maximum number of network hops (TTL equivalent).
- Service Level (SL) – Indicates QoS priority level on the fabric.
- Traffic Class – Classifies packets for differentiated services in IP-based networks.

4.1.7.2. Creating and Associating AH Attributes

To define an address handle and attach it to a QP modification request:

```
doca_error_t doca_verbs_ah_attr_create(
    struct doca_verbs_context *verbs_context,
    struct doca_verbs_ah_attr **verbs_ah);

doca_error_t doca_verbs_qp_attr_set_ah_attr(
    struct doca_verbs_qp_attr *verbs_qp_attr,
    struct doca_verbs_ah_attr *ah_attr);
```

The configured AH attribute can then be used during QP state transitions, particularly in INIT2RTR operations, to ensure proper remote addressing.

5.4.9.4.1.8 4.1.8. DOCA RDMA Verbs Completion Channel

A DOCA RDMA Verbs Completion Channel (CC) is a mechanism for managing asynchronous completion events from one or more associated Completion Queues (CQs). Completion channels enable efficient event-driven processing, reducing the need for continuous polling.

4.1.8.1. Creation and Association

To create a completion channel and associate it with a CQ:

```
doca_error_t doca_verbs_comp_channel_create(
    struct doca_verbs_context *verbs_context,
    struct doca_verbs_comp_channel **verbs_comp_channel);

doca_error_t doca_verbs_cq_attr_set_comp_channel(
    struct doca_verbs_cq_attr *cq_attr,
    struct doca_verbs_comp_channel *comp_channel);
```

4.1.8.2. Receiving and Acknowledging Events

To retrieve the next event (i.e., notification of completed work) from the completion channel:

```
doca_error_t doca_verbs_get_cq_event(
    struct doca_verbs_comp_channel *verbs_comp_channel,
    struct doca_verbs_cq **verbs_cq,
    void **cq_context);
```

Each event retrieved using **doca_verbs_get_cq_event()** must be acknowledged using:

```
void doca_verbs_ack_cq_events(struct doca_verbs_cq *verbs_cq, unsigned int nevents);
```

4.1.8.3. Integration with Epoll

The completion channel can be integrated into a standard event loop using **epoll** for scalable I/O multiplexing. To retrieve the file descriptor:

```
doca_event_handle_t doca_verbs_comp_channel_get_handle(
    struct doca_verbs_comp_channel *verbs_comp_channel);
```

This handle can be registered with **epoll** to receive readiness notifications, enabling synchronization across multiple RDMA and non-RDMA events.

5.4.9.4.1.9 4.1.9. DOCA RDMA Verbs Shared Received Queue (SRQ)

A Shared Receive Queue (SRQ) allows multiple RDMA Queue Pairs (QPs) to pull receive buffers from a common pool. Instead of maintaining separate receive queues per QP, the SRQ model supports scalable, efficient buffer sharing—especially useful in high-connection-count scenarios.

When work requests (WRs) are posted to the SRQ, any associated QP can consume them upon receiving data. Receive completions are still reported via the Completion Queue (CQ) attached to the individual QP, not the SRQ.

4.1.9.1. Advantages of SRQ-Based Design

- Scalable Design – Reduces the overhead of managing thousands of per-QP receive queues.
- Efficient Buffer Utilization – Ideal for servers with many clients where only a subset are active simultaneously.
- Lower Memory Footprint – A shared buffer pool minimizes overall memory usage.
- Simplified Buffer Management – Centralized posting reduces CPU and management complexity.

4.1.9.2. SRQ Types

- Linked List SRQ
  - Receive Work Queue (WQ) is organized as a linked list.
  - Supports out-of-order (OOO) processing across different QPs.
  - Supported by DOCA RDMA Verbs data path.
- Contiguous SRQ
  - Receive WQ is managed like a regular per-QP Receive Queue (RQ).
  - Supported only in external data-path mode (e.g., via **doca_dpa**, **doca_gpunetio**).

4.1.9.3. Associating SRQ with a QP

During QP initialization, an SRQ can be linked to a QP to enable shared receive functionality:

```
doca_error_t doca_verbs_qp_init_attr_set_srq(
    struct doca_verbs_qp_init_attr *verbs_qp_init_attr,
    struct doca_verbs_srq *srq);
```

4.1.9.4. SRQ Initialization

An **srq_init_attr** object is used to configure SRQ creation parameters, including:

- Queue size
- Maximum scatter/gather elements per WR
- SRQ type (linked list or contiguous)
- Data-path configuration

To create and configure an SRQ:

```
doca_error_t doca_verbs_srq_init_attr_create(
    struct doca_verbs_srq_init_attr **verbs_srq_init_attr);

doca_error_t doca_verbs_srq_create(
    struct doca_verbs_context *verbs_context,
    struct doca_verbs_srq_init_attr *verbs_srq_init_attr,
    struct doca_verbs_srq **verbs_srq);
```

4.1.9.5. Query SRQ Attributes:

Use **srq_query()** to retrieve the current configuration and parameters of an SRQ:

```
doca_error_t doca_verbs_srq_query(
    struct doca_verbs_srq *verbs_srq,
    struct doca_verbs_srq_init_attr *verbs_srq_init_attr);
```

4.1.9.6. External Data Path Mode

To offload the SRQ data path to an external engine (e.g., **doca_dpa**, **doca_gpunetio**), enable external data-path mode:

```
doca_error_t doca_verbs_srq_init_attr_set_external_datapath_en(
    struct doca_verbs_srq_init_attr *verbs_srq_init_attr,
    uint8_t external_datapath_en);
```

Refer to the Alternative Data Path Options section for further details.

5.4.9.4.1.10 4.1.10. DOCA RDMA Verbs Asynchronous Events

DOCA RDMA Verbs asynchronous events (async events) are non-blocking notifications that inform the application of significant hardware events or error conditions that occur during RDMA operations. These events are essential for detecting issues such as QP errors, memory registration failures, or transport problems.

Async events enable proactive error handling and health monitoring of RDMA resources without interrupting the primary application flow.

4.1.10.1. Retrieving and Acknowledging Events

To receive the next asynchronous event:

```
doca_error_t doca_verbs_get_async_event(
    struct doca_verbs_context *verbs_context,
```

```
    struct doca_verbs_async_event *async_event);
```

Each retrieved event must be acknowledged using the following function to release internal resources and prevent leaks:

```
void doca_verbs_ack_async_event(struct doca_verbs_async_event *async_event);
```

4.1.10.2. Event Handling Best Practices

- Applications should use the event handle obtained from the DOCA RDMA Verbs context to integrate async event monitoring into their event loop (e.g., using **epoll**).
- Async events should be processed promptly to avoid overflows in the internal event queue.
- Acknowledgment is mandatory for each event to ensure proper resource cleanup.

5.4.9.4.1.11 4.1.11. DOCA RDMA Verbs External Resources

DOCA RDMA Verbs supports both DOCA-managed and user-managed (external) modes for managing the low-level hardware resources used in RDMA operations. These include memory regions for work queues, user access regions (UARs), and doorbell registers.

For background on memory management concepts, refer to the DOCA Core Memory Subsystem documentation.

4.1.11.1. Key External Resources

- External UAR (User Access Region) – A memory-mapped region that provides direct access to hardware doorbell registers. UARs are used to signal the RDMA device when new work requests or completions are ready.
- External UMEM (User Memory) – Represents the physical memory region backing the Work Queue (WQ) structures, which hold posted Send and Receive Work Requests (WRs).
- External DBR UMEM (Doorbell Register UMEM) – A dedicated memory region used for doorbell registers that notify the hardware of new entries in the work queues.

4.1.11.2. Usage Modes

Mode	Description
DOCA-managed (default)	DOCA internally allocates and manages all UAR, UMEM, and DBR UMEM resources. The application is abstracted from memory alignment, sizing, and cleanup.
External mode	The application provides and manages its own UAR, UMEM, and DBR UMEM resources. This offers greater flexibility but requires a deep understanding of hardware constraints and careful memory management. External mode responsibilities: • Manual memory allocation with correct alignment and sizing • Lifetime management and cleanup of resources • Compliance with device and queue layout requirements

4.1.11.3. Setting External Resources

To configure external resource pointers during QP initialization:

```
doca_error_t doca_verbs_qp_init_attr_set_external_umem(
    struct doca_verbs_qp_init_attr *verbs_qp_init_attr,
```

```
    struct doca_umem *external_umem,
    uint64_t external_umem_offset);

doca_error_t doca_verbs_qp_init_attr_set_external_dbr_umem(
    struct doca_verbs_qp_init_attr *verbs_qp_init_attr,
    struct doca_umem *external_dbr_umem,
    uint64_t external_dbr_umem_offset);

doca_error_t doca_verbs_qp_init_attr_set_external_uar(
    struct doca_verbs_qp_init_attr *verbs_qp_init_attr,
    struct doca_uar *external_uar);
```

5.4.9.4.2 4.2. IB Verbs Bridge

DOCA RDMA Verbs provides a bridge between the traditional InfiniBand Verbs (IB Verbs) interface and DOCA abstractions, enabling a layered and interoperable architecture for RDMA operations. This bridge allows applications to integrate DOCA's enhanced features while preserving compatibility with the low-level IB Verbs API.

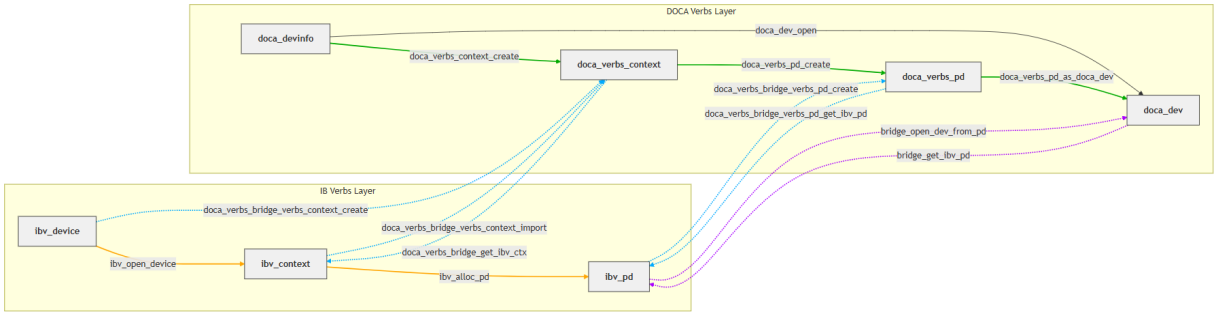
5.4.9.4.2.1 4.2.1. Bridge Configuration Phase

This dual-layer architecture allows applications to mix and match DOCA Verbs APIs and IB Verbs APIs based on their requirements. Developers can, for example, manage resources using DOCA abstractions while executing operations using native IB Verbs data structures.

- The **doca_rdma_verbs_context** is created using **doca_devinfo** and provides the entry point to the DOCA RDMA Verbs layer.
- The underlying **ibv_context** is created from an **ibv_device** using the standard IB Verbs API.
- The bridge components span across several API libraries: **doca_dev**, **doca_verbs**, **doca_verbs_bridge**, **doca_rdma_bridge**, and **libibverbs**.

> The recommended sequence is to first create the DOCA RDMA Verbs context, and then retrieve the DOCA device handle from it—not the other way around.

The following diagram illustrates the relationships between components (not included here, but referenced):



5.4.9.4.2.2 4.2.2. Bridge Execution Phase

The bridge also supports execution-time interoperability. Applications can use IB Verbs work requests (**ibv_send_wr**, **ibv_recv_wr**) and completions (**ibv_wc**) with DOCA-managed QPs and CQs.

4.2.2.1. Posting Work Requests

Use IB Verbs WRs with DOCA RDMA Verbs QPs to submit send or receive operations:

```
int doca_verbs_bridge_post_send(
    struct doca_verbs_qp *verbs_qp,
    struct ibv_send_wr *wr,
    struct ibv_send_wr **bad_wr);

int doca_verbs_bridge_post_recv(
    struct doca_verbs_qp *verbs_qp,
    struct ibv_recv_wr *wr,
    struct ibv_recv_wr **bad_wr);
```

4.2.2.2. Polling Completions

Poll a DOCA RDMA Verbs CQ for completions and retrieve results as an array of **ibv_wc** entries:

```
int doca_verbs_bridge_poll_cq(
    struct doca_verbs_cq *verbs_cq,
    int num_entries,
    struct ibv_wc *ibv_wc);
```

This enables applications to maintain compatibility with existing RDMA workflows while benefiting from DOCA's abstraction, resource management, and hardware integration.

5.4.9.5 5. Alternative Data Path Options

5.4.9.5.1 5.1. DPA Data Path

The DPA Data Path enables offloading RDMA operations from the host CPU to the device (e.g., BlueField DPU), allowing RDMA work to be initiated and completed entirely in the data plane. This mode is especially useful for accelerating networking applications by reducing host CPU involvement.

5.4.9.5.1.1 5.1.1. Configuration Phase

The DPA data path enables execution of RDMA operations directly on the device, bypassing the host CPU. This offload model improves performance and scalability for data-centric applications.

Key differences from host-side (CPU) configuration:

- External data path must be explicitly enabled.
- All RDMA operations, including buffer allocation and queue management, are performed on the device.
- RDMA queues are created to operate entirely within the DPA context.

To create a **doca_verbs_qp** in DPA mode, use one of the following approaches:

- Set DPA context in QP initialization attributes:

  ```
  doca_error_t doca_verbs_qp_init_attr_set_dpa(
      struct doca_verbs_qp_init_attr *verbs_qp_init_attr,
      struct doca_dpa *dpa_ctx);
  ```

  In this mode, external UMEM (for WQ and DBR), external UAR, and PD are created or retrieved internally using the provided DPA context. This is the *simplified* configuration mode, where DOCA manages all required resources.
- Set external UMEM/UAR manually using the standard external resource APIs: In this *advanced* configuration mode, the application allocates external UMEM regions for WQ and

DBR, and provides its own UAR. This offers maximum control over memory layout, alignment, and resource lifecycle.

- Use a DPA completion context instead of a traditional Verbs CQ:

```
doca_error_t doca_verbs_qp_init_attr_set_send_dpa_completion(
    struct doca_verbs_qp_init_attr *verbs_qp_init_attr,
    struct doca_dpa_completion *send_dpa_completion);
```

To use the QP from within the DPA kernel, retrieve the DPA QP handle using:

```
doca_error_t doca_verbs_qp_get_dpa_handle(
    struct doca_verbs_qp *verbs_qp,
    struct doca_dpa *dpa_ctx,
    doca_dpa_dev_verbs_qp_t *dpa_qp_handle);
```

The returned **dpa_qp_handle** can then be used to post send and receive work requests from DPA-side execution flows.

5.4.9.5.1.2 5.1.2. Execution Phase

Refer to section "DOCA DPA Verbs" for information.

5.4.9.5.2 5.2. GPU Data Path

The GPU data path enables RDMA operations to be initiated and completed directly on the GPU, bypassing the host CPU entirely. This mode is essential for accelerating GPU-to-GPU and GPU-to-network communication in high-performance computing and AI workloads.

5.4.9.5.2.1 5.2.1. Configuration Phase

Main differences from host-side (CPU) configuration:

- External data path must be explicitly enabled.
- All RDMA operations, including buffer management, are performed on the device side.
- The application must explicitly allocate GPU memory and bind it to DOCA RDMA Verbs structures.

To create a **doca_verbs_qp** for GPU use:

- Allocate a GPU buffer and use it to create a UMEM object, then assign it to the QP as external UMEM:

```
doca_error_t doca_verbs_qp_init_attr_set_external_umem(
    struct doca_verbs_qp_init_attr *verbs_qp_init_attr,
    struct doca_umem *external_umem,
    uint64_t external_umem_offset);
```

- Allocate another GPU buffer and use it to set an external UAR:

```
doca_error_t doca_verbs_qp_init_attr_set_external_uar(
    struct doca_verbs_qp_init_attr *verbs_qp_init_attr,
    struct doca_uar *external_uar);
```

To create a **doca_verbs_cq** for GPU use:

- Allocate a GPU buffer, wrap it as a UMEM object, and assign it as the external UMEM for the CQ:

```
doca_error_t doca_verbs_cq_attr_set_external_umem(
    struct doca_verbs_cq_attr *cq_attr,
    struct doca_umem *external_umem,
    uint64_t external_umem_offset);
```

- Enable CQ overrun mode, as it is required by most GPU-side data path operations:

```
doca_error_t doca_verbs_cq_attr_set_cq_overrun(
    struct doca_verbs_cq_attr *cq_attr,
    uint8_t overrun);
```

To use the QP on the GPU side, export a handle using DOCA GPUNetIO:

```
doca_error_t doca_gpu_verbs_export_qp(
    struct doca_gpu *gpu_dev,
    struct doca_dev *dev,
    struct doca_verbs_qp *qp,
    enum doca_gpu_dev_verbs_nic_handler nic_handler,
    void *gpu_qp_umem_dev_ptr,
    struct doca_verbs_cq *cq_sq,
    struct doca_verbs_cq *cq_rq,
    struct doca_gpu_verbs_qp **qp_out);
```

The exported **doca_gpu_verbs_qp** handle enables GPU kernels to post send and receive work requests directly from device code, facilitating low-latency, high-throughput RDMA operations within GPU execution contexts.

5.4.9.5.2.2 5.2.2. Execution Phase

Refer to section "GPU Functions – Verbs" for information.

5.4.9.6 6. Samples

5.4.9.6.1 6.1. Verbs Server Client

The **doca_verbs_server_client** sample demonstrates how to perform RDMA verbs operations using DOCA RDMA Verbs for a server-client mechanism. The sample queries the device port and executes the data path according to the link layer configuration (IB or RoCE).

Sample logic:

1. Resource setup:
    a. Creating **verbs_ctx** from device name given as an argument.
    b. Creating **verbs_pd** from **verbs_ctx**.
    c. Creating and setting **cq_attr** to create **verbs_cq**.
    d. Creating **verbs_ah** and setting attributes.
    e. Creating and setting **qp_init_attr** to create **verbs_qp**.
    f. Allocating local memory to execute the RDMA operations using **verbs_pd** and **ibv_reg_mr**.
2. Establishing connection between server and client using TCP socket.
3. Exchanging RDMA parameters: Server and client exchange local buffer addresses, MKEYs, QP numbers and GID/LID addresses

4. QP connection: Setting QP attributes, Modify QP states – RST →INIT → RTR → RTS
5. Data Path Execution: Performing RDMA operations according to user flags:
    a. Send/receive operation –
        - Client side – post send WR (opcode **IBV_WR_RDMA_SEND**), poll CQ completion
        - Server side – post receive WR, poll CQ completion, read the message from the client via local buffer
    b. Write operation –
        - Client side – post send WR (opcode **IBV_WR_RDMA_WRITE** and remote address), poll CQ completion
        - Server side – poll the local buffer and wait for the expected message
    c. Read operation –
        - Client side – post send WR (opcode **IBV_WR_RDMA_READ** and remote address), poll CQ completion
        - Server side – no action needed
6. Syncing completions of local and remote peer via socket message.
7. Verifying completions or received messages.
8. Cleanup: Destroying QP, AH, CQ, PD, Verbs Context, close device, TCP connection and free memory buffers.

Sample usage:

```
Usage: doca_verbs_server_client [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>

Program Flags:
  -c, --client <Sample is client, requires server OOB IP> Sample is client, requires server OOB IP
  -d, --device <IB device name>     IB device name
  -gid, --gid-index                 GID index for DOCA RDMA. Default value 0
  -ro, --rdma-oper-type             RDMA operation type: Send, Write, Read. Default type Send
  -s, --buffer-size                 Buffer size. Default value 64
  -qt, --qp-type                    QP Type: RC. Default type RC
  -n, --iters                       Number of iterations. Default value 1
  -b, --send-burst                  Send all messages at once in a burst
```

Reference:

- **/opt/mellanox/doca/samples/doca_verbs/verbs_server_client/verbs_server_client_sample.c**
- **/opt/mellanox/doca/samples/doca_verbs/verbs_server_client/verbs_server_client_main.c**
- **/opt/mellanox/doca/samples/doca_verbs/verbs_server_client/meson.build**

5.4.9.6.2 6.2. Verbs Wait CQ WR

This sample illustrates how to use the **wait_cq_pi** work request to chain messages and force order of execution.

Sample logic:

1. Locating DOCA device.
2. Initializing required DOCA Core structures.
3. Creating two pairs of QPs, each pair connected between them (QP1, QP2, QP3 and QP4 where QP1 is connected to QP2, QP3 is connected to QP4).
4. Creating three buffers and populate the first with data.

5. The send flows:
    a. QP1 sends data from the first buffer to QP2, which receives the data in the second buffer.
    b. QP3 sends the data from the second buffer to QP4, which receives the data in the third buffer.

    > For the data to arrive properly in the third buffer, the send operation on QP3 must be done only after the send operation on QP1 finishes successfully.

6. Posting recv WRs on QP2 and QP4 to prepare for receiving the data.
7. Posting a **wait_cq_pi** WR on QP3 to wait with the send execution until QP1 finishes sending its data.
8. Posting the send WR on QP3 which is only executed after the **wait_cq_pi** WR finishes waiting.
9. Posting the send WR on QP1 to start the data transfer.
10. Waiting for all WRs to complete.
11. Assuring the data on the third buffer is identical to the data on the first, indicating the WRs were executed in the correct order.

Sample usage:

```
Usage: wait_cq_wr [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>

Program Flags:
  -d, --device-name                 IBV device name to be used.
  -gi, --gid-index                  GID index to use, deafult: 0.
  --ipv6                            Use IPv6 instead of IPv4
```

References:

- **/opt/mellanox/doca/samples/doca_verbs/verbs_wait_cq_wr/verbs_wait_cq_wr_sample.c**
- **/opt/mellanox/doca/samples/doca_verbs/verbs_wait_cq_wr/verbs_wait_cq_wr_main.c**
- **/opt/mellanox/doca/samples/doca_verbs/verbs_wait_cq_wr/meson.build**

5.4.10 DOCA Ethernet

This guide provides an overview and configuration instructions for the DOCA ETH API.

5.4.10.1 1. Overview

> The quality status of DOCA libraries is listed here.

DOCA ETH is a high-performance library for transmitting and receiving Ethernet packets directly on NVIDIA networking devices, with hardware acceleration and flexible queue management.
It is built from two independent components - DOCA ETH RXQ (receive) and DOCA ETH TXQ (transmit) - and runs on either the Linux host or the NVIDIA® BlueField® DPU.

The control path (configuration and queue management) always runs on the CPU, while the data path (the actual packet processing) runs either on the CPU with DOCA ETH or on the GPU with DOCA GPUNetIO.

5.4.10.1.1 1.1. Key Features

- High-performance Ethernet send/receive with direct hardware acceleration.
- Choice of CPU or GPU datapath (GPU via DOCA GPUNetIO) for low-latency packet processing.
- Four receive buffer-management modes to trade off control and performance.
- Rich hardware offloads - LSO, L3/L4 checksum, RX hash, metadata, flow tag, timestamping, CQE compression, and inline data.
- Batching of sends and receives for higher throughput.
- Native packet steering with DOCA Flow and direct hand-off of received packets to other DOCA libraries.

5.4.10.1.2 1.2. High Level Usage Flow

1. Open a device - discover and open a **doca_dev** that supports the modes and offloads you need.
2. Create & configure - create the RXQ/TXQ context and set the mandatory parameters plus any optional offloads.
3. Provide buffers / enable operations - register packet memory (**doca_mmap** / **doca_buf**) or attach a shared mempool and enable the tasks/events you will use.
4. Start - start the context; connect it to a Progress Engine.
5. Steer - use DOCA Flow to route packets to the queue (apply the queue ID).
6. Run the datapath - submit send/receive tasks (or handle events) and poll the Progress Engine; when finished, stop and destroy the context.

5.4.10.2 2. Key Concepts

DOCA ETH builds on standard DOCA Core building blocks. The following are quick reminders - see the linked pages for the full definitions.

Term	In one line	Reference
Context	A configured, startable instance of the library (an RXQ or TXQ) that follows a state machine.	DOCA Core Context
Task	An asynchronous operation you allocate, submit, and later retrieve on completion (e.g., send/receive).	DOCA Core Task
Event	An asynchronous notification delivered through a callback you register (e.g., managed receive).	DOCA Core Event
Progress Engine (PE)	The object you poll to advance tasks and deliver event callbacks on the CPU datapath.	DOCA Core Execution Model
doca_buf	A buffer descriptor that points into mapped memory and holds a packet's data.	DOCA Core Memory Subsystem
doca_mmap	A memory map that registers application memory with the device so it can be used for packets.	DOCA Core Memory Subsystem

Batch (task / event batch)	A single submit/completion or callback that carries many packets, amortizing per-operation overhead for higher throughput.	Execution Phase (this guide)

5.4.10.3 3. Prerequisites & Environment

DOCA ETH follows the DOCA Core Context architecture and can run on either the Linux host or the NVIDIA® BlueField® DPU.

Recommended reading

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem
- DOCA Flow Programming Guide
- DOCA GPUNetIO (for GPU datapath)

Requirements

Before running an application, please ensure the following requirements are met:

- Root privileges – Applications must be run with root privileges.
- Packet steering – DOCA Flow is required to ensure packets are steered correctly to the ETH queues. Refer to the DOCA Flow documentation and the DOCA ETH samples for implementation examples.
- Timestamping – Applications that need to retrieve timestamps in the ETH RXQ must enable the device's real-time clock. This is done using **mlxconfig**:

```
mlxconfig -d <device_id> s REAL_TIME_CLOCK_ENABLE=1
```

5.4.10.4 4. Changes From Previous Releases

5.4.10.4.1 4.1. Changes in 3.5.0

Added

ETH RXQ

CQE Compression

- **doca_error_t doca_eth_rxq_set_cqe_format(struct doca_eth_rxq *eth_rxq, enum doca_eth_rxq_cqe_format format);**
- **doca_error_t doca_eth_rxq_cap_is_cqe_format_supported(const struct doca_devinfo *devinfo, enum doca_eth_rxq_cqe_format format);**

Shared Mempool

- **doca_eth_rxq_set_shared_mempool()**
- **doca_eth_rxq_event_batch_shared_mempool_recv_register()**
- **doca_eth_rxq_shared_mempool_create/destroy()**
- **doca_eth_rxq_shared_mempool_start/stop()**

- `doca_eth_rxq_shared_mempool_allocate_non_library_buffer()`
- `doca_eth_rxq_shared_mempool_free_buf/free_buf_bulk()`
- `doca_eth_rxq_event_batch_shared_mempool_recv_get_ctx/l3_ok/_l4_ok/metadata/flow_tag/rx_hash/timestamp_array()`
- `doca_eth_rxq_shared_mempool_set_headroom_size/tailroom_size/log_pkts_per_wqe_hint/user_extra_size_for_buf/non_library_buffers_count/mmap_cbs()`
- `doca_eth_rxq_shared_mempool_get_headroom_size//tailroom_size/log_pkts_per_wqe_hint/user_extra_size_for_buf/non_library_buffers_count/mmap_cbs()`

ETH TXQ

Inline Data

- `doca_error_t doca_eth_txq_set_max_inline_data_size(struct doca_eth_txq *eth_txq, uint16_t max_inline_data_size);`
- `doca_error_t doca_eth_txq_cap_get_max_inline_data_size(const struct doca_devinfo *devinfo, uint16_t *max_inline_data_size);`

5.4.10.5 5. Architecture

DOCA ETH is comprised of two parts: DOCA ETH RXQ and DOCA ETH TXQ.

5.4.10.5.1 5.1. DOCA ETH RXQ

5.4.10.5.1.1 5.1.1. Operating Modes

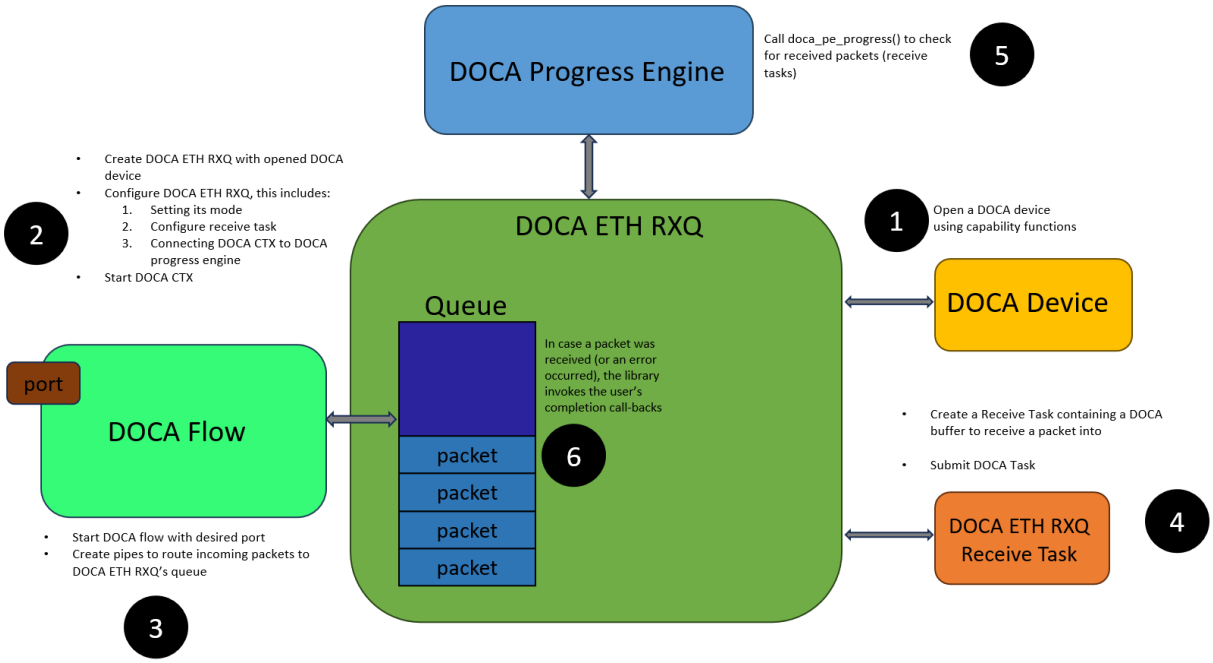
DOCA ETH RXQ can operate in one of four modes, each with a different datapath and buffer-management model. Use the matrix below to pick a mode, then see its short description and data-flow diagram. The mode is selected with `doca_eth_rxq_set_type` (), and support can be checked with `doca_eth_rxq_cap_is_type_supported` ().

Mode	Datapath	Buffer owner	If the app can't keep up	Forward to other DOCA libraries	Best when
Regular Receive	CPU	Application	App-driven - library never overwrites	Yes	You need control over the exact memory placement of each packet.
Cyclic Receive	GPU	Library (cyclic)	Packets may be **overwritten**	No	You want the best performance on GPU with deterministic processing.
Managed Memory Pool Receive	CPU	Library	Pool exhausts → packets **dropped**	Yes	You want high performance on CPU and to forward packets to other DOCA libraries.
Shared Memory Pool Receive	CPU	Library (shared across RXQs)	Pool exhausts → packets **dropped**	Yes	Many RX queues should share one pool at a high packet rate.

5.1.1.1. Regular Receive

> This mode is supported only for CPU datapath.

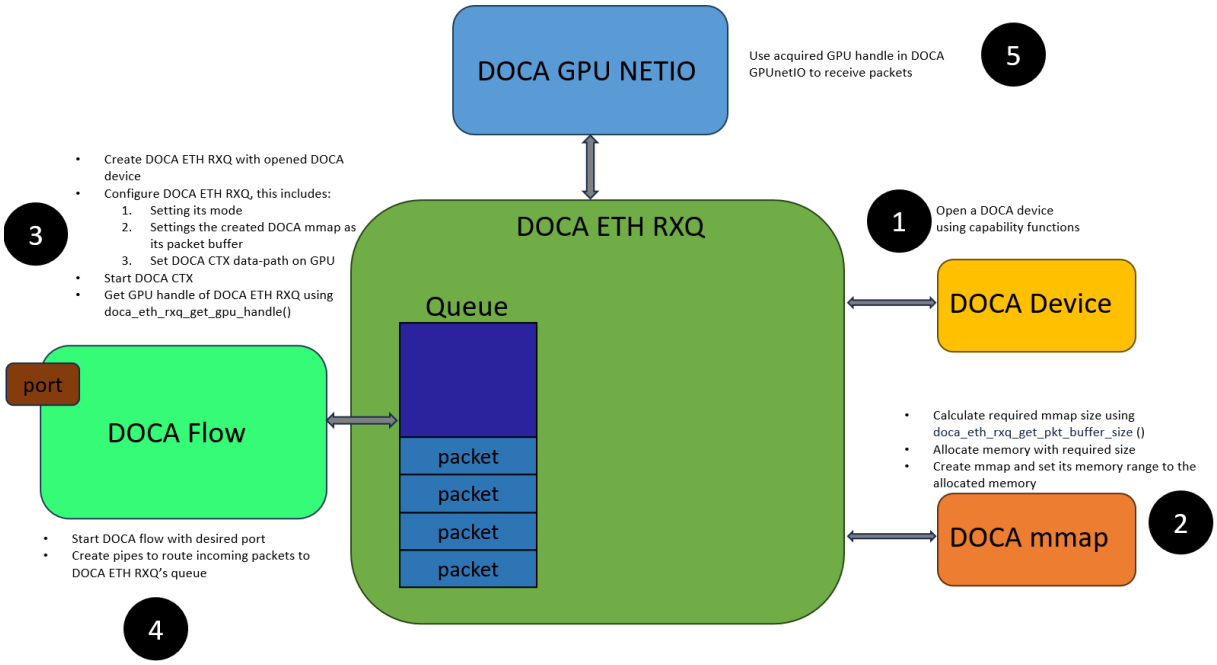
The application owns the packet memory. To receive, you submit a Receive Task containing a **doca_buf** that the library writes the packet into, so you control exactly where each packet lands and can forward it to other DOCA libraries. Delivered via the Receive Task (see Execution Phase).



5.1.1.2. Cyclic Receive

> This mode is supported only for GPU datapath.

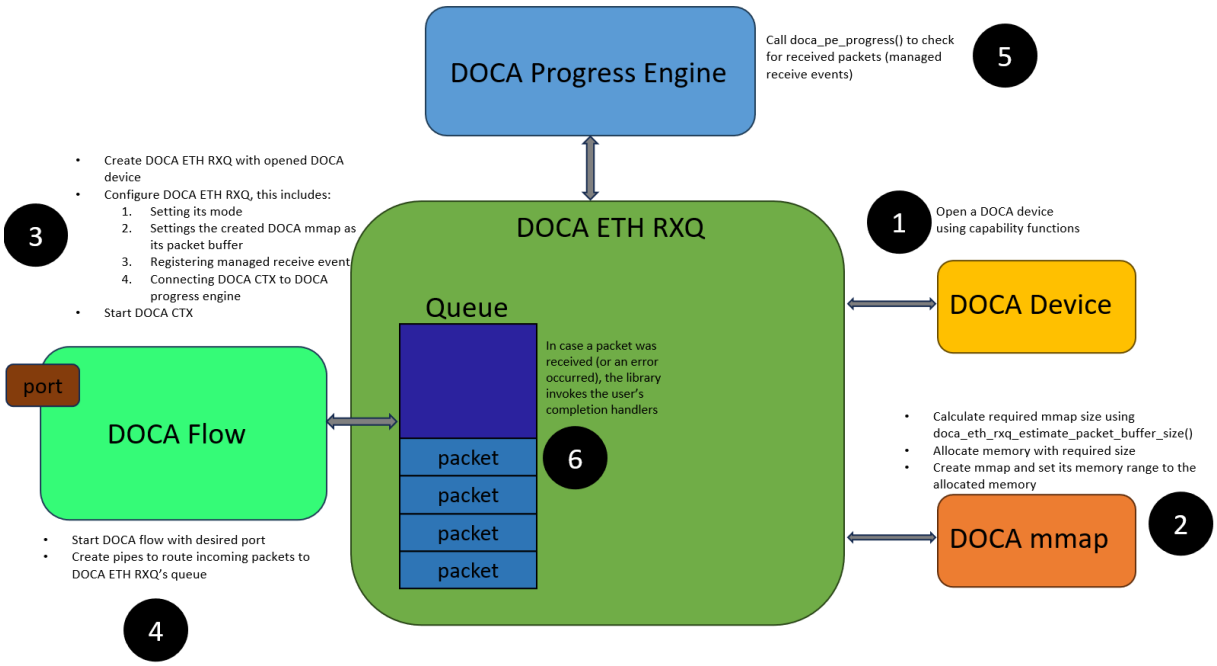
The library scatters packets cyclically into a packet buffer you supply as a **doca_mmap** (set it with **doca_eth_rxq_set_pkt_buf** (), sized with **doca_eth_rxq_estimate_packet_buf_size** ()). An acquired packet may be overwritten if it is not processed in time.
This makes Cyclic Receive best for deterministic, low-latency GPU processing where each packet is consumed before reuse, and it offers the best raw performance (see DOCA GPUNetIO).



5.1.1.3. Managed Memory Pool Receive

> This mode is supported only for CPU datapath.

The library manages the packet buffers (you still supply the backing memory via **doca_eth_rxq_set_pkt_buf** (), sized with **doca_eth_rxq_estimate_packet_buf_size** ()) and will not overwrite an acquired packet until you free it, so packets can be safely forwarded to other DOCA libraries. If buffers are not freed fast enough the pool is exhausted and packets are dropped. Delivered via the Managed Receive Event (see Execution Phase).

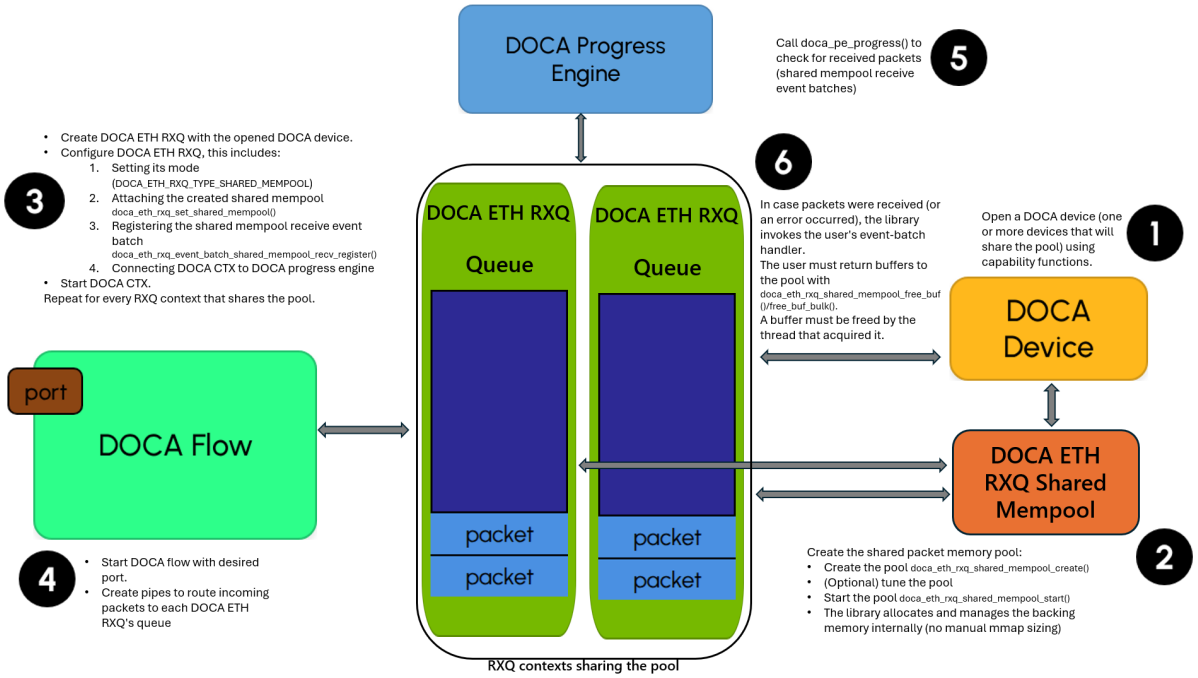


5.1.1.4. Shared Memory Pool Receive

> This mode is supported only for CPU datapath.

Multiple RXQ contexts draw buffers from one shared pool, while sustaining a high packet rate. As in Managed mode, an acquired packet is not overwritten until you free it and can be forwarded to other DOCA libraries.

Create the pool once with **doca_eth_rxq_shared_mempool_create** () and start it with **doca_eth_rxq_shared_mempool_start** (); attach it to each RXQ with **doca_eth_rxq_set_shared_mempool** () and return buffers with **doca_eth_rxq_shared_mempool_free_buf** () (or … **_free_buf_bulk** () for better performance). A buffer must be freed by the thread that acquired it. Delivered via the Shared Mempool Receive Event Batch (see Execution Phase).



5.4.10.5.1.2 5.1.2. Receive Offloads

The DOCA ETH RXQ supports several hardware offloads:

- Metadata – Supports retrieving the metadata value collected during packet flow table processing. (See **doca_eth_rxq_set_metadata_num()**).
- Flow Tag – Supports retrieving the flow tag value that was set by software (e.g., DOCA Flow) when the flow entry was created. (See **doca_eth_rxq_set_flow_tag()**).
- RX Hash – Supports retrieving the calculated RX hash value of the packet. (See **doca_eth_rxq_set_rx_hash()**).
- Packet Headroom/Tailroom - Reserve user-requested space in each packet's **doca_buf** headroom and tailroom. In Managed Memory Pool mode, set it on the RXQ context via **doca_eth_rxq_set_packet_headroom** () and **doca_eth_rxq_set_packet_tailroom** ().

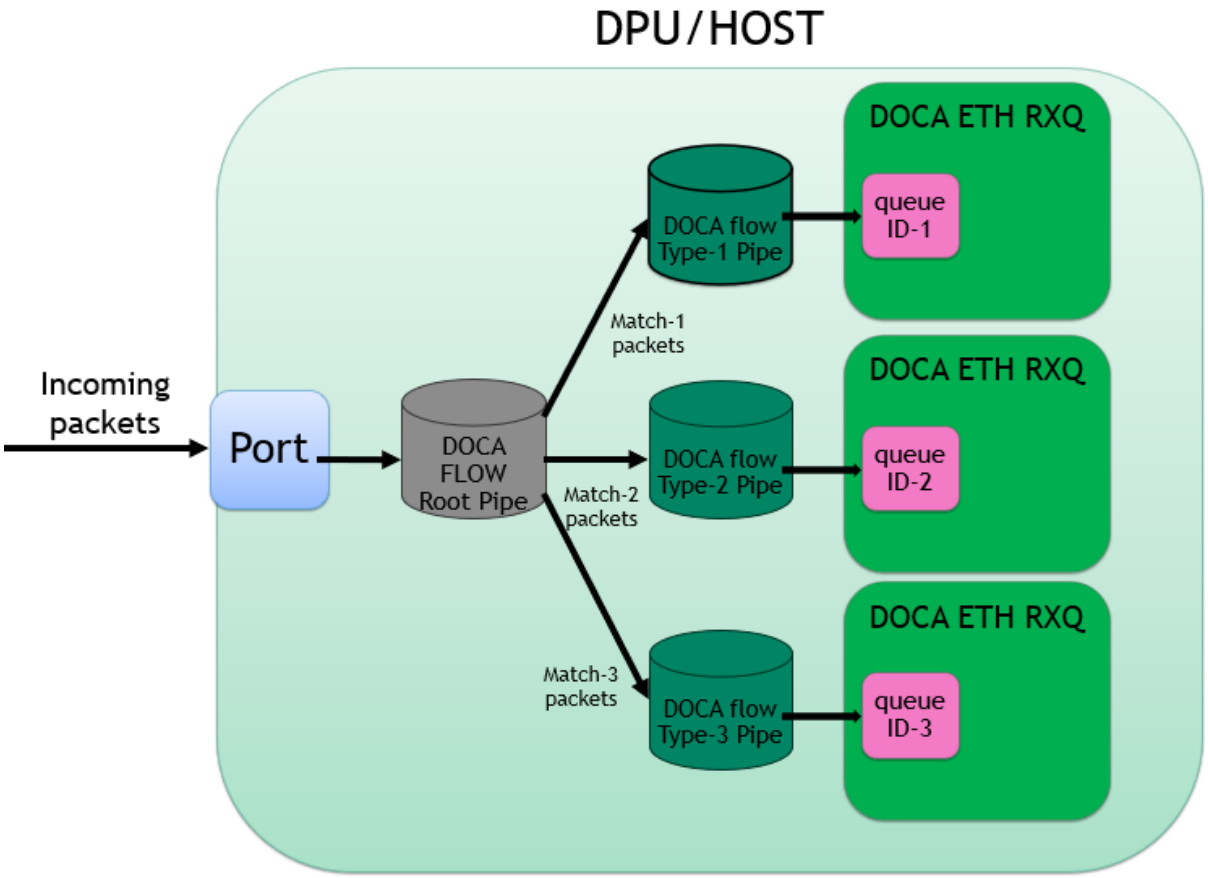
In Shared Memory Pool mode, set it on the pool via `doca_eth_rxq_shared_mempool_set_headroom_size` () and `doca_eth_rxq_shared_mempool_set_tailroom_size` ().

- Packet Timestamp – Supports retrieving the timestamp (time since epoch, in nanoseconds) of when the packet was received. (See `doca_eth_rxq_set_timestamp()`).
- CQE Compression – The hardware can pack multiple completions (CQEs) into a compact format, reducing the PCIe bandwidth consumed by completions and increasing the achievable packet rate (See `doca_eth_rxq_set_cqe_format()`).

5.4.10.5.1.3 5.1.3. Working with DOCA Flow

To route incoming packets to the desired DOCA ETH RXQ, applications need to use DOCA Flow. Applications need to do the following:

- Create and start DOCA Flow on the appropriate port (device)
- Create pipes to route packets into
- Apply unique queue ID of the queue (inside DOCA ETH RXQ) using `doca_eth_rxq_apply_queue_id()`
- Add an entry to a pipe which routes packets into the RX queue (using the queue ID we applied)

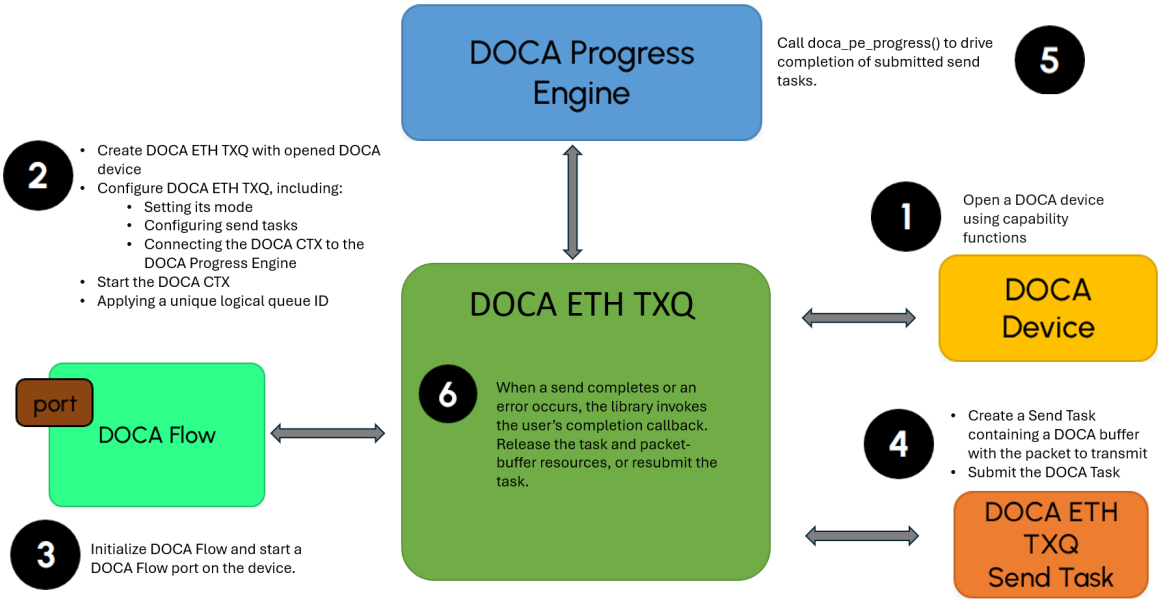


For more details see DOCA ETH RXQ samples and DOCA Flow.

5.4.10.5.2 5.2. 2 DOCA ETH TXQ

5.4.10.5.2.1 5.2.1. 1 Operating Mode

DOCA ETH TXQ operates in a single mode, Regular Send. On the CPU datapath, submit a Send Task containing the packet's **doca_buf** .
For the GPU datapath, see DOCA GPUNetIO.



5.4.10.5.2.2 5.2.2. Transmit Offloads

The DOCA ETH TXQ supports several hardware offloads:

- Large segment offloading (LSO) – The hardware can segment large TCP messages (over IPv4 or IPv6) into multiple valid TCP segments. The application provides a header template, which the hardware automatically replicates and updates (e.g., sequence numbers) for each generated segment (see LSO Send Task).
- L3/L4 checksum offloading – The hardware calculates L3 (IP) and L4 (TCP/UDP) checksums for outgoing IPv4/IPv6 packets - in tunneling scenarios, for the outer header - overwriting any value already present in the checksum fields (see **doca_eth_txq_set_l3_chksum_offload** () and **doca_eth_txq_set_l4_chksum_offload** ()).
- Metadata – Supports attaching a metadata value to the packet, which is then carried into the transmit flow table processing (see **doca_eth_txq_set_metadata_num()**).
- Inline data – For small packets, the hardware can copy the packet data directly into the send queue entry instead of referencing it by pointer, reducing latency. The decision is made per packet according to its length (See **doca_eth_txq_set_max_inline_data_size()**).
- Wait-on-time (scheduled send) - On the GPU datapath, the hardware can transmit a packet at a precise time instead of immediately. Enable it with **doca_eth_txq_set_wait_on_time_offload** (); it requires the device's real-time clock (see

Prerequisites & Environment); compute the send time with `doca_eth_txq_calculate_timestamp` ().

5.4.10.5.2.3 5.2.3. Working with DOCA Flow

To transmit packets through the desired port, applications need to use DOCA Flow. Applications need to:

- Create and start DOCA Flow on the appropriate port (device)
- Apply a unique queue ID to the DOCA ETH TXQ send queue using `doca_eth_txq_apply_queue_id()`

Unlike RXQ, TXQ does not require DOCA Flow pipes or entries because packets are transmitted directly through the selected port.

For more details, see the DOCA ETH TXQ samples and DOCA Flow.

5.4.10.6 6. Configuration Phase

To start using the library, the application first goes through the configuration phase described in DOCA Core Context Configuration Phase. This section covers how to configure and start the context so it can execute tasks and deliver events.
Configure the context to match your use case. To check whether a configuration is supported (and its min/max), see Device Support.

> On the GPU datapath, the context is not associated with a DOCA PE, since the datapath does not run on the CPU.

5.4.10.6.1 6.1. Mandatory Configurations

These configurations are mandatory and must be set by the application before attempting to start the context.

5.4.10.6.1.1 6.1.1. DOCA ETH RXQ

- At least one task/event/event_batch type must be configured. Refer to Receiving Packets (Execution Phase) for more information.
- Max packet size (the maximum size of packet that can be received) must be provided at creation time of the DOCA ETH RXQ context (see `max_packet_size` in `doca_eth_rxq_create()`).
- Max burst size (the maximum number of packets that the library can handle at the same time) must be provided at creation time of the DOCA ETH RXQ context (see `max_burst_size` in `doca_eth_rxq_create()`).
- A device with appropriate support must be provided upon creation (see `dev` in `doca_eth_rxq_create()`).
- When in Cyclic Receive or Managed Memory Pool Receive modes, a `doca_mmap` must be provided in-order write the received packets into (see `doca_eth_rxq_set_pkt_buf()`).

- When in Shared Memory Pool Receive mode, a started `doca_eth_rxq_shared_mempool` must be provided (instead of a packet buffer) using `doca_eth_rxq_set_shared_mempool()`.
- In case of a GPU datapath, A DOCA GPU sub-device must be provided using `doca_ctx_set_datapath_on_gpu()`.

5.4.10.6.1.2 6.1.2. DOCA ETH TXQ

- At least one `task` / `task_batch` type must be configured. Refer to Sending Packets (Execution Phase) for more information.
- Max burst size (the maximum number of packets that the library can handle at the same time) must be provided at creation time of the DOCA ETH TXQ context (see `max_burst_size` in `doca_eth_txq_create()`).
- A device with appropriate support must be provided on creation (see `dev` in `doca_eth_txq_create()`).
- In case of a GPU datapath, a DOCA GPU sub-device must be provided using `doca_ctx_set_datapath_on_gpu()`

5.4.10.6.2 6.2. Optional Configurations

The following configurations are optional. If they are not set, a default value is used.

5.4.10.6.2.1 6.2.1. DOCA ETH RXQ

Setting	API Function	Default Value	Affected Operations
RXQ Mode	`doca_eth_rxq_set_type()`	`Regular Receive`	N/A
Max. Receive Buffer List Length	`doca_eth_rxq_set_max_recv_buf_list_len()`	`1`	Receive Task
Metadata Retrieval	`doca_eth_rxq_set_metadata_num()`	`0` (Disabled)	Receive Task / Managed Receive Event / Managed Receive Event Batch / Shared Mempool Receive Event Batch
Flow Tag Retrieval	`doca_eth_rxq_set_flow_tag()`	Disabled	Receive Task / Managed Receive Event / Managed Receive Event Batch / Shared Mempool Receive Event Batch
RX Hash Retrieval	`doca_eth_rxq_set_rx_hash()`	Disabled	Receive Task / Managed Receive Event / Managed Receive Event Batch / Shared Mempool Receive Event Batch
*Packet Headroom	`doca_eth_rxq_set_packet_headroom()`	`0` (Disabled)	Managed Receive Event / Managed Receive Event Batch
*Packet Tailroom	`doca_eth_rxq_set_packet_tailroom()`	`0` (Disabled)	Managed Receive Event / Managed Receive Event Batch
Timestamp Retrieval	`doca_eth_rxq_set_timestamp()`	Disabled	Receive Task / Managed Receive Event / Managed Receive Event Batch / Shared Mempool Receive Event Batch

CQE Format	`doca_eth_rxq_set_cqe_format()`	`Regular` (Disabled)	Receive Task / Managed Receive Event / Managed Receive Event Batch / Shared Mempool Receive Event Batch

* Note: In Shared Memory Pool Receive mode, packet headroom/tailroom are configured on the pool via `doca_eth_rxq_shared_mempool_set_headroom_size` () / `doca_eth_rxq_shared_mempool_set_tailroom_size` (), not through the RXQ setters above.

5.4.10.6.2.2 6.2.2. DOCA ETH TXQ

Setting	API Function	Default Value	Affected Operations
TXQ Mode	`doca_eth_txq_set_type()`	`Regular Send`	N/A
Max. Send Buffer List Length	`doca_eth_txq_set_max_send_buf_list_len()`	`1`	Send Task / Send Task Batch / LSO Send Task / LSO Send Task Batch
L3 Checksum Offload	`doca_eth_txq_set_l3_chksum_offload()`	Disabled	Send Task / Send Task Batch / LSO Send Task / LSO Send Task Batch
L4 Checksum Offload	`doca_eth_txq_set_l4_chksum_offload()`	Disabled	Send Task / Send Task Batch / LSO Send Task / LSO Send Task Batch
LSO Default MSS	`doca_eth_txq_set_mss()`	`1500`	LSO Send Task / LSO Send Task Batch
LSO Max Header Size	`doca_eth_txq_set_max_lso_header_size()`	`74`	LSO Send Task / LSO Send Task Batch
Metadata Attachment	`doca_eth_txq_set_metadata_num()`	`0` (Disabled)	Send Task / Send Task Batch / LSO Send Task / LSO Send Task Batch
Max Inline Data Size	`doca_eth_txq_set_max_inline_data_size()`	`0` (Disabled)	Send Task / Send Task Batch

5.4.10.6.3 6.3. Device Support

DOCA ETH requires a device to operate. For picking a device, see DOCA Core Device Discovery.
To check if a device supports a specific mode, use the type capabilities functions (see `doca_eth_rxq_cap_is_type_supported()` and `doca_eth_txq_cap_is_type_supported()`).
Devices can allow the following capabilities:

- The maximum burst size
- The maximum buffer chain list (only for Regular Receive/Regular Send modes)
- The maximum packet size (only for DOCA ETH RXQ)
- L3/L4 checksum offloading capability (only for DOCA ETH TXQ)
- Maximum LSO message/header size (only for DOCA ETH TXQ)
- Wait-on-time offloading capability (only for DOCA ETH TXQ in GPU datapath)
- Max metadata number capability (only for CPU datapath)
- CQE compression capability (only for DOCA ETH RXQ in CPU datapath)
- Maximum inline data size (only for DOCA ETH TXQ in CPU datapath)

5.4.10.6.4 6.4. Buffer Support

DOCA ETH supports buffers (`doca_mmap` or `doca_buf`) with the following features:

Buffer Type	Send Task	LSO Send Task	Receive Task	Managed Receive Event	Shared Receive Event
Local mmap buffer	Yes	Yes	Yes	Yes	Yes
Mmap from PCIe export buffer	Yes	Yes	Yes	Yes	Yes
Mmap from RDMA export buffer	No	No	No	No	No
Linked list buffer	Yes	Yes	Yes	No	No

For buffer support in the case of GPU datapath, see DOCA GPUNetIO Programming Guide.

5.4.10.7 7. Execution Phase

This section describes execution on the CPU datapath using the DOCA Core Progress Engine (PE). For the GPU datapath, see GPU Datapath and the DOCA GPUNetIO Programming Guide.

5.4.10.7.1 7.1. Common Behaviour

The following applies to every task and event below:

- You either submit a task and poll the PE for its completion or register an event and the PE invokes your callback when it triggers. See DOCA Core Task and DOCA Core Event.
- Operations are not atomic. Once a buffer is submitted (task) or held by the library (event), do not read or write it until it is completed or returned to you.
- On failure, the context moves to the Stopping state (see State Machine); the buffer *object* itself is not modified.
- Task/event input and output follow the common DOCA Core definitions; only the ETH-specific additions are listed below.

5.4.10.7.2 7.2. Receiving Packets

DOCA ETH RXQ exposes one task and two events. Which one you use is determined by the RXQ mode you selected (see Architecture → Operating Modes).

	Receive Task *(task)*	Managed Receive Event *(event, + batch)*	Shared Mempool Receive Event *(event batch)*
Purpose	Receive into an application-provided buffer (Regular Receive mode).	Receive into a library-managed buffer (Managed Memory Pool mode); not overwritten until freed.	Receive into one pool shared by multiple RXQs (Shared Memory Pool mode) at high rate.
Enable	`doca_eth_rxq_task_recv_set_conf` () (cap: Regular Receive)	`doca_eth_rxq_event_managed_recv_register` () (cap: Managed Memory Pool Receive)	`doca_eth_rxq_event_batch_shared_mempool_recv_register` () (cap: Shared Memory Pool Receive)

Input	A packet `doca_buf` to write into.	— (Library provides the buffer)	— (Library provides the buffers)
On success	Poll the PE → task completes; packet written and data segment extended.	Per-packet success callback; process pkt, then **free it** (return to library). Not freeing causes packet loss.	Batch success callback; process packets, then return each buffer with `doca_eth_rxq_shared_mempool_free_buf` () (or ..._free_buf_bulk()).
On failure	Context → Stopping; `doca_buf` object unchanged (its contents may be modified).	Failure callback; context → Stopping; pkt is NULL.	Batch failure callback; context → Stopping.
Batch variant	— (no receive task batch)	Yes - `doca_eth_rxq_event_batch_managed_recv_register` () delivers multiple packets per callback.	This mechanism is batch-only.

- Reading a received packet – Get the packet's `doca_buf` (`doca_eth_rxq_task_recv_get_pkt` () for the Receive Task, or the buffer handed to your event callback), read it with `doca_buf_get_data` () / `doca_buf_get_data_len` (), then release it: decrement its refcount for a Receive Task, or free it back to the pool for the Managed/Shared events. A shared-mempool buffer must be freed by the same thread that acquired it.
- Per-packet fields (when enabled via the matching Optional Configuration) - L3/L4 checksum result, metadata, flow tag, RX hash, and timestamp - are read with the matching getter family: `doca_eth_rxq_task_recv_get_` * (Receive Task), `doca_eth_rxq_event_managed_recv_get_` * (Managed Event; the batch variant exposes the _array getters), and `doca_eth_rxq_event_batch_shared_mempool_recv_get_` *_array (Shared batch).
- See DOCA ETH Samples for end-to-end Regular Receive and Managed Receive examples.

5.4.10.7.3 7.3. Sending Packets

DOCA ETH TXQ exposes two send tasks; both support a batch variant for higher throughput.

	Send Task *(+ batch)*	**LSO Send Task *(+ batch)***
Purpose	Send a single packet (≤ MTU).	Send a large packet (> MTU); the hardware segments it into MTU-sized packets.
Enable	`doca_eth_txq_task_send_set_conf` () (cap: Regular Send)	`doca_eth_txq_task_lso_send_set_conf` () (cap: Regular Send)
Input	A packet `doca_buf` ; optionally a metadata array and per-packet offload flags (enum `doca_eth_txq_ol_flags`).	A payload `doca_buf` (no headers) + a headers gather list (struct `doca_gather_list`); optionally metadata, offload flags, and a per-packet MSS.
On success	Packet has entered the device's TX hardware and the `doca_buf` is released back to you. **Not** a wire-delivery guarantee.	Same as Send Task; the payload buffer and headers gather list are released.

On failure	Context → Stopping; `doca_buf` object unchanged.	Context → Stopping; payload `doca_buf` and headers `doca_gather_list` unchanged.
Batch variant	Submit an array of buffers (pkt_array); one completion per batch.	Submit a payload array (pkt_payload_array) + headers array (headers_array); one completion per batch.

The task pools can grow at runtime: `doca_eth_txq_task_send_num_expand` () / `doca_eth_txq_task_lso_send_num_expand` ()
(and … `_task_batch_send_num_expand` () / … `_task_batch_lso_send_num_expand` () for batches).
On the GPU datapath, send errors and completions are reported via the Error Send Packet (`doca_eth_txq_gpu_event_error_send_packet_register` ()) and Notify Send Packet (`doca_eth_txq_gpu_event_notify_send_packet_register` ()) events - see GPU Datapath.
See DOCA ETH Samples for Send, LSO Send, and their batch variants.

5.4.10.7.4 7.4. Typical Call Sequence

These are the CPU-datapath call sequences behind the samples.

Transmit (TXQ)

```
/* Prerequisites (see DOCA Core): doca_dev *dev, doca_mmap *mmap,
 * doca_buf_inventory *inv, and doca_pe *pe are already created. */
{
    /* (1) Create & configure - context is Idle here */
    doca_eth_txq_create(dev, max_burst_size, &txq);
    doca_eth_txq_set_type(txq, DOCA_ETH_TXQ_TYPE_REGULAR);
    doca_eth_txq_task_send_set_conf(txq, send_cb, send_cb, task_send_num);
    /* optional offloads, e.g. doca_eth_txq_set_l3_chksum_offload(txq, 1);  */

    /* (2) Start - view TXQ as a doca_ctx, connect to the PE, start */
    ctx = doca_eth_txq_as_doca_ctx(txq);
    doca_pe_connect_ctx(pe, ctx);
    doca_ctx_start(ctx);
    doca_eth_txq_apply_queue_id(txq, queue_id);   /* logical SQ id (after start) */

    /* (3) Steer - start a DOCA Flow port on the device, no pipe/entry needed (see Working with DOCA Flow) */

    /* (4) Run - fill a buffer, allocate a task, submit, poll */
    doca_buf_inventory_buf_get_by_data(inv, mmap, pkt_addr, pkt_len, &pkt);
    doca_eth_txq_task_send_allocate_init(txq, pkt, user_data, &task);
    doca_task_submit(doca_eth_txq_task_send_as_doca_task(task));   /* eth task -> doca_task */
    while (inflight)
        doca_pe_progress(pe);                     /* drives send_cb on completion */

    /* (6) Teardown */
    doca_ctx_stop(ctx);                           /* keep polling doca_pe_progress() until Idle */
    doca_eth_txq_destroy(txq);
}

/* (5) Send completion/error callback - invoked from doca_pe_progress() */
static void send_cb(struct doca_eth_txq_task_send *task, union doca_data task_ud, union doca_data ctx_ud)
{
    doca_eth_txq_task_send_get_pkt(task, &pkt);   /* packet handed back to you */
    doca_buf_dec_refcount(pkt, NULL);             /* release the packet buffer */
    doca_task_free(doca_eth_txq_task_send_as_doca_task(task));
}
```

Receive (RXQ, Regular Receive)

```
/* Prerequisites (see DOCA Core): dev, mmap, inv, pe already set up. */

{
    /* (1) Create & configure - context is Idle here */
    doca_eth_rxq_create(dev, max_burst_size, max_packet_size, &rxq);
    doca_eth_rxq_set_type(rxq, DOCA_ETH_RXQ_TYPE_REGULAR);
    doca_eth_rxq_task_recv_set_conf(rxq, recv_cb, recv_cb, task_recv_num);
    /* optional offloads: doca_eth_rxq_set_metadata_num(), _set_rx_hash(), _set_timestamp() */

    /* (2) Start */
    ctx = doca_eth_rxq_as_doca_ctx(rxq);
    doca_pe_connect_ctx(pe, ctx);
    doca_ctx_start(ctx);
    doca_eth_rxq_apply_queue_id(rxq, queue_id);    /* reuse this id in DOCA Flow */

    /* (3) Steer - route packets to queue_id via DOCA Flow (see Working with DOCA Flow) */
```

```
    /* (4) Run - hand the library an empty buffer, submit, poll */
    doca_buf_inventory_buf_get_by_addr(inv, mmap, buf_addr, buf_len, &buf);
    doca_eth_rxq_task_recv_allocate_init(rxq, buf, user_data, &task);
    doca_task_submit(doca_eth_rxq_task_recv_as_doca_task(task));
    while (running)
        doca_pe_progress(pe);                       /* drives recv_cb per packet */

    /* (6) Teardown - stop DOCA Flow first, then the context */
    doca_ctx_stop(ctx);                             /* keep polling until Idle */
    doca_eth_rxq_destroy(rxq);
}

/* (5) Receive completion/error callback - invoked from doca_pe_progress() */
static void recv_cb(struct doca_eth_rxq_task_recv *task, union doca_data task_ud, union doca_data ctx_ud)
{
    doca_eth_rxq_task_recv_get_pkt(task, &pkt);   /* the received packet */
    doca_buf_get_data(pkt, &data);            /* pointer to packet bytes */
    /* process data; read metadata/rx_hash/timestamp here if enabled     */
    doca_buf_dec_refcount(pkt, NULL);             /* return the buffer */
    doca_task_free(doca_eth_rxq_task_recv_as_doca_task(task));
}
```

For Managed / Shared Memory Pool modes, skip the per-packet Receive Task: register an event/event-batch (see Receiving Packets) and free each delivered buffer after processing.

5.4.10.8 8. State Machine

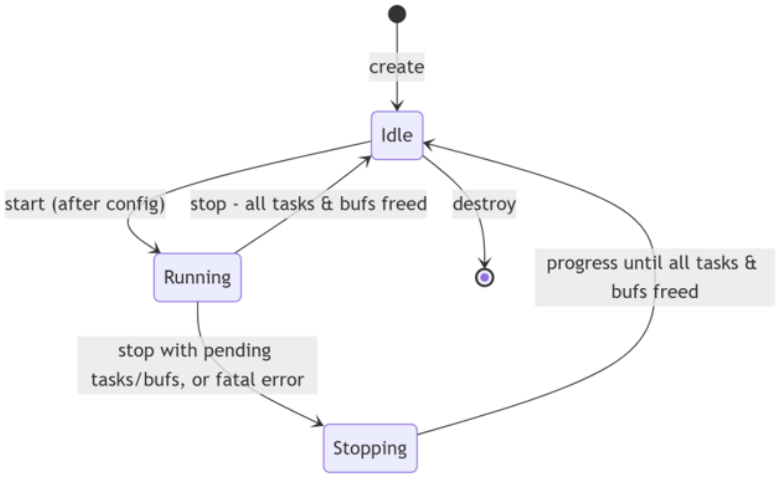
DOCA ETH follows the standard DOCA Core context state machine (see DOCA Core Context State Machine). It uses three of the four states - Idle, Running, and Stopping; the Starting state is not used.

States at a glance

- Idle - Configured but not running. *Allowed:* configure the context, start it, or destroy it.
- Starting - Not used by DOCA ETH (cannot be reached).
- Running - Datapath is active. *Allowed:* allocate and submit tasks, drive completions/events by calling progress, call **`doca_eth_rxq_apply_queue_id`** () to connect an RX queue to DOCA Flow, and call stop.
- Stopping - Draining. *Allowed*: call progress to complete all inflight tasks (which finish with failure), free completed tasks, and free any **`doca_buf`** returned by the Managed or Shared Mempool receive callbacks.

Transitions

From	To	Trigger / condition
(none)	Idle	Create the context
Idle	Running	Call start (after configuration)
Running	Idle	Call stop after all tasks are completed & freed and all **`doca_bufs`** returned by the Managed/Shared Mempool receive callbacks are freed
Running	Stopping	Call stop while tasks or returned **`doca_bufs`** are not yet freed, or a fatal error occurs during progress
Stopping	Idle	Call progress until all tasks are completed & freed and all returned **`doca_bufs`** are freed
Idle	*(destroyed)*	Destroy the context

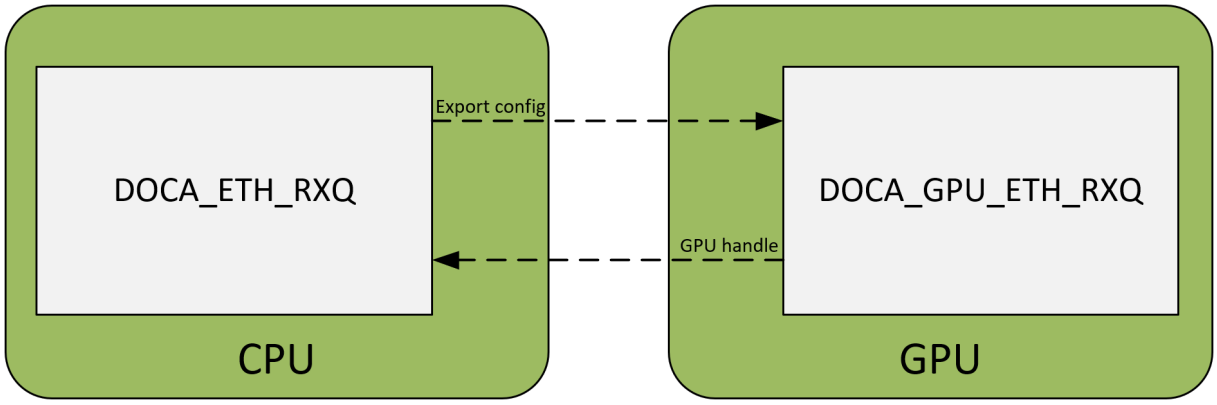


5.4.10.9 9. 9. GPU Datapath

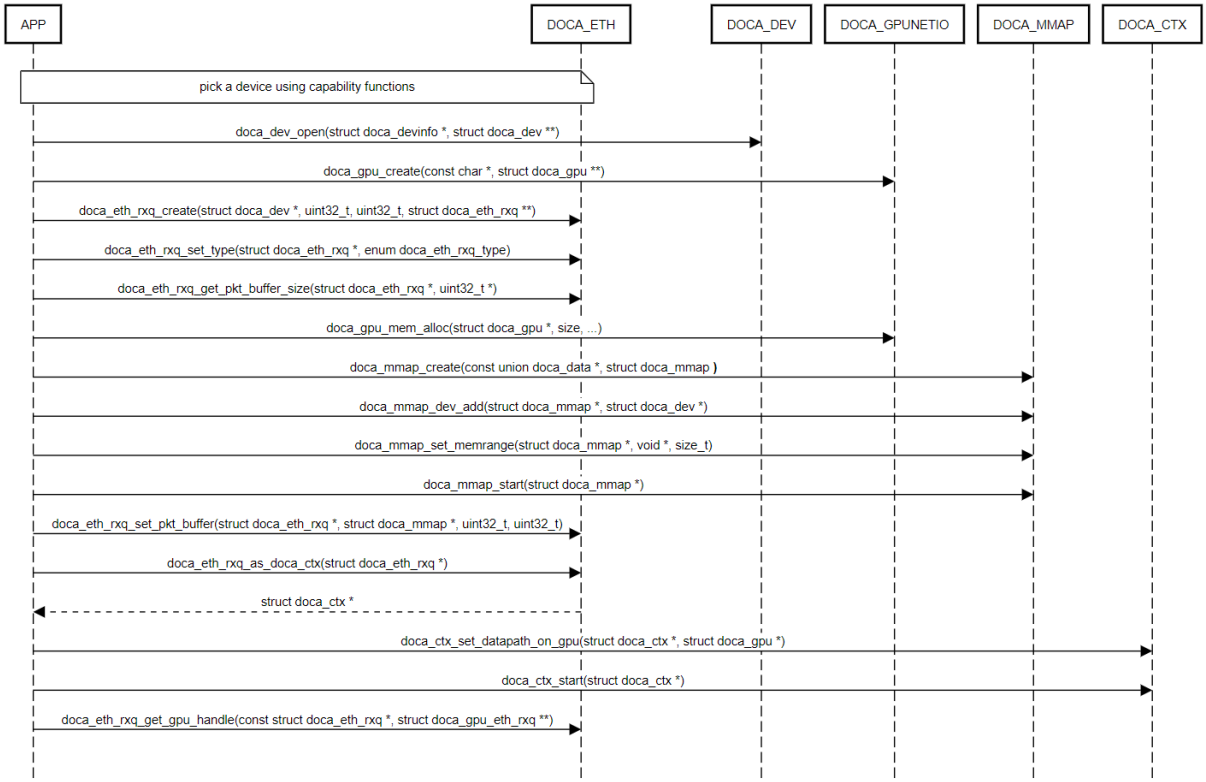
In addition to the CPU datapath (see Execution Phase), DOCA ETH can run its data path on the GPU. This frees the CPU from data-path management and enables low-latency GPU processing of network traffic. The control path (configuration) still runs on the CPU; only the data path moves to the GPU. The data path cannot be managed by the CPU and GPU at the same time.
To set up a GPU-operated context:

1. Create a DOCA GPU device handle.
2. Create the **doca_eth_rxq** / **doca_eth_txq** and configure its parameters.
3. Call **doca_ctx_set_datapath_on_gpu** () before **doca_ctx_start** ().
4. Start the context.
5. Get the GPU handle with **doca_eth_rxq_get_gpu_handle** () (or **doca_eth_txq_get_gpu_handle** () for TX); it points into GPU memory space.

The DOCA ETH context is configured on the CPU and then exported to the GPU:



The following example shows the expected flow for a GPU-managed datapath with packets being scattered to GPU memory (for **doca_eth_rxq**):



Send Events

On the GPU datapath, send outcomes are reported through events you register before **doca_ctx_start** ():

	Error Send Packet	Notify Send Packet
Purpose	Detect a failed packet send	Receive notify/completion info for a sent packet
Register	**doca_eth_txq_gpu_event_error_send_packet_register** ()	**doca_eth_txq_gpu_event_notify_send_packet_register** ()
Handler	Callback fires on a send error; get the failing packet's send-queue index via **doca_eth_txq_gpu_event_error_send_packet_get_position** ().	Callback fires with notify info; get the packet's send-queue index via **doca_eth_txq_gpu_event_notify_send_packet_get_position** ().
Availability	DOCA ETH TXQ on GPU datapath only	DOCA ETH TXQ on GPU datapath only.

For more information regarding the GPU datapath see DOCA GPUNetIO.

5.4.10.10 10. Performance Considerations

The following knobs help you get the most out of DOCA ETH. Measure with your own workload to choose the right combination.

- Batch operations. Use Send/LSO Send Task Batches and Managed/Shared Receive Event Batches to amortize per-operation cost and move many packets per Progress Engine call.
- Right-size max_burst_size. It bounds how many packets the queue handles concurrently; set it to your real in-flight needs.

- **Keep buffer-list length at 1 unless you need scatter/gather.** Longer Max Receive/Send Buffer List Lengths disable some fast paths (e.g., TX inline data).
- **Enable CQE compression on RX** (**doca_eth_rxq_set_cqe_format** ()) to reduce the PCIe bandwidth spent on completions and raise the achievable packet rate (CPU datapath).
- **Use inline data for small sends** (**doca_eth_txq_set_max_inline_data_size** ()) to cut latency by copying small payloads into the work-queue entry (CPU datapath; mutually exclusive with scatter/gather).
- **Use Managed Memory Pool Receive** for higher packet rate.
- **Enable only the offloads you consume.** Flow tag, RX hash, timestamp, and metadata each add per-packet work and are disabled by default.
- **Free received buffers promptly.** Prefer bulk free (**doca_eth_rxq_shared_mempool_free_buf_bulk** ()) to avoid pool exhaustion and packet drops.
- **Tune the Progress Engine polling loop** (tight poll vs. backoff) to match your latency/CPU trade-off.
- **Offload the data path to the GPU** (see GPU Datapath) for the lowest CPU involvement and latency.

5.4.10.11 11. DOCA ETH Samples

This section describes DOCA ETH samples based on the DOCA ETH library. The samples illustrate how to use the DOCA ETH API to do the following:

- Send "regular" packets (smaller than MTU) using DOCA ETH TXQ
- Send "large" packets (larger than MTU) using DOCA ETH TXQ
- Receive packets using DOCA ETH RXQ in Regular Receive mode
- Receive packets using DOCA ETH RXQ in Managed/Shared Memory Pool Receive mode

> All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

- The following samples are for the CPU datapath. For GPU datapath samples, see DOCA GPUNetIO.
- These samples are also available on GitHub.

5.4.10.11.1 11.1. Running the Samples

1. Refer to the following documents:
   a. DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   b. DOCA Troubleshooting for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample (in this case, **eth_txq_send_ethernet_frames**) run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

```
cd /opt/mellanox/doca/samples/doca_eth/eth_txq_send_ethernet_frames
meson /tmp/build
ninja -C /tmp/build
```

The binary **eth_txq_send_ethernet_frames** is created under **/tmp/build/** .

1. Sample (e.g., **eth_txq_send_ethernet_frames**) usage:

```
Usage: doca_eth_txq_send_ethernet_frames [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -d, --device                      IB device name - default: mlx5_0
  -m, --mac-addr                    Destination MAC address to associate with the ethernet frames -
default: FF:FF:FF:FF:FF:FF
```

2. For additional information per sample, use the **-h** option:

```
/tmp/build/<sample_name> -h
```

5.4.10.11.2 11.2. Common Sample Skeleton

All samples follow the end-to-end call sequence in §1.2 (High level usage flow), §7.4 (Typical Call Sequence) and use the operations described in §7.2 (Receiving Packets) and §7.3 (Sending Packets). Only each sample's differences are listed below.

5.4.10.11.3 11.3. TXQ Samples

5.4.10.11.3.1 11.3.1. ETH TXQ Send Ethernet Frames

This sample illustrates how to send a "regular" packet (smaller than MTU) using DOCA ETH TXQ.

Unique to this sample:

- Configure a single Send Task pool with **doca_eth_txq_task_send_set_conf** ().
- Take one **doca_buf** from the inventory over your mmap (**doca_buf_inventory_buf_get_by_data** ()) and write the full Ethernet frame (header + payload) into it.
- Allocate the task with **doca_eth_txq_task_send_allocate_init** (), then **doca_task_submit** ().
- In the completion callback: **doca_eth_txq_task_send_get_pkt** (), check **doca_task_get_status** (), then release the buffer (**doca_buf_dec_refcount** ()) and the task (**doca_task_free** ()).

Reference:

- **/opt/mellanox/doca/samples/doca_eth/eth_txq_send_ethernet_frames/ eth_txq_send_ethernet_frames_sample.c**
- **/opt/mellanox/doca/samples/doca_eth/eth_txq_send_ethernet_frames/ eth_txq_send_ethernet_frames_main.c**
- **/opt/mellanox/doca/samples/doca_eth/eth_txq_send_ethernet_frames/meson.build**

5.4.10.11.3.2 11.3.2. ETH TXQ LSO Send Ethernet Frames

This sample illustrates how to send a "large" packet (larger than MTU) using DOCA ETH TXQ.

Unique to this sample:

- Configure LSO: set the segment size with **doca_eth_txq_set_mss** () and the header-template limit with **doca_eth_txq_set_max_lso_header_size** (); register the LSO pool with **doca_eth_txq_task_lso_send_set_conf** ().
- Split payload and headers: put the payload only in a **doca_buf** from the inventory; put the headers in a **doca_gather_list** node (set its addr/len/next) - the headers are a gather list, not a **doca_buf** .
- Allocate with **doca_eth_txq_task_lso_send_allocate_init** (). The hardware replicates the header template and emits MSS-sized segments.
- In the callback: **doca_eth_txq_task_lso_send_get_pkt_payload** (), check status, release the payload buffer, free the task, and free the headers gather list.

Reference:

- **/opt/mellanox/doca/samples/doca_eth/eth_txq_lso_send_ethernet_frames/eth_txq_lso_send_ethernet_frames_sample.c**
- **/opt/mellanox/doca/samples/doca_eth/eth_txq_lso_send_ethernet_frames/eth_txq_lso_send_ethernet_frames_main.c**
- **/opt/mellanox/doca/samples/doca_eth/eth_txq_lso_send_ethernet_frames/meson.build**

5.4.10.11.3.3 11.3.3. ETH TXQ Batch Send Ethernet Frames

This sample illustrates how to send a batch of "regular" packets (smaller than MTU) with one submit/one completion using DOCA ETH TXQ.

Unique to this sample:

- Configure a batch pool with **doca_eth_txq_task_batch_send_set_conf** (), choosing the max tasks per batch and the number of task batches.
- Build an array of **doca_buf** (one per packet) carved from the mmap.
- Allocate the batch with **doca_eth_txq_task_batch_send_allocate** (), which hands back pkt_array and task_user_data_array; fill pkt_array[i] with each packet buffer (and a per-task user data), then **doca_task_batch_submit** ().
- In the batch callback: loop over tasks_num, check the per-packet status_array[i], read each pkt_array[i], release every buffer, then **doca_task_batch_free** ().

Reference:

- **/opt/mellanox/doca/samples/doca_eth/eth_txq_batch_send_ethernet_frames/eth_txq_batch_send_ethernet_frames_sample.c**
- **/opt/mellanox/doca/samples/doca_eth/eth_txq_batch_send_ethernet_frames/eth_txq_batch_send_ethernet_frames_main.c**
- **/opt/mellanox/doca/samples/doca_eth/eth_txq_batch_send_ethernet_frames/meson.build**

5.4.10.11.3.4 11.3.4. ETH TXQ Batch LSO Send Ethernet Frames

This sample illustrates how to send a batch of "large" packets (larger than MTU) using DOCA ETH TXQ.

Unique to this sample:

- Configure both LSO (**doca_eth_txq_set_mss** (), **doca_eth_txq_set_max_lso_header_size** ()) and the LSO batch pool (**doca_eth_txq_task_batch_lso_send_set_conf** ()).
- Build two arrays: payload **doca_bufs** and header **doca_gather_lists** (one of each per packet).
- Allocate with **doca_eth_txq_task_batch_lso_send_allocate** (), which returns pkt_payload_array, headers_array, and task_user_data_array; populate every entry, then **doca_task_batch_submit** ().
- In the batch callback: loop over tasks_num, check status_array[i], release each payload buffer, free each headers gather list, then **doca_task_batch_free** ().

Reference:

- **/opt/mellanox/doca/samples/doca_eth/eth_txq_batch_lso_send_ethernet_frames/eth_txq_batch_lso_send_ethernet_frames_sample.c**
- **/opt/mellanox/doca/samples/doca_eth/eth_txq_batch_lso_send_ethernet_frames/eth_txq_batch_lso_send_ethernet_frames_main.c**
- **/opt/mellanox/doca/samples/doca_eth/eth_txq_batch_lso_send_ethernet_frames/meson.build**

5.4.10.11.4 11.4. RXQ Samples

5.4.10.11.4.1 11.4.1. ETH RXQ Regular Receive

This sample illustrates how to receive a packet into an application-owned buffer using DOCA ETH RXQ in Regular Receive mode.

Unique to this sample:

- Create the RXQ with both max_burst_size and max_packet_size; set type **DOCA_ETH_RXQ_TYPE_REGULAR** ; register the receive-task pool with **doca_eth_rxq_task_recv_set_conf** (); optionally enable offloads (**doca_eth_rxq_set_metadata_num** (), … **_set_rx_hash** (), … **_set_timestamp** ()).
- After start, apply the queue ID and create a DOCA Flow pipe that steers packets to it.
- Provide the target buffer yourself: take an empty **doca_buf** from the inventory (**doca_buf_inventory_buf_get_by_addr** ()) and submit it with **doca_eth_rxq_task_recv_allocate_init** (); the library writes the packet into it.
- In the completion callback: **doca_eth_rxq_task_recv_get_pkt** (), check status, read the packet (data length, plus any enabled metadata/rx_hash/timestamp), then release the buffer and free the task.

Reference:

- **/opt/mellanox/doca/samples/doca_eth/eth_rxq_regular_receive/eth_rxq_regular_receive_sample.c**

- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_regular_receive/eth_rxq_regular_receive_main.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_regular_receive/meson.build`

5.4.10.11.4.2 11.4.2. ETH RXQ Managed Mempool Receive

This sample illustrates how to receive packets into library-managed buffers using DOCA ETH RXQ in Managed Memory Pool Receive mode.

Unique to this sample:

- Size the pool first: compute the packet-buffer size with `doca_eth_rxq_estimate_packet_buf_size` () and back it with an mmap.
- Set type `DOCA_ETH_RXQ_TYPE_MANAGED_MEMPOOL` ; hand the backing memory to the library with `doca_eth_rxq_set_pkt_buf` (); register the event with `doca_eth_rxq_event_managed_recv_register` (). There is no receive task and no per-packet buffer from the inventory - the library owns the buffers.
- Run by polling the PE; the success callback receives the packet's `doca_buf` directly, exposes per-packet getters (`doca_eth_rxq_event_managed_recv_get_metadata_array` (), ... `_get_rx_hash` (), ... `_get_timestamp` ()), and you must return the buffer with `doca_buf_dec_refcount` () when done - otherwise the pool exhausts and packets drop.
- The error callback is invoked with a NULL packet; query `doca_eth_rxq_event_managed_recv_get_status` ().

Reference:

- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_managed_mempool_receive/eth_rxq_managed_mempool_receive_sample.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_managed_mempool_receive/eth_rxq_managed_mempool_receive_main.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_managed_mempool_receive/meson.build`

5.4.10.11.4.3 11.4.3. ETH RXQ Batch Managed Mempool Receive

This sample illustrates how to receive batches of packets using DOCA ETH RXQ in Managed Memory Pool Receive mode, delivering many packets per callback.

Unique to this sample:

- Register a batch event with `doca_eth_rxq_event_batch_managed_recv_register` (), choosing the max/min events per batch.
- The success callback delivers a whole batch at once: events_number and a `pkt_array` [], with array getters for the per-packet fields (... `_get_metadata_array` (), ... `_get_rx_hash_array` (), ... `_get_timestamp_array` ()).
- Process all packets in the loop, then free the entire array in one call with `doca_eth_rxq_event_batch_managed_recv_pkt_array_free` ().

Reference:

- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_batch_managed_mempool_receive/eth_rxq_batch_managed_mempool_receive_sample.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_batch_managed_mempool_receive/eth_rxq_batch_managed_mempool_receive_main.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_batch_managed_mempool_receive/meson.build`

5.4.10.11.4.4 11.4.4. ETH RXQ Batch Shared Mempool Receive

This sample illustrates how to receive batches of packets using DOCA ETH RXQ in Shared Memory Pool Receive mode, where a single memory pool is shared between multiple DOCA ETH RXQ context.

Unique to this sample:

- Create one shared pool for a configurable number of RXQs with `doca_eth_rxq_shared_mempool_create` (), then `doca_eth_rxq_shared_mempool_start` () - the pool owns the backing memory.
- Set each RXQ to type `DOCA_ETH_RXQ_TYPE_SHARED_MEMPOOL`, attach it with `doca_eth_rxq_set_shared_mempool` (), register a batch event with `doca_eth_rxq_event_batch_shared_mempool_recv_register` (), and apply a unique queue ID.
- The success callback delivers a whole batch (packets_number and `buf_array` []) with array getters (... `_get_metadata_array` (), ... `_get_rx_hash_array` (), ... `_get_timestamp_array` ()).
- Return the whole batch to the pool with `doca_eth_rxq_shared_mempool_free_buf_bulk` () and destroy the RXQ contexts before stopping/destroying the shared pool.

Reference:

- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_batch_shared_mempool_receive/eth_rxq_batch_shared_mempool_receive_sample.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_batch_shared_mempool_receive/eth_rxq_batch_shared_mempool_receive_main.c`
- `/opt/mellanox/doca/samples/doca_eth/eth_rxq_batch_shared_mempool_receive/meson.build`

5.4.11 DOCA GPUNetIO

This document provides an overview and configuration instructions for DOCA GPUNetIO API.

5.4.11.1 Introduction

The quality status of DOCA libraries is listed here.

DOCA GPUNetIO enables real-time GPU processing for network packets, making it ideal for application domains such as:

- Signal processing
- Network security
- Information gathering

- Input reconstruction

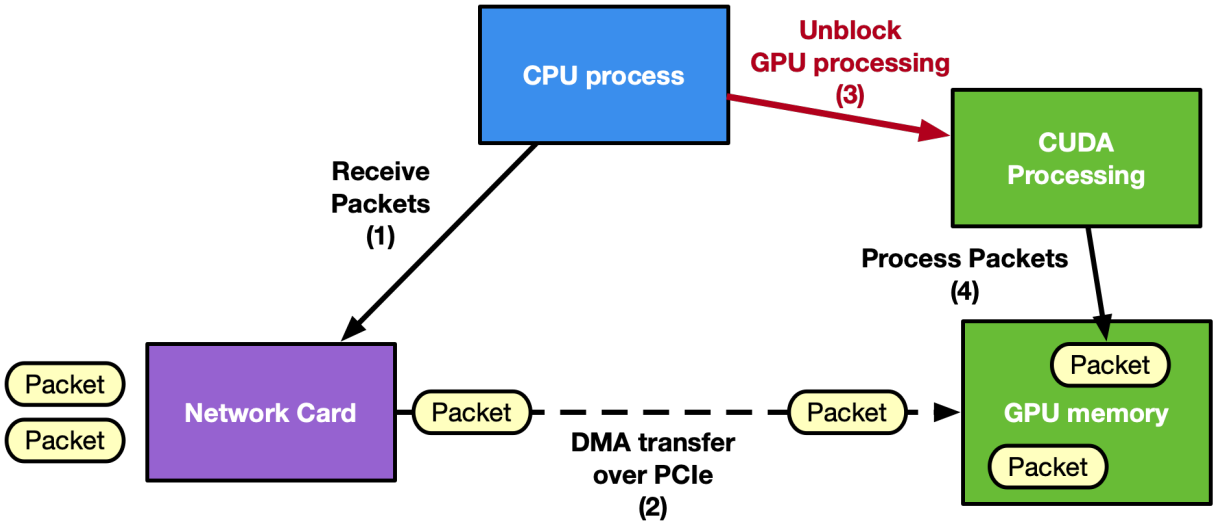
Traditional approaches often rely on a CPU-centric model, where the CPU coordinates with the NIC to receive packets in GPU memory using GPUDirect RDMA. Afterward, the CPU notifies a CUDA kernel on the GPU to process the packets. However, on low-power platforms, this CPU dependency can become a bottleneck, limiting GPU performance and increasing latency.

DOCA GPUNetIO addresses this challenge by offering a GPU-centric solution that removes the CPU from the critical path. By combining multiple NVIDIA technologies, it provides a highly efficient and scalable method for network packet processing.

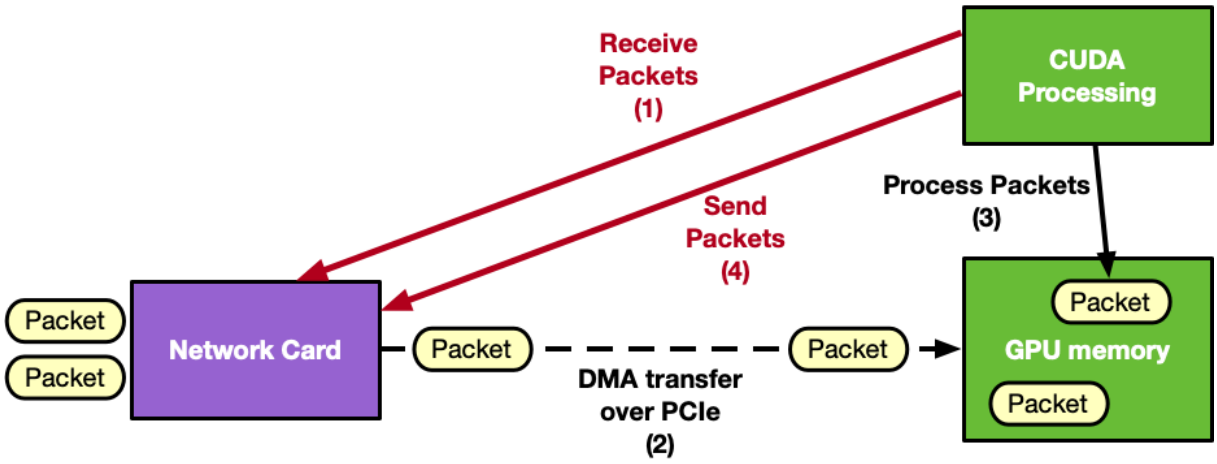
Technologies integrated with DOCA GPUNetIO:

- GPUDirect RDMA – Enables direct packet transfer between the NIC and GPU memory, eliminating unnecessary memory copies
- GPUDirect Async Kernel-Initiated (GDAKI) – Allows CUDA kernels to control network operations without CPU intervention
    - GDAKI is also named IBGDA when used with the RDMA protocol
- GDRCopy Library – Allows the CPU to access GPU memory directly
- NVIDIA BlueField DMA Engine – Supports GPU-triggered memory copies

The following is an example diagram of a CPU-centric approach:



The following is an example diagram of a GPU-centric approach:



Key features of DOCA GPUNetIO include:

- GPUDirect Async Kernel-Initiated (GDAKI)
  - GDAKI network communications – a GPU CUDA kernel can control network communications to send or receive data
    - GPU can control Ethernet communications (Ethernet/IP/UDP/TCP/ICMP)
    - GPU can control RDMA communications (InfiniBand or RoCE are supported)
    - CPU intervention is unnecessary in the application critical path
- GPUDirect RDMA
  - Enables direct data transfers to and from GPU memory without CPU staging copies.
- DMA Engine Control
  - CUDA kernels can initiate memory copies using BlueField's DMA engine
- Semaphores for Low-Latency Communication
  - Supports efficient message passing between CUDA kernels or between CUDA kernels and CPU threads
- Smart Memory Allocation
  - Allocates aligned GPU memory buffers, optimizing memory access
  - GDRCopy library to allocate a GPU memory buffer accessible from the CPU
- Accurate Send Scheduling
  - Provides precise control over Ethernet packet transmission based on user-defined timestamps.

NVIDIA applications that use DOCA GPUNetIO include:

- Aerial 5G SDK – For ultra-low latency 5G network operations
- NIXL – NVIDIA Inference Xfer Library (NIXL) is targeted for accelerating point to point communications in AI inference frameworks (e.g., NVIDIA Dynamo)
- Holoscan Advanced Network Operator – Powering real-time data processing in edge AI environments
- NVQLink – New GPU RoCE transceiver Holoscan sensor bridge operator
- UCX – new GDAKI module via GPUNetIO functions
- NCCL – GIN transport enabled via GPUNetIO GPU communications
- NVSHMEM – GPUNetIO transport for GPU communications replacing the IBGDA transport

For more information about DOCA GPUNetIO, refer to the following NVIDIA blog posts:

- Inline GPU Packet Processing with NVIDIA DOCA GPUNetIO
- Unlocking GPU-Accelerated RDMA with NVIDIA DOCA GPUNetIO
- Realizing the Power of Real-time Network Processing with NVIDIA DOCA GPUNetIO

5.4.11.2 Changes From Previous Releases

5.4.11.2.1 Changes in 3.4.0

- GPUNetIO Verbs:
  - New unreliable BlueFlame mode introduced in **doca_gpu_dev_verbs_submit_bf** function via **DOCA_GPUNETIO_VERBS_GPU_CODE_OPT_BF_UNRELIABLE** code_opt template parameter
  - Improved **gpunetio_verbs_write_lat** example
  - New extended atomic wqe introduced **doca_gpu_dev_verbs_wqe_prepare_atomic_ext**
  - Lighter fence introduced in poll_cq functions **doca_gpu_dev_verbs_fence_acquire_nvidia_nic**
  - Various enhancements to CUDA device **doca_gpu_dev_verbs_get** and related **gpunetIo_verbs_get_bw** example
  - Introduced standalone **doca_gpu_dev_verbs_mcst** function to enable MCST in case of Ampere or older GPU
- All GPUNetIO Verbs Bandwidth examples now returns performance in Gbps

5.4.11.3 General Performance and Best Practices

Adhere to the following guidelines to maximize performance and streamline development with DOCA GPUNetIO.

5.4.11.3.1 Build Configuration

- Use release builds: Always compile applications and samples by setting **buildtype = 'release'** in the **meson.build** file. This ensures maximum throughput and minimal latency compared to the default debug mode.

5.4.11.3.2 Hardware and System Setup

- Driver Selection for **dmabuf**: Install the **nvidia-open** drivers to utilize **dmabuf**. If the environment relies on the closed-source NVIDIA drivers (**cuda-drivers**), the system does not support **dmabuf**; use **nvidia-peermem** instead.
- Optimize PCIe topology: For peak throughput, ensure the GPU and NIC share a PIX or PXB topology (connected via a single PCIe bridge). Avoid NODE or SYS topologies, as routing data across NUMA nodes or SMP interconnects reduces efficiency.
- Expand BAR1 size: If an application fails to map memory buffers to the NIC, verify the GPU's BAR1 size using **nvidia-smi -q**. Enable Resizable BAR in the system BIOS to allocate sufficient space.

- Systems Without GPUDirect RDMA: For hardware that does not support GPUDirect RDMA (such as DGX Spark), use the `DOCA_GPU_MEM_TYPE_CPU_GPU` memory type and enable CPU proxy mode for transmissions.

5.4.11.3.3 Programming Guidelines

- CUDA context initialization: Ensure an active CUDA context exists on the target device before calling DOCA GPUNetIO functions (such as `doca_gpu_create`). Initialize this context effectively by calling `cudaFree(0)`.
- Optimize execution scopes: When utilizing high-level Ethernet APIs, select the widest possible execution scope (Warp or Block) rather than Thread scope. This minimizes contention on atomic operations and significantly increases doorbell ringing efficiency.
- Strict pointer management: Strictly separate memory access boundaries to prevent segmentation faults. Use `memptr_gpu` exclusively within CUDA kernels, and use `memptr_cpu` solely for CPU-side management.

5.4.11.3.4 Permissions

- Root requirements: Execute all Ethernet samples and applications with `sudo` or root privileges, as they rely on DOCA Flow.
- Non-root alternatives: Verbs, RDMA, and DMA operations can run without root access. To enable this functionality, configure the NVIDIA driver with the option `NVreg_RegistryDwords="PeerMappingOverride=1;"`.

5.4.11.4 Navigation Hub

Choose the path that matches your current goal:

Goal	Description	Link
Set up my system	Install packages, configure NIC firmware (ConnectX/BlueField), and verify PCIe topology	Installation and Setup
Learn the concepts	Understand GDAKI, CPU vs. GPU control paths, and the GPUNetIO memory model	Architecture and Design
Start coding	Reference CPU and GPU functions for Ethernet, RDMA, Verbs, and DMA	API Reference
Run a demo	Walk through build instructions and execute provided sample applications	Sample Guide

5.4.11.5 GPUNetIO Installation and Setup

DOCA GPUNetIO is included in the `doca-all` package, which is available from the DOCA downloads portal for all supported operating systems.

To install the required DOCA GPUNetIO components, use the package manager for your OS.

- For Ubuntu/Debian:

```
apt install doca-all doca-sdk-gpunetio libdoca-sdk-gpunetio-dev
```

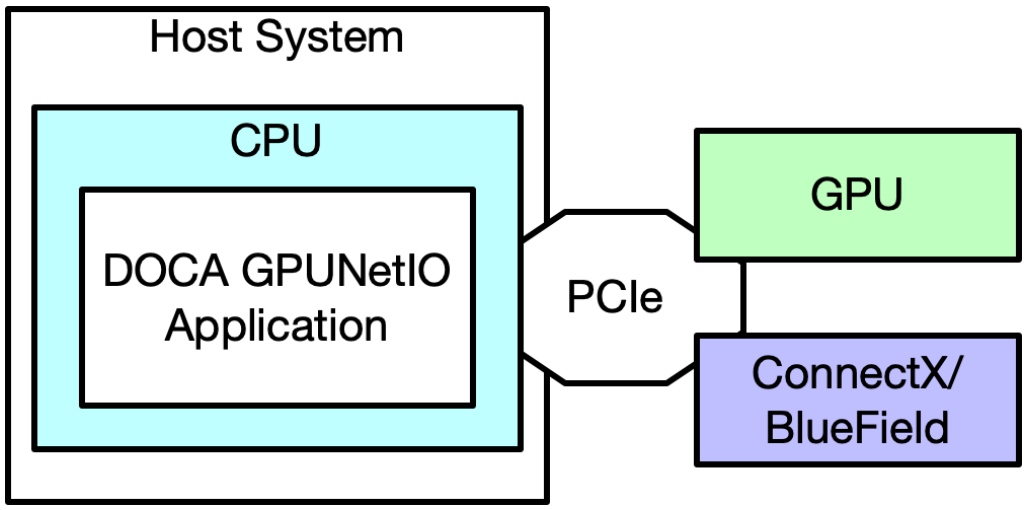
- For RHEL:

```
yum install doca-all doca-sdk-gpunetio doca-sdk-gpunetio-devel
```

Build Type and Performance

To achieve the best performance when building any DOCA GPUNetIO sample or application, you must set the `buildtype` to `release` in the `meson.build` file (e.g., `buildtype = 'release'`). Building in the default `debug` mode will result in significantly lower performance.

To run a DOCA GPUNetIO application, the system must be configured with both a GPU and a NIC (either NVIDIA® ConnectX® or NVIDIA® BlueField®), connected to the system via PCIe.



The system's internal hardware topology should be GPUDirect-RDMA-friendly to maximize the internal throughput between the GPU and the NIC. To verify the type of connection between the GPU and NIC:

```
$ nvidia-smi topo -m
        GPU0    NIC0    NIC1    CPU Affinity    NUMA Affinity   GPU NUMA ID
GPU0     X      NODE    NODE    12-23,36-47     1               N/A
NIC0    NODE     X      PIX
NIC1    NODE    PIX      X

Legend:

  X    = Self
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
  PXB  = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge)
  PIX  = Connection traversing at most a single PCIe bridge
  NV#  = Connection traversing a bonded set of # NVLinks

NIC Legend:

  NIC0: mlx5_0
  NIC1: mlx5_1
```

To maximize throughput between the GPU and NIC, the system should have a PIX (or PXB) topology with a dedicated PCIe connection. A PHB topology is still acceptable if the GPU and NIC are on the same PCIe Host Bridge and NUMA node, although performance may vary depending on the platform. For optimal performance, it's recommended to avoid NODE and SYS topologies, as they may negatively impact performance despite the application remaining functional.

Environment Support

DOCA GPUNetIO has been fully tested on bare-metal systems and within Docker containers. Support for virtualized environments is currently considered experimental.

5.4.11.5.1 NIC Configuration

5.4.11.5.1.1 ConnectX NIC

Ensure the ConnectX firmware is compatible with the current DOCA release. NVIDIA recommends using ConnectX-6 Dx or later adapters.

1. Start MST:

```
$ sudo mst start
```

2. Check MST status to get the MST device identifier:

```
$ sudo mst status -v
```

Example output:

```
MST modules:
------------
    MST PCI module is not loaded
    MST PCI configuration module loaded
PCI devices:
------------
DEVICE_TYPE             MST                           PCI       RDMA            NET                       
NUMA
ConnectX6DX(rev:0)      /dev/mst/mt4125_pciconf0.1    b5:00.1   mlx5_1          net-ens6f1                0
ConnectX6DX(rev:0)      /dev/mst/mt4125_pciconf0      b5:00.0   mlx5_0          net-ens6f0                0
```

3. Configure ConnectX NIC:
   - For Ethernet transport, run the following commands, replacing `<mst_device>` with the actual MST device name (e.g., `/dev/mst/mt4125_pciconf0`):

```
mlxconfig -d <mst_device> s KEEP_ETH_LINK_UP_P1=1 KEEP_ETH_LINK_UP_P2=1 KEEP_IB_LINK_UP_P1=0
KEEP_IB_LINK_UP_P2=0
# This is required only if application uses the Accurate Send Scheduling feature
mlxconfig -d <mst_device> --yes set ACCURATE_TX_SCHEDULER=1 REAL_TIME_CLOCK_ENABLE=1
```

     The following example assumes that the adapter is dual port. If single port, only P1 options apply.

   - For InfiniBand transport, run:

```
mlxconfig -d <mst_device> s KEEP_ETH_LINK_UP_P1=0 KEEP_ETH_LINK_UP_P2=0 KEEP_IB_LINK_UP_P1=1
KEEP_IB_LINK_UP_P2=1
```

```
# Accurate Send Scheduling feature can't be used with InfiniBand
```

> The following example assumes that the adapter is dual port. If single port, only P1 options apply.

4. Perform a cold reboot to apply the changes:

```
ipmitool power cycle
```

5.4.11.5.1.2 BlueField NIC

To use NVIDIA BlueField-2 or BlueField-3 with DOCA GPUNetIO, the DPU must be in NIC mode to expose the internal ConnectX to the host application.

1. Start MST:

```
$ sudo mst start
```

2. Check MST status to get the mst device identifier:

```
$ sudo mst status -v
```

Example output:

```
MST modules:
------------
    MST PCI module is not loaded
    MST PCI configuration module loaded
PCI devices:
------------
DEVICE_TYPE                MST                           PCI       RDMA            NET
NUMA
BlueField3(rev:1)          /dev/mst/mt41692_pciconf0.1   9f:00.1   mlx5_1          net-ens6f1np1
1
BlueField3(rev:1)          /dev/mst/mt41692_pciconf0     9f:00.0   mlx5_0          net-ens6f0np0
1
```

3. Configure BlueField NIC:
   - For Ethernet transport:

```
sudo mlxconfig -d /dev/mst/mt41692_pciconf0 --yes set LINK_TYPE_P1=2 LINK_TYPE_P2=2
 INTERNAL_CPU_MODEL=1 INTERNAL_CPU_PAGE_SUPPLIER=1 INTERNAL_CPU_ESWITCH_MANAGER=1
 INTERNAL_CPU_IB_VPORT0=1 INTERNAL_CPU_OFFLOAD_ENGINE=DISABLED
# This is required only if application uses the Accurate Send Scheduling feature
sudo mlxconfig -d /dev/mst/mt41692_pciconf0 --yes set ACCURATE_TX_SCHEDULER=1
 REAL_TIME_CLOCK_ENABLE=1
```

   - For InfiniBand transport:

```
sudo mlxconfig -d /dev/mst/mt41692_pciconf0 --yes set LINK_TYPE_P1=1 LINK_TYPE_P2=1
 INTERNAL_CPU_MODEL=1 INTERNAL_CPU_PAGE_SUPPLIER=1 INTERNAL_CPU_ESWITCH_MANAGER=1
 INTERNAL_CPU_IB_VPORT0=1 INTERNAL_CPU_OFFLOAD_ENGINE=DISABLED
# Accurate Send Scheduling feature can't be used with Infiniband
```

4. Do a cold reboot to apply the changes:

```
ipmitool power cycle
```

5. Example verification command for Ethernet.

```
sudo mlxconfig -d /dev/mst/mt41692_pciconf0 q LINK_TYPE_P1 LINK_TYPE_P2 INTERNAL_CPU_MODEL
INTERNAL_CPU_PAGE_SUPPLIER INTERNAL_CPU_ESWITCH_MANAGER INTERNAL_CPU_IB_VPORT0 INTERNAL_CPU_OFFLOAD_ENGINE
ACCURATE_TX_SCHEDULER REAL_TIME_CLOCK_ENABLE
```

Example output (Ethernet):

```
        LINK_TYPE_P1                                ETH(2)
        LINK_TYPE_P2                                ETH(2)
        INTERNAL_CPU_MODEL                          EMBEDDED_CPU(1)
        INTERNAL_CPU_PAGE_SUPPLIER                  EXT_HOST_PF(1)
        INTERNAL_CPU_ESWITCH_MANAGER                EXT_HOST_PF(1)
        INTERNAL_CPU_IB_VPORT0                      EXT_HOST_PF(1)
        INTERNAL_CPU_OFFLOAD_ENGINE                 DISABLED(1)
        ACCURATE_TX_SCHEDULER                       True(1)
        REAL_TIME_CLOCK_ENABLE                      True(1)
```

5.4.11.5.2 PCIe Configuration

On some x86 systems, the Access Control Services (ACS) must be disabled to ensure direct communication between the NIC and GPU, whether they reside on the same converged accelerator DPU or on different PCIe slots in the system. The recommended solution is to disable ACS control via BIOS (e.g., Supermicro or HPE) on your PCIe bridge. Alternatively, it is also possible to disable it via command line, but it may not be as effective as the BIOS option.

The following `lspci -tvvv` output illustrates a typical system topology:

```
$ lspci -tvvv...+-[0000:b0]-+-00.0  Intel Corporation Device 09a2
 |           +-00.1  Intel Corporation Device 09a4
 |           +-00.2  Intel Corporation Device 09a3
 |           +-00.4  Intel Corporation Device 0998
 |           \-02.0-[b1-b6]----00.0-[b2-b6]--+-00.0-[b3]--+-00.0  Mellanox Technologies MT42822 BlueField-2
integrated ConnectX-6 Dx network controller
 |                                          |            +-00.1  Mellanox Technologies MT42822 BlueField-2
integrated ConnectX-6 Dx network controller
 |                                          |            \-00.2  Mellanox Technologies MT42822 BlueField-2 SoC
Management Interface
 |                                         \-01.0-[b4-b6]----00.0-[b5-b6]----08.0-[b6]----00.0  NVIDIA
Corporation Device 20b8
```

The PCIe switch address to consider is **b2:00.0** (entry point of the DPU). ACSCtl must have all negative values:

PCIe set

```
setpci -s b2:00.0 ECAP_ACS+0x6.w=0000
```

To verify that the setting has been applied correctly:

PCIe check

```
$ sudo lspci -s b2:00.0 -vvvv | grep -i ACSCtl
ACSCtl: SrcValid- TransBlk- ReqRedir- CmpltRedir- UpstreamFwd- EgressCtrl- DirectTrans-
```

Please refer to this page and this page for more information.

If the application still does not report any received packets, try to disable IOMMU. On some systems, it can be done from the BIOS looking for the the **VT-d** or **IOMMU** from the NorthBridge configuration and change that setting to **Disable** and save it. The system may also require adding **intel_iommu=off** or **amd_iommu=off** to the kernel options. That can be done through the grub command line as follows:

IOMMU

```
$ sudo vim /etc/default/grub
# GRUB_CMDLINE_LINUX_DEFAULT="iommu=off intel_iommu=off <more options>"
$ sudo update-grub
$ sudo reboot
```

5.4.11.5.3 GPU Configuration

5.4.11.5.3.1 CUDA Dependency

DOCA GPUNetIO components have a dependency on CUDA. These dependencies differ for the CPU-side shared library versus the GPU-side datapath components.

- CPU Shared Library (**libdoca_gpunetio.so**) This library has a dependency on **libcuda.so** (CUDA Driver API). Because it does not use the CUDA Runtime API, it is not subject to potential versioning issues associated with the runtime.
- GPU Datapath Components The data path functions are delivered as both header files and a static library, which have different requirements:
  - Header-only APIs (GPUNetIO Ethernet, GPUNetIO Verbs): These are inlined functions. Since they are compiled with your application, they are flexible and can be used with any recent CUDA version (e.g., CUDA 12.x or 13.x).
  - Static Library APIs (GPUNetIO DMA, CommCh, RDMA): This library is pre-built with CUDA 13.0. Therefore, any application using functions from this static library must be built with CUDA 13.0 or newer.

It is generally recommended to use CUDA 12.6 or newer wherever possible to take advantage of new features.

To decrease initial application startup latency, it is highly recommended to enable NVIDIA driver persistence mode:

```
nvidia-smi -pm 1
```

5.4.11.5.3.2 GDRCopy Installation

To enable direct CPU access to GPU memory without using CUDA APIs, DOCA requires the GDRCopy kernel module and library.

1. Install necessary packages:

   ```
   sudo apt install -y check kmod
   ```

2. Clone the GDRCopy repository:

   ```
   git clone https://github.com/NVIDIA/gdrcopy.git /opt/mellanox/gdrcopy
   ```

3. Build GDRCopy:

   ```
   cd /opt/mellanox/gdrcopy && make
   ```

4. Load the GDRCopy kernel module:

```
./insmod.sh
```

5. Check if the `gdrdrv` and `nvidia-peermem` modules are loaded:

```
lsmod | egrep gdrdrv
```

Example output:

```
gdrdrv                 24576  0
nvidia              55726080  4 nvidia_uvm,nvidia_peermem,gdrdrv,nvidia_modeset
```

6. Export the GDRCopy library path:

```
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/opt/mellanox/gdrcopy/src
```

7. Ensure CUDA library paths are in the environment variables:

```
export PATH="/usr/local/cuda/bin:${PATH}"
export LD_LIBRARY_PATH="/usr/local/cuda/lib:/usr/local/cuda/lib64:${LD_LIBRARY_PATH}"
export CPATH="$(echo /usr/local/cuda/targets/{x86_64,sbsa}-linux/include | sed 's/ /:/'):${CPATH}"
```

GDRCopy is optional. If not installed, DOCA GPUNetIO cannot allocate memory using the `DOCA_GPU_MEM_TYPE_GPU_CPU` flag. If GDRCopy is not detected, DOCA GPUNetIO will log warning messages.

If GDRCopy is not required for your application, you can safely ignore the related warning messages. To use GDRCopy, ensure its installation path is included in the `LD_LIBRARY_PATH` environment variable or specified using the `GDRCOPY_PATH_L` environment variable.

5.4.11.5.3.3 GPU Memory Mapping

To enable the NIC to send and receive packets using GPU memory, a memory mapping mechanism must be used. DOCA supports two methods:

1. **dmabuf** (default method): The preferred, modern method for mapping GPU memory.
2. **nvidia-peermem** (fallback method): A legacy method used if **dmabuf** is not available or fails.

Using dmabuf

This is the primary method for mapping GPU memory. The prerequisites for this approach are:

- Linux Kernel version 6.2 or later
- **libibverbs** version 1.14.44 or later
- CUDA Toolkit:
  - Version 12.5 or older: Must be installed with the **-m=kernel-open** flag (implying open-source NVIDIA driver mode).
  - Version 12.6 or newer: Open kernel mode is enabled by default.

dmabuf NVIDIA driver

Please ensure your system has `nvidia-open` drivers installed. If it shows `cuda-drivers` instead, it means the NVIDIA driver is installed as close source version and then dmabuf can't be used.

Using nvidia-peermem

This method is used if `dmabuf` is unavailable. It requires the `nvidia-peermem` kernel module, which is installed with the CUDA Toolkit, to be loaded:

Launch nvidia-peermem

```
sudo modprobe nvidia-peermem
```

Implementation and Fallback Logic

The recommended implementation is to attempt to get a `dmabuf` file descriptor first. If that fails, the application should fall back to the `nvidia-peermem` method.

The following code snippet demonstrates how to use `dmabuf` for GPU memory mapping with DOCA mmap, including the fallback logic:

GPU Configuration

```
/* Get the dmabuf file-descriptor for the GPU memory buffer from CUDA */
result = doca_gpu_dmabuf_fd(gpu_dev, gpu_buffer_addr, gpu_buffer_size, &(dmabuf_fd));

if (result != DOCA_SUCCESS) {
    /* Fallback to nvidia-peermem legacy method if dmabuf fails */
    doca_mmap_set_memrange(gpu_buffer_mmap, gpu_buffer_addr, gpu_buffer_size);
} else {
    /* Create DOCA mmap using dmabuf */
    doca_mmap_set_dmabuf_memrange(gpu_buffer_mmap, dmabuf_fd, gpu_buffer_addr, 0, gpu_buffer_size);
}
```

Handling dmabuf Failure

A failure in `doca_gpu_dmabuf_fd` (the `if` block in the example) likely indicates that the NVIDIA driver is not in open-source mode.

When `doca_mmap_start` is subsequently called, DOCA will attempt to map the GPU memory. If `dmabuf` was not set, it will automatically fall back to the legacy `nvidia-peermem` method. In this case, the following warning message is logged:

GPU Configuration

```
[DOCA][WRN][linux_devx_adapter.cpp:374] devx adapter 0x5566a16018e0: Registration using dmabuf is not supported,
falling back to legacy registration
```

If your application can rely on `nvidia-peermem` and does not strictly require `dmabuf`, this warning message can be safely ignored.

Sample Implementations

- GPUNetIO Ethernet samples use DOCA mmap with **dmabuf** and **nvidia-peermem** as the fallback (following the logic in the code example above).
- GPUNetIO Verbs samples show an alternative verbs-based method, using **ibv_reg_dmabuf_mr** (for **dmabuf**) and **ibv_reg_mr** (as the fallback).

5.4.11.5.3.4 GPU BAR1 Size

Every time a GPU buffer is mapped to the NIC (e.g., buffers associated with send or receive queues), a portion of the GPU BAR1 mapping space is used. Therefore, it is important to check that the BAR1 mapping is large enough to hold all the bytes the DOCA GPUNetIO application is trying to map. To verify the BAR1 mapping space of a GPU you can use **nvidia-smi**:

BAR1 mapping

```
$ nvidia-smi -q

==============NVSMI LOG==============
.....
Attached GPUs                             : 1
GPU 00000000:CA:00.0
    Product Name                          : NVIDIA A100 80GB PCIe
    Product Architecture                  : Ampere
    Persistence Mode                      : Enabled
.....
    BAR1 Memory Usage
        Total                             : 131072 MiB
        Used                              : 1 MiB
        Free                              : 131071 MiB
```

By default, some GPUs (e.g. RTX models) may have a very small BAR1 size:

BAR1 mapping

```
$ nvidia-smi -q | grep -i bar -A 3
    BAR1 Memory Usage
    Total : 256 MiB
    Used : 6 MiB
    Free : 250 MiB
```

If the BAR1 size is not enough, DOCA GPUNetIO applications may exit with errors because DOCA mmap fails to map the GPU memory buffers to the NIC (e.g., **Failed to start mmap DOCA Driver call failure**). To overcome this issue, the GPU BAR1 must be increased from the BIOS. The system should have "Resizable BAR" option enabled. For further information, refer to this NVIDIA forum post.

5.4.11.5.3.5 Running without Root Privileges

All DOCA GPUNetIO samples and applications using Ethernet rely on DOCA Flow. Therefore, they must be executed with **sudo** or root privileges.

However, Verbs, RDMA and DMA samples can be run without **sudo** privileges if a specific option is enabled in the NVIDIA driver:

1. Create a configuration file for the NVIDIA driver:

```
cat <<EOF | sudo tee /etc/modprobe.d/nvidia.conf
options nvidia NVreg_RegistryDwords="PeerMappingOverride=1;"
```

```
EOF
```

2. Perform a cold reboot to ensure the changes take effect.
3. Verify that the configuration has been applied using the following command:

```
$ grep RegistryDwords /proc/driver/nvidia/params
```

You should see the following output confirming the setting:

```
RegistryDwords: "PeerMappingOverride=1;"
```

5.4.11.5.4 DGX Spark

Due to hardware topology limitations, DGX Spark does not support GPUDirect RDMA. However, DOCA GPUNetIO applications can still execute on these systems by utilizing CPU pinned memory (**DOCA_GPU_MEM_TYPE_CPU_GPU**) instead of GPU memory. This applies to both queue allocation and packet/data memory.

5.4.11.5.4.1 Ethernet Configuration

When creating Rx or Tx queues with DOCA Ethernet, you must use the setters in **doca_eth_rxq_gpu_data_path.h** and **doca_eth_txq_gpu_data_path.h** to allocate queues on the CPU-GPU shared memory:

DOCA Ethernet queue memory setters

```
doca_error_t doca_eth_rxq_gpu_set_rq_mem_type(struct doca_eth_rxq *eth_rxq, enum doca_gpu_mem_type rq_mem_type);
doca_error_t doca_eth_txq_gpu_set_sq_mem_type(struct doca_eth_txq *eth_txq, enum doca_gpu_mem_type sq_mem_type);
```

For Tx queues, you must also enable CPU proxy mode to handle transmission:

DOCA Ethernet CPU proxy mode

```
doca_error_t doca_eth_txq_gpu_set_uar_on_cpu(struct doca_eth_txq *eth_txq);
```

See the **gpunetio_simple_send_sample** for an implementation of CPU proxy on the data path.

5.4.11.5.4.2 Verbs RDMA Configuration

A similar approach applies to GPUNetIO Verbs applications, with the distinction that the application is responsible for explicitly allocating the queue memory (QP UMEM) using the **DOCA_GPU_MEM_TYPE_CPU_GPU** memory type.

For the send side, CPU proxy mode is also supported. Refer to the following GPUNetIO Verbs examples for implementation details:

- **gpunetio_verbs_put_bw_sample**
- **gpunetio_verbs_put_counter_bw_sample**
- **gpunetio_verbs_write_bw_sample**

- **gpunetio_verbs_write_lat_sample**

5.4.11.6 GPUNetIO Architecture and Design

5.4.11.6.1 Architecture

A DOCA GPUNetIO network application is split into two fundamental phases:

- Configuration Phase (CPU): The CPU handles all initial setup, such as device configuration, memory allocation, and launching CUDA kernels.
- Data Path Phase (GPU): The GPU and NIC interact directly to execute high-speed packet processing functions.

DOCA GPUNetIO provides the building blocks to create a full data path pipeline that runs entirely on the GPU, often in combination with other libraries like DOCA Ethernet, DOCA RDMA Verbs, DOCA RDMA or DOCA DMA.

5.4.11.6.1.1 Setup and Component Model

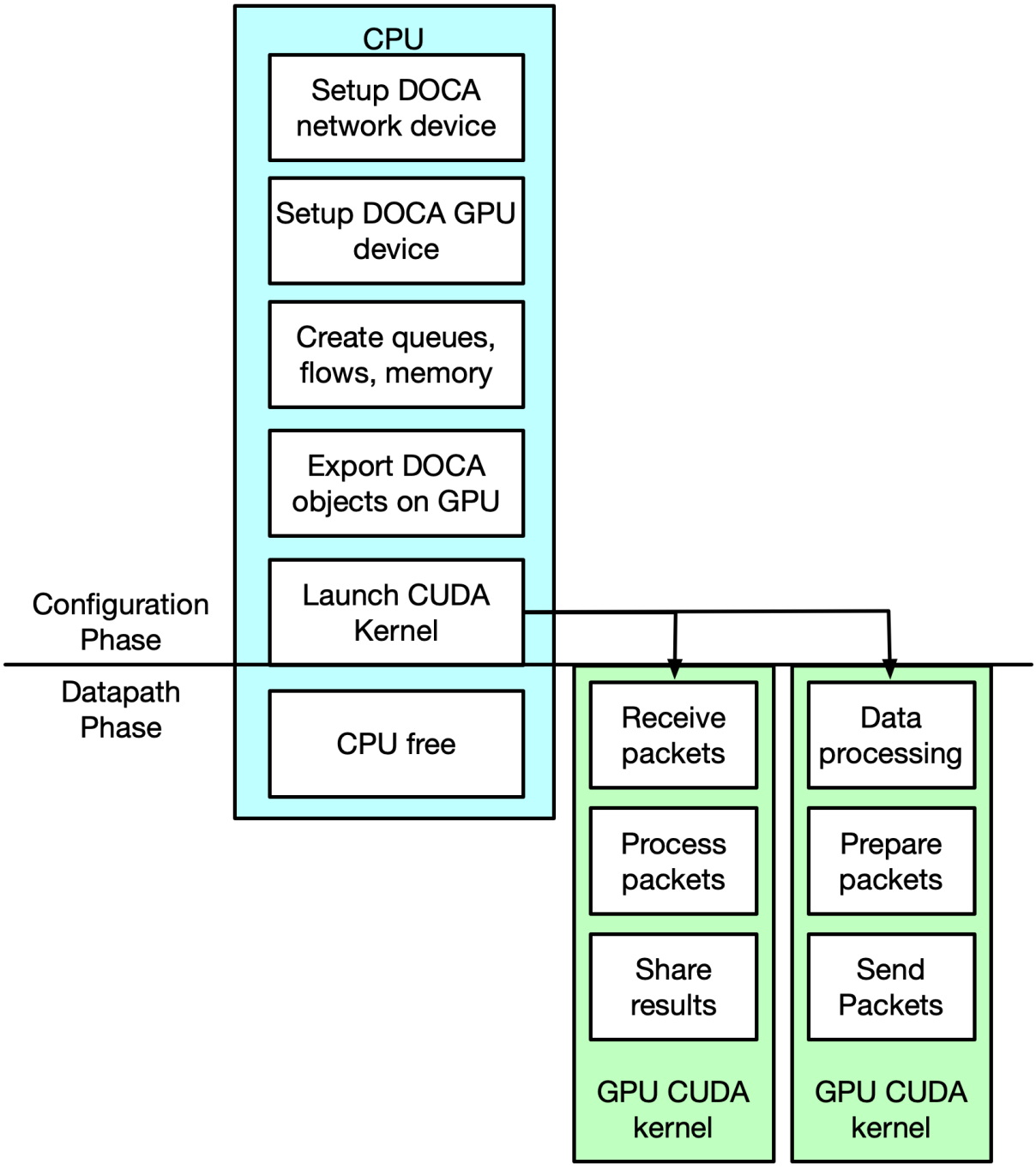
During the setup phase, the CPU-based application must:

1. Prepare all required objects (e.g., queues, contexts) on the CPU.
2. Export a GPU-specific handle for these objects.
3. Launch a CUDA kernel, passing the object's GPU handle to it so the kernel can work with the object during the data path phase.

This "CPU-setup, GPU-run" model is why DOCA GPUNetIO is composed of several distinct components:

- **libdoca_gpunetio.so** (CPU Control Path) A shared library containing control-path functions. The CPU application uses these to prepare the GPU, allocate memory, and configure objects.
- **libdoca_gpunetio_device.a** (GPU Data Path – Static Library) A static library containing data-path functions for GPUNetIO RDMA, GPUNetIO DMA, and GPUNetIO CommCh. These functions are invoked by the GPU from within a CUDA kernel.
- **doca_gpunetio_dev_*.cuh** (GPU Data Path – Headers) A set of header files providing inline data-path functions for GPUNetIO Ethernet and GPUNetIO Verbs. These are compiled directly into the application's CUDA kernels.

The following diagram presents the typical flow:



5.4.11.6.1.2 Library Linking

- The pkgconfig file for the CPU shared library is `doca-gpunetio.pc`.
- There is no pkgconfig file for the GPU static library (`libdoca_gpunetio_device.a`). If your application requires these CUDA device functions, you must explicitly link this library.
  - Default path: `/opt/mellanox/doca/lib/x86_64-linux-gnu/libdoca_gpunetio_device.a`

5.4.11.6.2 GDAKI Data Path for Different Protocols

DOCA GPUNetIO provides GPU GDAKI (GPU Direct Access Kernel Interface) functions to control objects for various transports and protocols that were created using other DOCA libraries. This section explains the correlation between DOCA GPUNetIO and these other libraries.

5.4.11.6.2.1 Ethernet GDAKI Communications

To enable GPU-accelerated communications over the Ethernet transport, an application must use a combination of three DOCA libraries:

- DOCA GPUNetIO: For GPU-specific handles and data path functions.
- DOCA Ethernet: To create and manage the underlying TX/RX queues.
- DOCA Flow: To steer packets to the correct GPU-managed queues.

Control Path Phase: Initial CPU Configuration

Before any data path operations can occur on the GPU, the CPU must first configure all the necessary resources.

1. Create a DOCA Core device handler for the network card.
2. Create a DOCA GPUNetIO device handler for the GPU.
3. Use the DOCA Ethernet library to:
    - Create the required Send Queues (TXQ) and/or Receive Queues (RXQ).
    - Set the data path for these queue handlers to the GPU.
    - Export a GPU-specific handle that represents these queues.
4. Use the DOCA Flow library to create and install flow steering rules that direct the desired types of packets to the newly created DOCA Ethernet receive queues.

Data Path Phase: GPU Kernel Execution

After the configuration phase is complete, the application can launch a CUDA Kernel, passing the GPU handles for the Ethernet queues as input arguments. This allows DOCA GPUNetIO CUDA device functions to operate directly on the queues from within the kernel.

All GPUNetIO Ethernet CUDA device functions are provided as inlined functions in the following header files:

- **doca_gpunetio_dev_eth_rxq.cuh**
- **doca_gpunetio_dev_eth_txq.cuh**

These functions are provided in two distinct APIs:

- Low-level API: Provides fine-grained control over fundamental **mlx5** elements, such as posting Work Queue Entries (WQEs), ringing the network card's doorbell, and polling for Completion Queue Entries (CQEs).
- High-level API: Provides more complex, pre-packaged functions that implement advanced features:
    - Shared Send QP: Allows a single Send Queue to be safely accessed concurrently by different CUDA threads, warps, or blocks.
    - Cooperative Receive QP: Allows a single thread, all threads in a warp, or all threads in a block to cooperate for parallel packet reception from a single Receive Queue.

- Memory Consistency (MCST): A feature for pre-Hopper GPUs to manage memory mappings on the receive side.

Both APIs support CPU proxy mode, a fallback mechanism for systems where direct DoorBell ringing from the GPU is not possible.

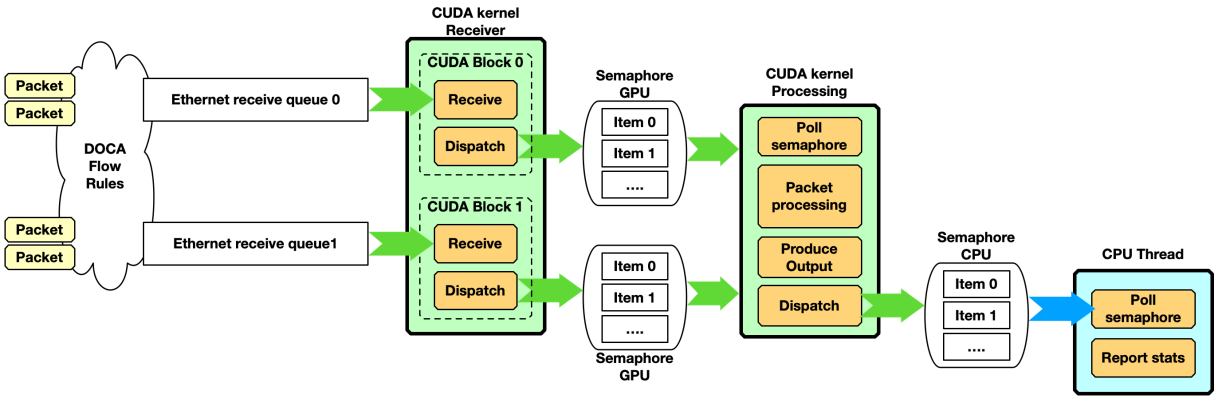
> For examples of how to use both the high-level and low-level GPUNetIO Ethernet APIs, refer to the "GPUNetIO Sample Guide".

Example Use Cases and Further Reading

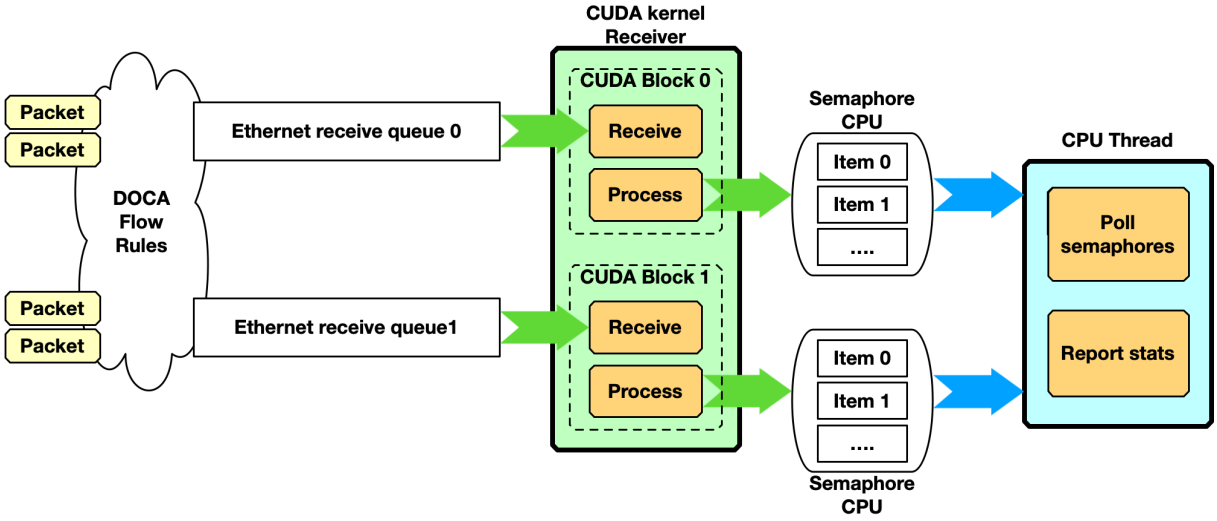
Refer to the DOCA GPU Packet Processing Application Guide (`doca_gpu_packet_processing`) and samples (`doca_gpunetio_simple_send`, `doca_gpunetio_simple_receive`, `doca_gpunetio_send_wait_time`) for examples of Ethernet GPU communications.

> For a deeper understanding of the underlying Ethernet send and receive structures, objects, and functions, refer to the DOCA Ethernet library documentation.

An example diagram when multiple queues and/or semaphores are used to receive Ethernet traffic:



Receiving and dispatching packets to another CUDA kernel is not required. A simpler scenario can have a single CUDA kernel receiving and processing packets:



5.4.11.6.2.2 RDMA Verbs GDAKI Communications (IBGDA)

DOCA GPUNetIO provides GPU data path functions for objects created with the DOCA RDMA and DOCA RDMA Verbs libraries. This enables GPU communications over RDMA transport protocols (IB or RoCE).

DOCA GPUNetIO and DOCA RDMA

This approach uses the high-level DOCA RDMA library, which abstracts most low-level `mlx5` and IBVerbs details. The GPUNetIO CUDA data path functions follow a similarly high-level API.

Key characteristics:

- Provides a high-level API for generic RDMA operations (Write, Send, Read, Recv).
- Delivered as a closed-source CUDA static library (`libdoca_gpunetio_device.a`).
- Does not include built-in shared queue management. Applications must manually manage simultaneous access to queues from different CUDA threads.
- Best suited for simpler GDAKI applications performing basic RDMA operations, as it requires less deep knowledge of IBVerbs or `mlx5` details.

Weak vs. Strong Operation Modes

Some RDMA GPU functions offer two operation modes:

- Weak Mode: The application is responsible for calculating the next available position in the queue.
  - Helper functions (e.g., `doca_gpu_rdma_get_info`) provide the next available position and queue size mask (for index wrapping).
  - The developer must specify the exact queue descriptor number, ensuring no descriptors are skipped.
  - More complex, but offers better performance and allows developers to optimize for GPU memory coalescing.

- Strong Mode: The GPU function automatically enqueues the RDMA operation in the next available position.
    - Simpler to manage, as the developer does not need to track the position.
    - May introduce extra latency due to atomic operations. It also does not guarantee that sequential operations use sequential memory locations.

> ⚠ All strong mode functions operate at the CUDA block level. It is not possible to access the same RDMA queue from two different CUDA blocks simultaneously.

Configuration and Usage

1. Create a device handler for the network card using DOCA Core.
2. Create a GPU device handler for the GPU card using DOCA GPUNetIO.
3. Use DOCA RDMA to:
    - Create send and/or receive queue handlers.
    - Set the queue handlers' data path to the GPU.
    - Export a GPU handler representing those queues.

After configuration, launch a CUDA Kernel, passing the GPU handlers for the RDMA queues as input arguments. Use the functions defined in **doca_gpunetio_dev_rdma.cuh** (starting with **doca_gpu_dev_rdma_***) for RDMA communications in the kernel.

Example Use Cases

Refer to the **doca_gpunetio_rdma_client_server_write** sample for examples of GPUNetIO RDMA functions.

> ✓ For a deeper understanding of RDMA operations, refer to the DOCA RDMA documentation.

DOCA GPUNetIO and DOCA Verbs

This approach uses the lower-level DOCA RDMA Verbs library. The GPUNetIO Verbs CUDA data path functions are provided as inlined functions in the **doca_gpunetio_dev_verbs_*.cuh** header files.

These functions are offered as two different APIs:

- Low-level API: For direct manipulation of fundamental RDMA **mlx5** elements, such as posting Work Queue Entries (WQEs), ringing doorbells, and polling Completion Queues (CQEs). This supports both one-sided (Read, Write, Atomic) and two-sided (Send, Recv) operations.
- High-level API: More complex helper functions that implement common patterns:
    - Shared QP: Allows a single QP to be safely accessed concurrently by different CUDA threads or warps.
    - Combined Operations: Building blocks for concatenating multiple operations (e.g., **put_signal**, which combines an RDMA Write and an Atomic Fetch-and-Add).

- Memory Consistency (MCST): A feature for pre-Hopper GPUs to manage memory mappings on the RDMA Get or Receive side.
- ConnectX-8 reliable doorbell feature: no need to update the DBREC

Both APIs support CPU proxy mode, a fallback mechanism for systems where direct DoorBell ringing from the GPU is not possible. The **samples/doca_gpunetio/verbs_high_level.cpp** file provides helper functions (e.g., **doca_gpu_verbs_create_qp_hl()**) that simplify the CPU-side setup for these Verbs QPs.

The GPUNetIO Verbs APIs are currently experimental. Please report any issues encountered to help improve code quality and robustness.

Configuration and Usage

1. Create a device handler for the network card using DOCA Core.
2. Create a GPU device handler for the GPU card using DOCA GPUNetIO.
3. Use DOCA RDMA Verbs to:
    - Create send and/or receive queue handlers.
    - Set the queue handlers' data path to the GPU.
    - Export a GPU handler representing those queues.

After configuration, launch a CUDA Kernel, passing the GPU handlers for the Verbs queues as input arguments.

Example Use Cases

Refer to samples `doca_gpunetio_verbs_*` for examples of GPUNetIO Verbs functions.

For a deeper understanding of Verbs operations, refer to the DOCA RDMA Verbs documentation.

5.4.11.6.2.3 DMA GDAKI Memory Copies

To enable GPU-triggered memory copies using the DMA engine, an application requires DOCA GPUNetIO and DOCA DMA libraries.

Initial CPU Configuration Phase

1. Create a device handler for the network card using DOCA Core.
2. Create a GPU device handler for the GPU card using DOCA GPUNetIO.
3. Use DOCA DMA to:
    - Create DMA queue handlers.
    - Set queue handlers' data path on the GPU.
    - Export a GPU handler representing those queues.

Data Path Phase on GPU

After completing the configuration phase, launch a CUDA Kernel, passing the GPU handlers for DMA queues as input arguments. This enables DOCA GPUNetIO CUDA device functions to operate within the CUDA Kernel.

For DMA memory copies, use functions defined in `doca_gpunetio_dev_dma.cuh`.

Example Use Case

Refer to the sample `doca_gpunetio_dma_memcpy` for an example of triggering DMA memory copies from a CUDA Kernel.

For a deeper understanding of DMA operations, refer to the DOCA DMA documentation.

5.4.11.7 GPUNetIO API Reference

This section details the specific structures and operations related to the main DOCA GPUNetIO API on CPU and GPU. GPUNetIO headers are:

- **doca_gpunetio.h** – CPU functions to create GPU handlers, allocate GPU memory, etc..
- **doca_gpunetio_dev_eth_rxq.cuh** – Open source GPU functions to manage a DOCA Ethernet receive queue
- **doca_gpunetio_dev_eth_txq.cuh** – Open source GPU functions to manage a DOCA Ethernet send queue
- **doca_gpunetio_dev_verbs_*.cuh** – Open source GPU functions to manage a DOCA RDMA Verbs object
- **doca_gpunetio_dev_buf.cuh** – GPU functions to manage a DOCA buffer array
- **doca_gpunetio_dev_sem.cuh** – GPU functions to manage a DOCA GPUNetIO semaphore
- **doca_gpunetio_dev_rdma.cuh** – GPU functions to manage a DOCA RDMA queue
- **doca_gpunetio_dev_dma.cuh** – GPU functions to manage a DOCA DMA queue

This section lists the main functions of DOCA GPUNetIO. All DOCA Core, Ethernet, Verbs, RDMA and DMA objects used in combination with GPUNetIO have a GPU export function to obtain a GPU handler for that object.

5.4.11.7.1 CPU Functions

In this section there is the list of DOCA GPUNetIO functions that can be used on the CPU only.

5.4.11.7.1.1 doca_gpu_mem_type

This enum lists all the possible memory types that can be allocated with GPUNetIO.

```
enum doca_gpu_mem_type {
    DOCA_GPU_MEM_TYPE_GPU        = 0,
    DOCA_GPU_MEM_TYPE_GPU_CPU    = 1,
    DOCA_GPU_MEM_TYPE_CPU_GPU    = 2,
};
```

With regards to the syntax, the text string after the **DOCA_GPU_MEM_TYPE_** prefix signifies **<where-memory-resides>_<who-has-access>**.

- **DOCA_GPU_MEM_TYPE_GPU** – memory resides on the GPU and is accessible from the GPU only

- **DOCA_GPU_MEM_TYPE_GPU_CPU** – memory resides on the GPU and is accessible also by the CPU
- **DOCA_GPU_MEM_TYPE_CPU_GPU** – memory resides on the CPU and is accessible also by the GPU

Typical usage of the **DOCA_GPU_MEM_TYPE_GPU_CPU** memory type is to send a notification from the CPU to the GPU (e.g., a CUDA kernel periodically checking to see if the exit condition set by the CPU is met).

5.4.11.7.1.2 doca_gpu_create

This is the first function a GPUNetIO application must invoke to create an handler on a GPU device. The function initializes a pointer to a structure in memory with type **struct doca_gpu *** .

```
doca_error_t doca_gpu_create(const char *gpu_bus_id, struct doca_gpu **gpu_dev);
```

- **gpu_bus_id** – **<PCIe-bus>:<device>.<function>** of the GPU device you want to use in your application
- **gpu_dev [out]** – GPUNetIO handler to that GPU device

To get the PCIe address, users can use the commands **lspci** or **nvidia-smi** .

5.4.11.7.1.3 doca_gpu_mem_alloc

This CPU function allocates different flavors of memory.

```
doca_error_t doca_gpu_mem_alloc(struct doca_gpu *gpu_dev, size_t size, size_t alignment, enum doca_gpu_mem_type
mtype, void **memptr_gpu, void **memptr_cpu)
```

- **gpu_dev** – GPUNetIO device handler
- **size** – Size, in bytes, of the memory area to allocate
- **alignment** – Memory address alignment to use. If 0, default one will be used
- **mtype** – Type of memory to allocate
- **memptr_gpu [out]** – GPU pointer to use to modify that memory from the GPU if memory is allocated on or is visible by the GPU
- **memptr_cpu [out]** – CPU pointer to use to modify that memory from the CPU if memory is allocated on or is visible by the CPU. Can be NULL if memory is GPU-only

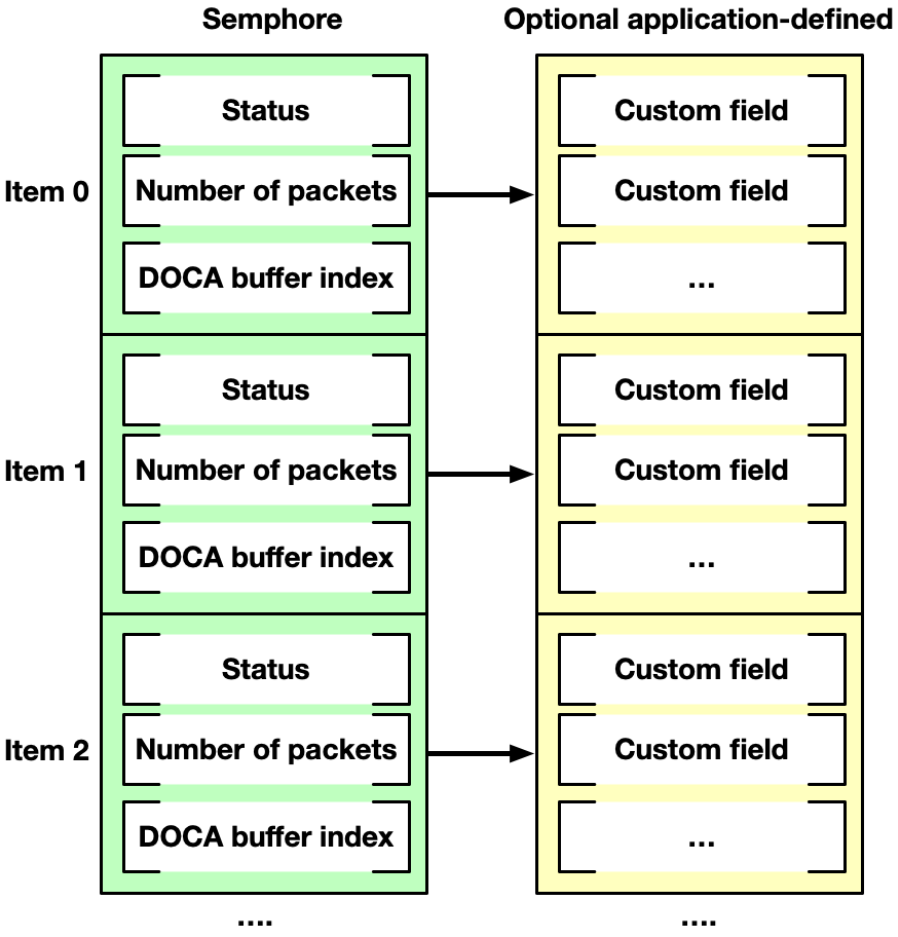
Make sure to use the right pointer on the right device! If an application tries to access the memory using the **memptr_gpu** address from the CPU, a segmentation fault will result.

5.4.11.7.1.4 doca_gpu_semaphore_create

Creates a new instance of a DOCA GPUNetIO semaphore. A semaphore is composed by a list of items each having, by default, a status flag, number of packets, and the index of a **doca_gpu_buf** in a **doca_gpu_buf_arr** .

For example, a GPUNetIO semaphore can be used in applications where a CUDA kernel is responsible for receiving packets in a **doca_gpu_buf_arr** array associated with an Ethernet receive queue object, **doca_gpu_eth_rxq** (see section "doca_gpu_dev_eth_rxq_receive_*"), and dispatching packet info to a second CUDA kernel which processes them.

Another way to use a GPUNetIO semaphore is to exchange data across different entities like two CUDA kernels or a CUDA kernel and a CPU thread. The reason for this scenario may be that the CUDA kernel needs to provide the outcome of the packet processing to the CPU which would in turn compile a statistics report. Therefore, it is possible to associate a custom application-defined structure with each item in the semaphore. This way, the semaphore can be used as a message passing object.



Entities communicating through a semaphore must adopt a poll/update mechanism according to the following logic:

- Update:
  a. Populate the next item of the semaphore (packets' info and/or custom application-defined info).
  b. Set status flag to READY.
- Poll:
  a. Wait for the next item to have a status flag equal to **READY**.

b. Read and process info.
c. Set status flag to **DONE**.

```
doca_error_t doca_gpu_semaphore_create(struct doca_gpu *gpu_dev, struct doca_gpu_semaphore **semaphore)
```

- **gpu_dev** – GPUNetIO handler
- **semaphore [out]** – GPUNetIO semaphore handler associated to the GPU device

5.4.11.7.1.5 doca_gpu_semaphore_set_memory_type

This function defines the type of memory for the semaphore allocation.

```
doca_error_t doca_gpu_semaphore_set_memory_type(struct doca_gpu_semaphore *semaphore, enum doca_gpu_mem_type mtype)
```

- **semaphore** – GPUNetIO semaphore handler
- **mtype** – Type of memory to allocate the custom info structure
  - If the application must share packet info only across CUDA kernels, then **DOCA_GPU_MEM_GPU** is the suggested memory type.
  - If the application must share info from a CUDA kernel to a CPU (e.g., to report statistics or output of the pipeline computation), then **DOCA_GPU_MEM_CPU_GPU** is the suggested memory type

5.4.11.7.1.6 doca_gpu_semaphore_set_items_num

This function defines the number of items in a semaphore.

```
doca_error_t doca_gpu_semaphore_set_items_num(struct doca_gpu_semaphore *semaphore, uint32_t num_items)
```

- **semaphore** – GPUNetIO semaphore handler
- **num_items** – Number of items to allocate

5.4.11.7.1.7 doca_gpu_semaphore_set_custom_info

This function associates an application-specific structure to semaphore items as explained under "doca_gpu_semaphore_create".

```
doca_error_t doca_gpu_semaphore_set_custom_info(struct doca_gpu_semaphore *semaphore, uint32_t nbytes, enum
 doca_gpu_mem_type mtype)
```

- **semaphore** – GPUNetIO semaphore handler
- **nbytes** – Size of the custom info structure to associate
- **mtype** – Type of memory to allocate the custom info structure
  - If the application must share packet info only across CUDA kernels, then **DOCA_GPU_MEM_GPU** is the suggested memory type
  - If the application must share info from a CUDA kernel to a CPU (e.g., to report statistics or output of the pipeline computation), then **DOCA_GPU_MEM_CPU_GPU** is the suggested memory type

5.4.11.7.1.8 doca_gpu_semaphore_get_status

From the CPU, query the status of a semaphore item. If the semaphore is allocated with **DOCA_GPU_MEM_GPU**, this function results in a segmentation fault.

```
doca_error_t doca_gpu_semaphore_get_status(struct doca_gpu_semaphore *semaphore_cpu, uint32_t idx, enum
 doca_gpu_semaphore_status *status)
```

- **semaphore_cpu** – GPUNetIO semaphore CPU handler
- **idx** – Semaphore item index
- **status [out]** – Output semaphore status

5.4.11.7.1.9 doca_gpu_semaphore_get_custom_info_addr

From the CPU, retrieve the address of the custom info structure associated to a semaphore item. If the semaphore or the custom info is allocated with **DOCA_GPU_MEM_GPU** this function results in a segmentation fault.

```
doca_error_t doca_gpu_semaphore_get_custom_info_addr(struct doca_gpu_semaphore *semaphore_cpu, uint32_t idx, void
 **custom_info)
```

- **semaphore_cpu** – GPUNetIO semaphore CPU handler
- **idx** – Semaphore item index
- **custom_info [out]** – Output semaphore custom info address

5.4.11.7.1.10 doca_gpu_verbs_export_qp

The `doca_gpu_verbs_export_qp` function creates a GPUNetIO handler from a DOCA RDMA Verbs QP object. It takes a DOCA RDMA Verbs QP as input and returns a DOCA GPUNetIO Verbs QP object (`struct doca_gpu_verbs_qp`) allocated on the CPU. To use this object in a CUDA kernel, the application must extract a GPU device handler (`struct doca_gpu_dev_verbs_qp`) using the `doca_gpu_verbs_get_qp_dev` function.

```
doca_error_t doca_gpu_verbs_export_qp(struct doca_gpu *gpu_dev, struct doca_dev *dev, struct doca_verbs_qp *qp,
enum doca_gpu_dev_verbs_nic_handler nic_handler, void *gpu_qp_umem_dev_ptr, struct doca_verbs_cq *cq_sq, struct
 doca_verbs_cq *cq_rq, struct doca_gpu_verbs_qp **qp_out);
```

- `gpu_dev`: GPUNetIO device handler.
- `dev`: DOCA device handler.
- `qp`: DOCA RDMA Verbs QP handler.
- `nic_handler`: Type of NIC handler.
- `gpu_qp_umem_dev_ptr`: GPU memory pointer to UMEM.
- `cq_sq` and `cq_rq`: CQs associated with the Send and Receive Queues in the QP.

  > ⚠ While either `cq_sq` or `cq_rq` can be NULL, they cannot both be NULL simultaneously.

- `qp_out`: DOCA GPUNetIO Verbs QP handler in CPU memory.

5.4.11.7.1.11 doca_gpu_verbs_get_qp_dev

Extracts a GPU device handler (**struct doca_gpu_dev_verbs_qp**) from a DOCA GPUNetIO Verbs QP object.

```
doca_error_t doca_gpu_verbs_get_qp_dev(struct doca_gpu_verbs_qp *qp, struct doca_gpu_dev_verbs_qp **qp_gpu);
```

- qp: DOCA GPUNetIO Verbs QP handler.
- qp_gpu: DOCA GPUNetIO Verbs QP GPU device handler in GPU memory.

5.4.11.7.1.12 doca_gpu_verbs_unexport_qp

Unexports a previously exported DOCA GPUNetIO Verbs QP object (**struct doca_gpu_verbs_qp**).

```
doca_error_t doca_gpu_verbs_export_qp(ruct doca_gpu *gpu_dev, struct doca_gpu_verbs_qp *qp);
```

- gpu_dev: GPUNetIO device handler.
- qp: DOCA GPUNetIO Verbs QP handler in CPU memory.

5.4.11.7.1.13 doca_gpu_verbs_bridge_export_qp

The doca_gpu_verbs_bridge_export_qp function creates a DOCA GPUNetIO Verbs QP object from application-defined parameters, acting as a bridge between IBVerbs/mlx5 objects and DOCA GPUNetIO. This allows applications to create required objects like QP, CQ, and UAR using IBVerbs and mlx5 commands and then pass the relevant information to this DOCA GPUNetIO function.

The function returns a DOCA GPUNetIO Verbs QP object (struct doca_gpu_verbs_qp) allocated on the CPU. To use this object in a CUDA kernel, the application must extract a GPU device handler (struct doca_gpu_dev_verbs_qp) using the doca_gpu_verbs_get_qp_dev function.

It is the application's responsibility to ensure that all passed parameters are correctly created and set.

```
doca_error_t doca_gpu_verbs_bridge_export_qp(struct doca_gpu *gpu_dev, uint32_t sq_qpn, void *sq_wqe_addr, uint16_t
sq_wqe_num, uint32_t *sq_dbrec, uint64_t *sq_db, size_t uar_size, uint32_t sq_cqn, void *sq_cqe_addr, uint32_t
sq_cqe_num, uint32_t *sq_cq_dbrec, uint32_t rq_qpn, void *rq_wqe_addr, uint16_t rq_wqe_num, uint32_t *rq_dbrec,
uint32_t rcv_wqe_size, uint32_t rq_cqn, void *rq_cqe_addr, uint32_t rq_cqe_num, uint32_t *rq_cq_dbrec, enum
 doca_gpu_dev_verbs_nic_handler nic_handler, struct doca_gpu_verbs_qp **qp_out);
```

- gpu_dev: GPUNetIO device handler.
- sq_qpn: Send QP queue number.
- sq_wqe_addr: Send QP WQE buffer memory address.
- sq_wqe_num: Send QP number of WQEs.
- sq_dbrec: Send QP Doorbell Record address.
- sq_db: Send QP Doorbell address.
- uar_size: Size of UAR.
- sq_cqn: Send CQ number.
- sq_cqe_addr: Send CQ CQE buffer memory address.
- sq_cqe_num: Send CQ number of CQEs.
- qp_out: DOCA GPUNetIO Verbs QP handler in CPU memory.

5.4.11.7.2 DOCA PE

A DOCA Ethernet Txq context, exported for GPUNetIO usage, can be tracked via DOCA PE on the CPU side to check if there are errors when sending packets or to retrieve notification info after sending a packet with any of the `doca_gpu_dev_eth_txq_*_enqueue_*` functions on the GPU. An example can be found in the DOCA GPU packet processing application with ICMP traffic.

A DOCA Comch Producer or Consumer context, exported for GPUNetIO usage must still be attached to a DOCA PE on the CPU to ensure that remote producer and consumer connections and disconnections are handled properly.

5.4.11.7.3 GPU Functions – Ethernet

This section provides a list of DOCA GPUNetIO functions that can be used for Ethernet GDAKI network operations within GPU CUDA kernels.

5.4.11.7.3.1 Header Files

- `doca_gpunetio_eth_def.h` : Contains constants, enums, and structure definitions.
- `doca_gpunetio_dev_eth_common.cuh` : Provides CUDA utility functions common to both Txq and Rxq header files.
- `doca_gpunetio_dev_eth_txq.cuh` : Provides CUDA functions for Ethernet send operations. This includes both high-level (with shared QP feature) and low-level (post WQE, ring DB, poll CQ, etc.) functions.
- `doca_gpunetio_dev_eth_rxq.cuh` : Provides CUDA functions for Ethernet receive operations.

Experimental API

The new GPUNetIO Ethernet APIs are experimental. Please report any issues encountered during use, as your feedback is crucial for enhancing the code's quality and robustness.

Private Functions

When using the GPUNetIO Ethernet header files, avoid calling functions that start with `doca_priv_*`. These are internal functions intended for use by the public API only.

Debug Macro

The `doca_gpunetio_eth_def.h` file contains the macro `#define DOCA_GPUNETIO_ETH_ENABLE_DEBUG 0`. This default setting disables debug prints, including CQE errors. To enable error messages, set this macro to `1`.

5.4.11.7.3.2 Execution Scope and Shared Queue

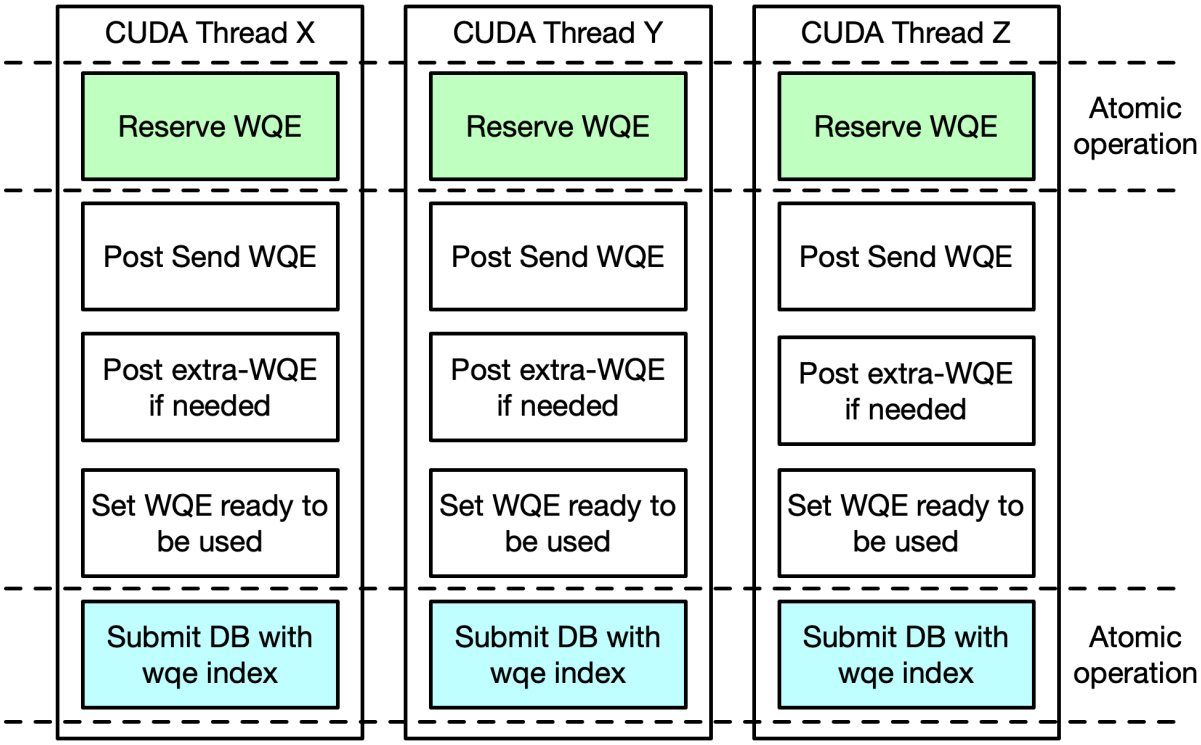
The execution scope for high-level GPUNetIO Ethernet APIs (thread, warp, or block) is specified via the `doca_gpu_dev_eth_exec_scope` enum as a template argument. This tells the function how many threads are participating in the operation.

- Send (Txq): The high-level send API supports the shared queue feature. This allows different threads, belonging to different scopes, to concurrently access the same Txq without race conditions.
- Receive (Rxq): The high-level receive API does not support the shared queue feature. If a thread, warp, or block is using an Rxq, other threads, warps, or blocks cannot use the same Rxq in parallel.

The following subsections detail the send execution scopes.

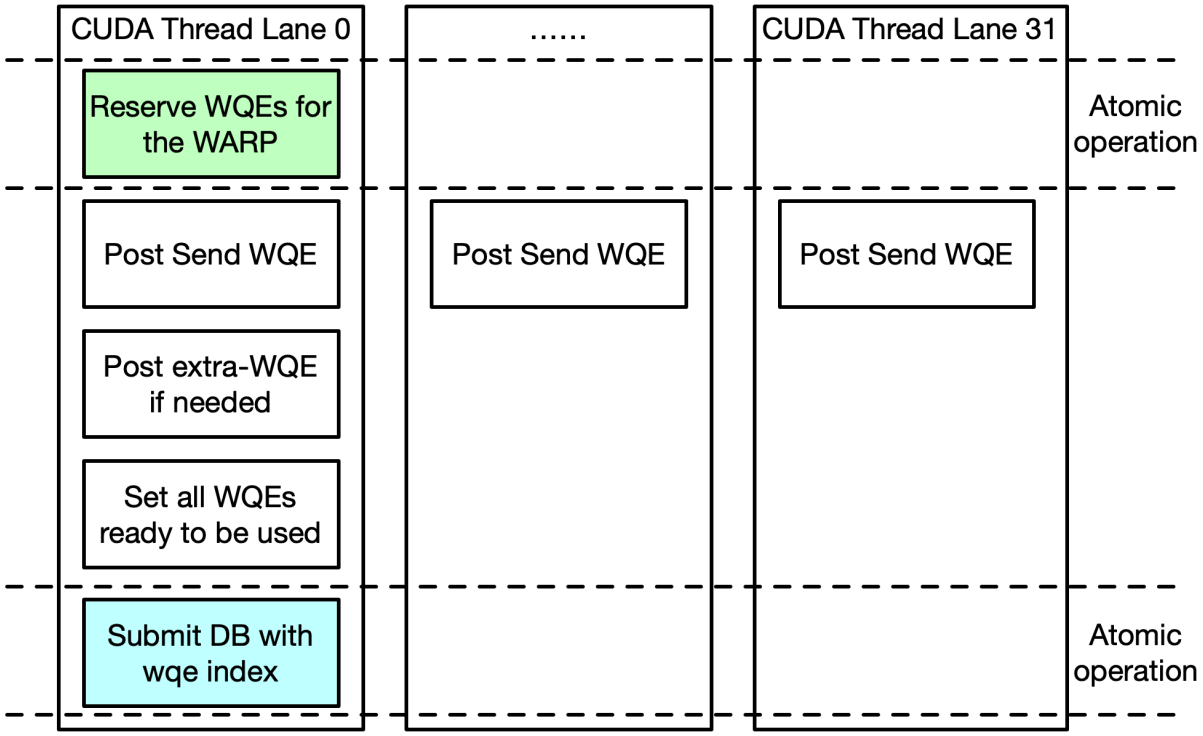
Thread Scope (`DOCA_GPUNETIO_ETH_EXEC_SCOPE_THREAD`)

- Each thread acts as a standalone entity.
- Each thread posts one or more WQEs and is responsible for its own submit (doorbell ring).
- Performance: This is the slowest approach due to high contention on atomic operations.



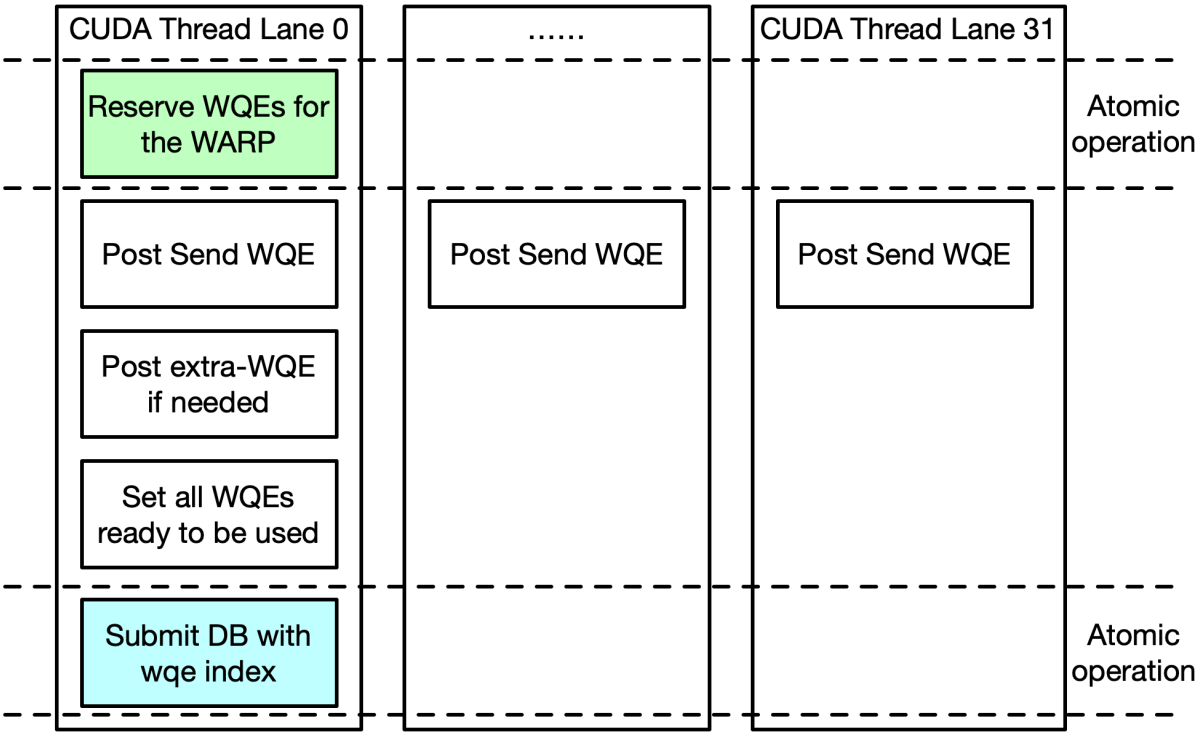
Warp Scope (`DOCA_GPUNETIO_ETH_EXEC_SCOPE_WARP`)

- All threads within the warp must call the function.
- Each thread posts a WQE to a different location, but only the first thread (**`lane_idx 0`**) performs the submit operation.
- Performance: Reduces contention to a per-warp basis.



Block Scope (`DOCA_GPUNETIO_ETH_EXEC_SCOPE_BLOCK`)

- All threads within the block must call the function.
- Each thread posts a WQE to a different location, but only the first thread (**`threadIdx 0`**) performs the submit operation.
- Performance: Reduces contention to a per-block basis.



5.4.11.7.3.3 Ring DB and CPU Proxy

The `doca_gpu_dev_eth_nic_handler` enum, passed as a template argument, determines who is responsible for ringing the network card's doorbell.

The available modes are:

- `DOCA_GPUNETIO_ETH_NIC_HANDLER_AUTO`: Automatically detects the best doorbell ringing option.
- `DOCA_GPUNETIO_ETH_NIC_HANDLER_CPU_PROXY`: Enables CPU proxy mode. GPU submit functions provide information to the CPU, which then rings the doorbell.
- `DOCA_GPUNETIO_ETH_NIC_HANDLER_GPU_SM_DB`: Enables regular GDAKI mode, where CUDA threads ring the doorbell directly.

> **CPU Proxy Mode**
>
> When `CPU_PROXY` mode is enabled, a CPU thread must invoke `doca_eth_txq_gpu_cpu_proxy_progress` in a loop to detect the GPU's information and ring the doorbell. The `GPUNetIO Ethernet Simple Send` sample demonstrates this feature.

5.4.11.7.3.4 Memory Consistency (MCST) Algorithm

During a receive operation, the network card writes packet data via PCIe to the application's mapped memory. For pre-Hopper GPUs (see CUGPUDirectRDMAWritesOrdering), memory consistency (MCST) must be ensured when a CUDA kernel is receiving data into GPU memory.

DOCA GPUNetIO provides two ways to enable the MCST algorithm:

1. CPU configuration phase: When creating the DOCA Ethernet Rxq, use the function **doca_eth_rxq_gpu_enable_mcst_qp** to create an internal queue dedicated to the MCST algorithm.

```
cudaGetDeviceProperties(&prop, cuda_id);
// If pre-Hopper GPU with __CUDA_ARCH__ < 900
if (prop.major < 9)
    doca_eth_rxq_gpu_enable_mcst_qp(rxq->eth_rxq_cpu);
```

2. GPU data path phase: Pass the **doca_gpu_dev_eth_mcst_mode** enum as a template argument to the **recv** function.

5.4.11.7.3.5 Receive, High-level Function: doca_gpu_dev_eth_rxq_recv

This function receives Ethernet packets in a CUDA kernel and is available in different flavors for per-thread, per-warp, or per-block scopes.

```
template <enum doca_gpu_dev_eth_exec_scope exec_scope = DOCA_GPUNETIO_ETH_EXEC_SCOPE_THREAD,
      enum doca_gpu_dev_eth_mcst_mode mcst_mode = DOCA_GPUNETIO_ETH_MCST_DISABLED,
      enum doca_gpu_dev_eth_nic_handler nic_handler = DOCA_GPUNETIO_ETH_NIC_HANDLER_AUTO,
      bool enable_attributes = false>
__device__ inline doca_error_t doca_gpu_dev_eth_rxq_recv(struct doca_gpu_eth_rxq *rxq,
                           uint32_t max_rx_pkts,
                           uint64_t timeout_ns,
                           volatile uint64_t *out_first_pkt_idx,
                           uint32_t *out_pkt_num,
                           struct doca_gpu_dev_eth_rxq_attr *out_attr);
```

Parameters:

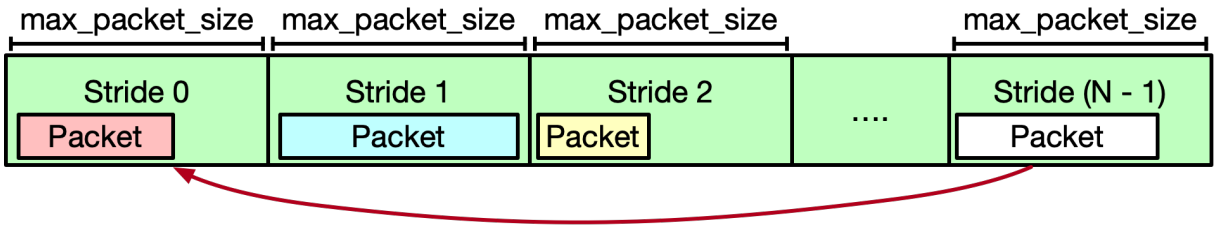
- **rxq** : The Ethernet receive queue GPU handler.
- **max_rx_pkts** : Maximum number of packets to receive. The function guarantees the returned packet count is <= this number.
- **timeout_ns** : Nanoseconds to wait for packets before returning.
- **out_first_pkt_idx** [out]: Index of the first packet received. For block or warp scopes, this variable must be in shared or global memory to be visible to all threads.
- **out_pkt_num** [out]: The actual number of packets received. For block or warp scopes, this variable must be in shared or global memory.
- **out_attr** [out]: If **enable_attributes** is true, this structure is filled with per-packet attributes. The application must ensure this array is large enough to hold attributes for **max_rx_pkts** .

Notes

- If both **max_rx_pkts** and **timeout_ns** are set to **0** , the function will hang.
- For **BLOCK** or **WARP** scopes, the **max_rx_pkts** value should be at least the number of threads in the scope (e.g., **blockDim.x** or **warpSize**), as every thread will try to receive at least once.
- This function does not support the shared QP feature. Access to the **rxq** must be exclusive to the calling thread, warp, or block. No other scope can work on the same **rxq** in parallel.

Packet Indexing and Circular Buffer

When the Rxq is created on the CPU, its GPU memory is divided into fixed-size strides (based on **max_packet_size**). The **doca_gpu_dev_eth_rxq_recv** function guarantees that these strides are filled sequentially with new packets.



The function returns the following output parameters to identify the received packets:

- **out_first_packet_idx**: The index of the first stride (packet) filled.
- **out_pkt_num**: The total number of packets received.

For example, if **out_first_packet_idx** is **X** and **out_pkt_num** is **Y**, the function has received **Y** packets, filling the GPU memory buffer strides from index **X** to **(X + Y - 1)**.

> ⚠ The output parameters **out_first_packet_idx** and **out_pkt_num** must be visible to all threads in the scope (e.g., via CUDA shared memory for warp and block scopes).

The receive buffer is treated as a circular buffer. Once the last stride is filled, the queue wraps around to the first stride.

> ⚠ It is the application's responsibility to consume packets before they are overwritten by the queue wrapping around. This requires proper dimensioning of the Rxq GPU buffer and scaling across multiple receive queues.

5.4.11.7.3.6 Receive: doca_gpu_dev_eth_rxq_get_pkt_addr

This utility function retrieves the memory address of a specific packet in a specific stride.

```
__device__ inline uint64_t doca_gpu_dev_eth_rxq_get_pkt_addr(struct doca_gpu_eth_rxq *rxq, uint64_t packet_idx);
```

Parameter:

- **rxq**: Ethernet receive queue GPU handler.
- **packet_idx**: Index of the stride where the packet resides.

5.4.11.7.3.7 Send, High-level Function: doca_gpu_dev_eth_txq_send

This function sends Ethernet packets from a CUDA kernel, with flavors for per-thread, per-warp, or per-block scopes. It enables the shared queue feature, allowing concurrent access to the same Txq.

```
template <enum doca_gpu_dev_eth_resource_sharing_mode resource_sharing_mode =
DOCA_GPUNETIO_ETH_RESOURCE_SHARING_MODE_GPU,
      enum doca_gpu_dev_eth_sync_scope sync_scope = DOCA_GPUNETIO_ETH_SYNC_SCOPE_GPU,
      enum doca_gpu_dev_eth_nic_handler nic_handler = DOCA_GPUNETIO_ETH_NIC_HANDLER_AUTO,
      enum doca_gpu_dev_eth_exec_scope exec_scope = DOCA_GPUNETIO_ETH_EXEC_SCOPE_THREAD>
__device__ static inline void doca_gpu_dev_eth_txq_send(struct doca_gpu_eth_txq *txq,
                            uint64_t addr,
                            uint32_t mkey,
                            size_t size,
                            enum doca_gpu_eth_send_flags flags,
                            doca_gpu_dev_eth_ticket_t *out_ticket)
```

Parameters:

- **txq** : Ethernet send queue GPU handler.
- **addr** : Memory address of the packet to send.
- **mkey** : Memory key of the packet.
- **size** : Size of the packet in bytes.
- **flags** : Send flags from the **doca_gpu_eth_send_flags** enum.
- **out_ticket** : Returns the WQE index (position) in the send queue where the packet was posted.

5.4.11.7.3.8 Send, High-level Function: doca_gpu_dev_eth_txq_wait_send

This function combines the "Accurate Send Scheduling" feature (wait on timestamp) with the send operation. The first thread in the scope posts a "wait on time" barrier, and then all threads in the scope post their send. This function also enables the shared queue feature.

```
template <enum doca_gpu_dev_eth_resource_sharing_mode resource_sharing_mode =
DOCA_GPUNETIO_ETH_RESOURCE_SHARING_MODE_GPU,
      enum doca_gpu_dev_eth_sync_scope sync_scope = DOCA_GPUNETIO_ETH_SYNC_SCOPE_GPU,
      enum doca_gpu_dev_eth_nic_handler nic_handler = DOCA_GPUNETIO_ETH_NIC_HANDLER_AUTO,
      enum doca_gpu_dev_eth_exec_scope exec_scope = DOCA_GPUNETIO_ETH_EXEC_SCOPE_THREAD>
__device__ static inline void doca_gpu_dev_eth_txq_wait_send(struct doca_gpu_eth_txq *txq,
                              const uint64_t wait_on_time_ts,
                              uint64_t addr,
                              uint32_t mkey,
                              size_t size,
                              enum doca_gpu_eth_send_flags flags,
                              doca_gpu_dev_eth_ticket_t *out_ticket);
```

Parameters:

- **txq** : Ethernet send queue GPU handler.
- **wqe_ptr** : Memory pointer to the WQE (can be obtained via **doca_gpu_dev_eth_txq_get_wqe_ptr**).
- **wqe_idx** : Index of the WQE.
- **addr** : Packet memory address.
- **mkey** : Packet memory key.
- **nbytes** : Packet size in bytes.
- **flags** : Send flags from the **doca_gpu_eth_send_flags** enum.

5.4.11.7.3.9 Send, Low-level Function: doca_gpu_dev_eth_txq_wqe_prepare_send

This low-level function posts a send WQE. It is used by the high-level functions but can also be called directly by the application.

```
__device__ __inline__ static doca_error_t doca_gpu_dev_eth_txq_wqe_prepare_send(
    const struct doca_gpu_eth_txq *txq,
    struct doca_gpu_dev_eth_txq_wqe *wqe_ptr,
```

```
    const uint16_t wqe_idx,
    const uint64_t addr,
    const uint32_t mkey,
    const uint32_t nbytes,
    const enum doca_gpu_eth_send_flags flags);
```

Parameters:

- **txq** : Ethernet send queue GPU handler.
- **wqe_ptr** : Memory pointer to the WQE (can be obtained via **doca_gpu_dev_eth_txq_get_wqe_ptr**).
- **wqe_idx** : Index of the WQE.
- **addr** : Packet memory address.
- **mkey** : Packet memory key.
- **nbytes** : Packet size in bytes.
- **flags** : Send flags from the **doca_gpu_eth_send_flags** enum.

5.4.11.7.3.10 Send, Low-level Function: doca_gpu_dev_eth_txq_wqe_prepare_wait_time

This low-level function posts a "wait on time" WQE to create a time barrier. It is used by **doca_gpu_dev_eth_txq_wait_send** but can also be called directly.

```
__device__ __inline__ static doca_error_t doca_gpu_dev_eth_txq_wqe_prepare_wait_time(
    const struct doca_gpu_eth_txq *txq,
    struct doca_gpu_dev_eth_txq_wqe *wqe_ptr,
    const uint16_t wqe_idx,
    const uint64_t wait_on_time_ts,
    const enum doca_gpu_eth_send_flags flags);
```

Parameters:

- **txq** : Ethernet send queue GPU handler.
- **wqe_ptr** : Memory pointer to the WQE (can be obtained via **doca_gpu_dev_eth_txq_get_wqe_ptr**).
- **wqe_idx** : Index of the WQE.
- **wait_on_time_ts** : Timestamp barrier. The timestamp can be calculated on the GPU (via **doca_gpu_dev_eth_txq_calculate_timestamp**) or CPU (via **doca_eth_txq_calculate_timestamp**).
- **flags** : Send flags from the **doca_gpu_eth_send_flags** enum.

5.4.11.7.3.11 Send, Low-level Function: doca_gpu_dev_eth_txq_submit

This function "rings the doorbell" to notify the network card to execute all WQEs posted since the last submit. It supports the shared queue feature; internal atomics allow multiple threads to call it in parallel, and the doorbell register will be updated only by the thread with the greatest WQE index. It also allows choosing between GPU and CPU Proxy handlers via the template parameter.

```
template <enum doca_gpu_dev_eth_resource_sharing_mode resource_sharing_mode =
DOCA_GPUNETIO_ETH_RESOURCE_SHARING_MODE_GPU,
      enum doca_gpu_dev_eth_sync_scope sync_scope = DOCA_GPUNETIO_ETH_SYNC_SCOPE_GPU,
      enum doca_gpu_dev_eth_nic_handler nic_handler = DOCA_GPUNETIO_ETH_NIC_HANDLER_AUTO>
__device__ static inline void doca_gpu_dev_eth_txq_submit(struct doca_gpu_eth_txq *txq, uint64_t prod_index);
```

Parameters:

- **txq** : Ethernet send queue GPU handler.

- **prod_index** : The Producer Index (WQE index) to write into the network card's doorbell register.

5.4.11.7.3.12 Send, Low-level Functions: ...update_dbr , ...ring_db , and ...submit_proxy

These functions can replace **doca_gpu_dev_eth_txq_submit** in special situations, such as when the application has a well-determined send pattern and does not need atomics for shared queue access. The typical sequence is **doca_gpu_dev_eth_txq_update_dbr** (updates DBREC) followed by **doca_gpu_dev_eth_txq_ring_db** (updates Doorbell Register).

```
template <enum doca_gpu_dev_eth_sync_scope sync_scope = DOCA_GPUNETIO_ETH_SYNC_SCOPE_GPU>
__device__ static inline void doca_gpu_dev_eth_txq_update_dbr(struct doca_gpu_eth_txq *txq, uint32_t prod_index);
```

Parameters:

- **txq** : Ethernet send queue GPU handler.
- **prod_index** : The WQE index to write into the network card DBREC.

```
template <enum doca_gpu_dev_eth_sync_scope sync_scope = DOCA_GPUNETIO_ETH_SYNC_SCOPE_GPU>
__device__ static inline void doca_gpu_dev_eth_txq_ring_db(struct doca_gpu_eth_txq *txq, uint64_t prod_index);
```

Parameters:

- **txq** : Ethernet send queue GPU handler.
- **prod_index** : The WQE index to write into the network card doorbell register.

doca_gpu_dev_eth_txq_ring_db works for direct GPU doorbell ringing. To ring the doorbell via CPU proxy, **doca_gpu_dev_eth_txq_submit_proxy** must be used instead:

```
template <enum doca_gpu_dev_eth_resource_sharing_mode resource_sharing_mode =
DOCA_GPUNETIO_ETH_RESOURCE_SHARING_MODE_GPU,
      enum doca_gpu_dev_eth_sync_scope sync_scope = DOCA_GPUNETIO_ETH_SYNC_SCOPE_GPU>
__device__ static inline void doca_gpu_dev_eth_txq_submit_proxy(struct doca_gpu_eth_txq *txq, uint64_t prod_index);
```

Parameter:

- **txq** : Ethernet send queue GPU handler.
- **prod_index** : The WQE index to write into the CPU proxy shared memory.

5.4.11.7.3.13 Send, Low-level Functions: Poll Completion

These functions allow waiting for send operations to complete. A Completion (CQE) is generated only if the **DOCA_GPUNETIO_ETH_SEND_FLAG_NOTIFY** flag was used when posting the WQE.

doca_gpu_dev_eth_txq_poll_completion waits for a number of CQEs. It operates at thread scope but supports the shared queue feature (each thread locks **num_cqe** slots to poll).

```
template <enum doca_gpu_dev_eth_cq_poll_mode cqe_poll = DOCA_GPUNETIO_ETH_CQ_POLL_ALL,
      enum doca_gpu_dev_eth_resource_sharing_mode resource_sharing_mode =
DOCA_GPUNETIO_ETH_RESOURCE_SHARING_MODE_GPU,
      enum doca_gpu_dev_eth_sync_scope sync_scope = DOCA_GPUNETIO_ETH_SYNC_SCOPE_GPU>
__device__ static inline doca_error_t doca_gpu_dev_eth_txq_poll_completion(struct doca_gpu_eth_txq *txq,
                                    const uint32_t num_cqe,
                                    enum doca_gpu_dev_eth_wait_flags wait_mode,
```

```
                                                uint32_t *num_completed);
```

Parameters:

- **txq** : Ethernet send queue GPU handler.
- **num_cqe** : Number of CQEs to wait for.
- **wait_mode** : Wait in blocking or non-blocking mode (flags from **doca_gpu_dev_eth_wait_flags**).
- **num_completed** : Number of CQEs detected.

> ⚠ When **num_cqe** > 1, setting the **cqe_poll** template to **DOCA_GPUNETIO_ETH_CQ_POLL_ALL** can reduce latency, as the function will only poll the *latest* CQE (**num_cqe - 1**) instead of polling sequentially.

doca_gpu_dev_eth_txq_poll_completion_at allows the application to poll for a specific CQE position. It does not support the shared queue feature.

```
template <enum doca_gpu_dev_eth_resource_sharing_mode resource_sharing_mode =
DOCA_GPUNETIO_ETH_RESOURCE_SHARING_MODE_GPU,
      enum doca_gpu_dev_eth_sync_scope sync_scope = DOCA_GPUNETIO_ETH_SYNC_SCOPE_GPU>
__device__ static inline doca_error_t doca_gpu_dev_eth_txq_poll_completion_at(struct doca_gpu_eth_txq *txq,
                                        const uint64_t cqe_idx,
                                        enum doca_gpu_dev_eth_wait_flags wait_mode);
```

Parameters:

- **txq** : Ethernet send queue GPU handler.
- **cqe_idx** : Index of the CQE to poll.
- **wait_mode** : Wait in blocking or non-blocking mode (flags from **doca_gpu_dev_eth_wait_flags**).

5.4.11.7.4 GPU Functions – Verbs

GPUNetIO Verbs data path functions are defined as inline implementations in header files named **doca_gpunetio_dev_verbs_*** .

5.4.11.7.4.1 Header Files

- **doca_gpunetio_verbs_def.h** : Contains constants, enums, and structure definitions.
- **doca_gpunetio_dev_verbs_qp.cuh** : Provides CUDA functions for interacting with a QP, including WQE creation and ringing the network card doorbell.
- **doca_gpunetio_dev_verbs_cq.cuh** : Offers CUDA functions for interacting with a CQ, including CQE polling and validation.
- **doca_gpunetio_dev_verbs_onesided.cuh** : Enables one-sided operations (e.g., **put**) with the shared QP feature at the CUDA thread and warp levels.
- **doca_gpunetio_dev_verbs_twosided.cuh** : Enables two-sided operations (e.g., **send** / **recv**) with the shared QP feature at the CUDA thread and warp levels.

- `doca_gpunetio_dev_verbs_counter.cuh` : Provides CUDA functions for counter-like operations via RDMA atomics with the shared QP feature. This requires "Core Direct" to be enabled as a QP attribute.

Experimental API

The GPUNetIO Verbs APIs are currently experimental. Please report any issues encountered during use, as your feedback is crucial for enhancing the code's quality and robustness.

Private Functions

When using the GPUNetIO Verbs header files, avoid calling functions that start with `doca_priv_*` . These are internal functions intended for use by the public API only.

Debug Macro

The `doca_gpunetio_verbs_def.h` file contains the macro `#define DOCA_GPUNETIO_VERBS_ENABLE_DEBUG 0` . This default setting disables debug prints, including CQE errors. To enable error messages, set this macro to `1` .

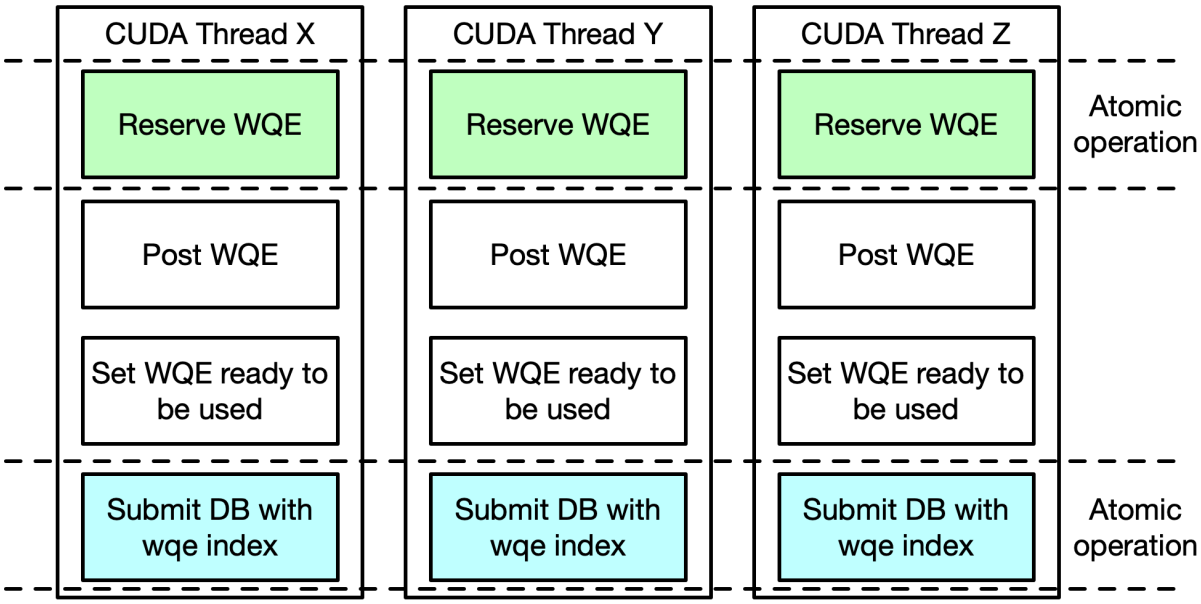
5.4.11.7.4.2 Execution Scope and Shared Queue

The high-level GPUNetIO Verbs API can be executed at a thread or warp scope by specifying a value from the `doca_gpu_dev_verbs_exec_scope` enum as a template argument. This tells the function how many threads are participating in the operation.

The high-level API also supports the shared queue feature, allowing different threads from different scopes to concurrently access the same queue without race conditions.

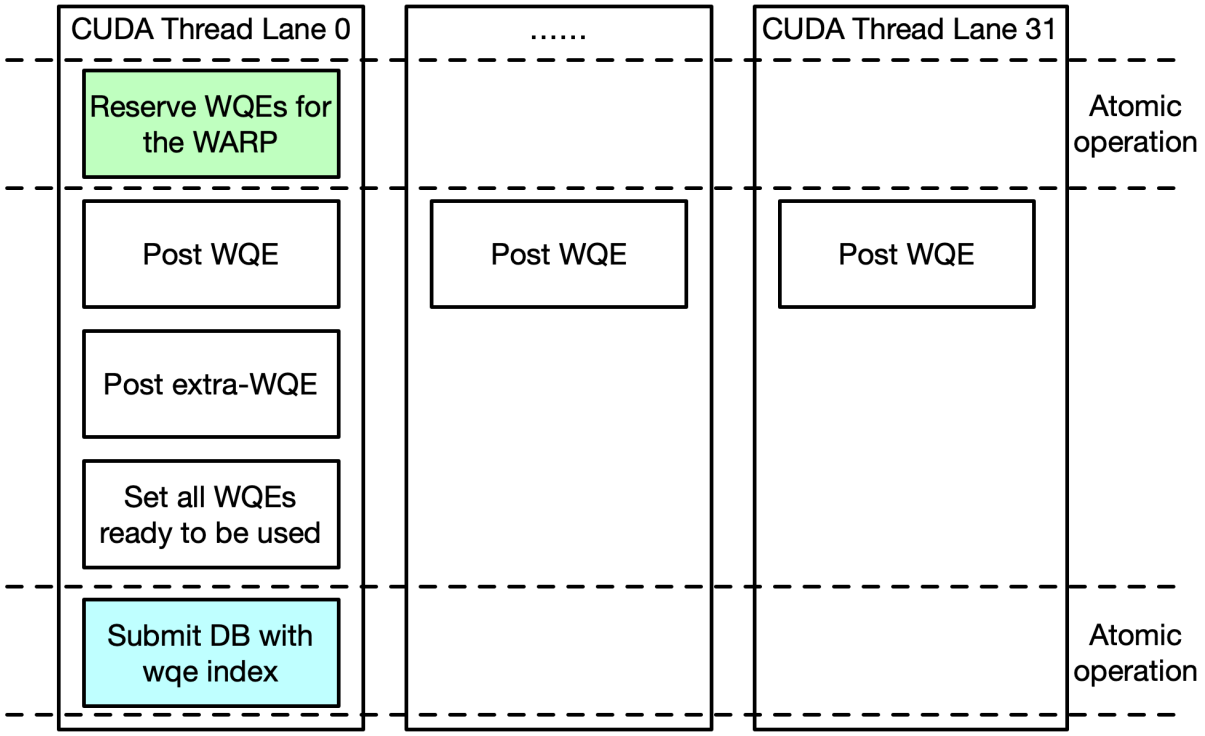
Thread Scope (`DOCA_GPUNETIO_VERBS_EXEC_SCOPE_THREAD`)

- Each thread acts as a standalone entity, posting WQEs and ringing the doorbell.
- This is the slowest approach due to high contention on atomic operations.



Warp Scope (`DOCA_GPUNETIO_VERBS_EXEC_SCOPE_WARP`)

- All threads within the warp must call the function.
- Each thread posts a WQE to a different location, but only the first thread (`lane_idx 0`) performs the submit (doorbell ring).
- This reduces contention to a per-warp basis.



5.4.11.7.4.3 Ring DB and CPU Proxy

The `doca_gpu_dev_verbs_nic_handler` enum, passed as a template argument, determines who is responsible for ringing the doorbell.

The available modes are:

- `DOCA_GPUNETIO_VERBS_NIC_HANDLER_AUTO` : Automatically detects the best doorbell ringing option.
- `DOCA_GPUNETIO_VERBS_NIC_HANDLER_CPU_PROXY` : Enables CPU proxy mode. GPU submit functions provide information to the CPU, which then rings the doorbell.
- `DOCA_GPUNETIO_VERBS_NIC_HANDLER_GPU_SM_DB` : Enables regular GDAKI mode, where CUDA threads ring the doorbell directly.
- `DOCA_GPUNETIO_VERBS_NIC_HANDLER_GPU_SM_BF` : Experimental BlueFlame doorbell ringing mode (may fail in the current release).

⚠ CPU Proxy Mode When `CPU_PROXY` mode is enabled, a CPU thread must invoke `doca_gpu_verbs_cpu_proxy_progress` in a loop to detect the GPU's information and ring the doorbell. This mode can be enabled via a command-line option in most GPUNetIO Verbs samples.

5.4.11.7.4.4 Memory Consistency (MCST) Algorithm

During an RDMA Read or Receive operation, the network card writes data via PCIe into the application's mapped memory. For pre-Hopper GPUs (see CUGPUDirectRDMAWritesOrdering), memory consistency (MCST) must be ensured when a CUDA kernel is receiving data into GPU memory.

DOCA GPUNetIO Verbs applications can enable the MCST algorithm on the GPU by setting the **doca_gpu_dev_verbs_mcst_mode mcst_mode** template parameter. This parameter is available in functions such as **doca_gpu_dev_verbs_get** , **doca_gpu_dev_verbs_get_wait** , and **doca_gpu_dev_verbs_recv_wait** .

The **gpunetio_verbs_twosided_bw** sample demonstrates this by checking the CUDA architecture:

```
#if __CUDA_ARCH__ < 900
    doca_gpu_dev_verbs_recv_wait<DOCA_GPUNETIO_VERBS_RESOURCE_SHARING_MODE_GPU,
DOCA_GPUNETIO_VERBS_NIC_HANDLER_AUTO, DOCA_GPUNETIO_VERBS_MCST_ENABLED>(
                                qp, doca_gpu_dev_verbs_addr{.addr = (uint64_t)dump_flag, .key = dump_flag_mkey});
#else
    doca_gpu_dev_verbs_recv_wait<DOCA_GPUNETIO_VERBS_RESOURCE_SHARING_MODE_GPU,
DOCA_GPUNETIO_VERBS_NIC_HANDLER_AUTO, DOCA_GPUNETIO_VERBS_MCST_DISABLED>(
                                qp, doca_gpu_dev_verbs_addr{.addr = 0, .key = 0});
#endif
```

5.4.11.7.4.5 Queue Type

Many functions (e.g., submit doorbell, poll CQ) can apply to either the Send Queue or Receive Queue of a QP. A template argument from the **doca_gpu_dev_verbs_qp_type** enum is used to specify which queue to use.

- **DOCA_GPUNETIO_VERBS_QP_SQ** : Uses the Send Queue from the input QP.
- **DOCA_GPUNETIO_VERBS_QP_RQ** : Uses the Receive Queue from the input QP.

5.4.11.7.4.6 ConnectX-8 Reliable Doorbell

The ConnectX-8 Reliable Doorbell feature optimizes the send data path by allowing applications to skip the Queue Pair Doorbell Record (QP DBREC) update. This reduces the number of operations the GPU must execute to ring the doorbell for send WQEs (Write, Read, Send, etc.).

The implementation steps are as follows:

1. Verify support: Check if the device supports the feature using **doca_verbs_device_attr_get_is_send_dbr_mode_supported()** .
2. Enable feature: Set the new QP attribute using **doca_verbs_qp_init_attr_set_send_dbr_mode()** .
3. Handle GPU data path: If enabled, use the GPUNetIO handler mode **DOCA_GPUNETIO_VERBS_NIC_HANDLER_GPU_SM_NO_DBR** .

The **gpunetio_verbs_write_lat** and **gpunetio_verbs_put_counter_lat** examples demonstrate this feature. Enable it via the command line option **-r 1** .

5.4.11.7.4.7 Header: doca_gpunetio_dev_verbs_qp.cuh

This header provides functions for QP manipulation, categorized into three groups:

- **doca_gpu_dev_verbs_wqe_prepare_*** : Prepares and posts various RDMA WQEs (Write, Read, Send, Recv, Atomic, Wait, Dump) on QP memory.
- **doca_gpu_dev_verbs_submit** : Updates network card registers for doorbell submission (regular, CPU proxy, and BlueFlame) for both SQ and RQ.
- **doca_gpu_dev_verbs_wait_*** : Waits for CQEs to arrive on the SQ or RQ's completion queue.

For detailed explanations, refer to the header file. Examples can be found in the **doca_gpunetio_verbs_write_bw** and **doca_gpunetio_verbs_write_lat** samples.

5.4.11.7.4.8 Header: doca_gpunetio_dev_verbs_cq.cuh

This header provides functions for manipulating Completion Queues (CQs). Specifically, the **doca_gpu_dev_verbs_poll_*** functions are used to poll CQEs, waiting for the completion of WQEs in the connected QP.

5.4.11.7.4.9 Header: doca_gpunetio_dev_verbs_onesided.cuh

This header provides shared-QP-aware functions for one-sided RDMA operations at the thread and warp levels.

- **doca_gpu_dev_verbs_put** : Posts an RDMA Write WQE and submits. In **WARP** scope, only the first thread submits.
- **doca_gpu_dev_verbs_put_signal** : Similar to **put** , but adds an RDMA Atomic Fetch and Add. In THREAD scope, each thread posts a Write, an Atomic, and submits. In WARP scope, each thread posts a Write, but only the first thread posts the Atomic and submits.
- **doca_gpu_dev_verbs_get** : Posts an RDMA Read WQE and submits. In **WARP** scope, only the first thread submits. Optionally posts a Dump WQE to ensure data consistency (e.g., ensures Read data is present in memory after CQE arrival, e.g., enum CUGPUDirectRDMAWritesOrdering).

For details, refer to the header file. Examples are in the **doca_gpunetio_verbs_put_bw** , **doca_gpunetio_verbs_put_signal_bw** , and **doca_gpunetio_verbs_put_signal_lat** samples.

5.4.11.7.4.10 Header: doca_gpunetio_dev_verbs_twosided.cuh

This header provides shared-QP-aware functions for two-sided RDMA operations at the thread and warp levels.

- **doca_gpu_dev_verbs_send** : Posts an RDMA Send WQE and submits. In WARP scope, only the first thread submits.
- **doca_gpu_dev_verbs_recv** : Posts an RDMA Receive WQE and submits. In WARP scope, only the first thread submits.

Requirement flow for two-sided communication:

1. Peer A posts one or more RDMA Receive WQEs.
2. Peer A notifies Peer B that it is ready to receive.
3. Peer B waits for this notification.
4. Peer B posts one or more RDMA Send WQEs.

The notification can be sent via RDMA Write or Atomic. An example using RDMA Write is in the **doca_gpunetio_verbs_twosided_bw** sample.

5.4.11.7.4.11 Header: doca_gpunetio_dev_verbs_counter.cuh

This header provides shared-QP functions that use the counter feature. This allows a CQE arrival on a main QP to trigger a WQE on a companion QP. This requires Core Direct to be enabled via **doca_verbs_qp_init_attr_set_core_direct_master()** .

- **doca_gpu_dev_verbs_put_counter** : Posts an RDMA Write on the main QP and an RDMA Atomic FetchAdd on the companion QP. The Atomic is executed only after the Write's CQE arrives, allowing for buffer reuse.
- **doca_gpu_dev_verbs_submit_multi_qps** : Submits both the main and companion QP in a single call.

For details, refer to the header file. Examples are in the **doca_gpunetio_verbs_put_counter_bw** and **doca_gpunetio_verbs_put_counter_lat** samples.

5.4.11.7.5 GPU Functions – RDMA

This section provides a list of DOCA GPUNetIO functions that can be used on the GPU only within a CUDA kernel to execute RDMA operations. These functions offer a strong and a weak mode.

```
__device__ doca_error_t __device__ doca_error_t doca_gpu_dev_rdma_get_info(struct doca_gpu_dev_rdma *rdma, uint32_t
connection_index, uint32_t *curr_position, uint32_t *mask_max_position)
```

- **rdma** – RDMA queue GPU handler
- **connection_index** – In case of RDMA CM, the connection index must be specified. By default, it is 0.
- **curr_position** – Next available position in the queue
- **mask_max_position** – Mask of the total number of positions in the queue

```
__device__ doca_error_t __device__ doca_error_t doca_gpu_dev_rdma_recv_get_info(struct doca_gpu_dev_rdma_r *rdma_r,
uint32_t *curr_position, uint32_t *mask_max_position)
```

- **rdma_r** – RDMA receive queue GPU handler
- **curr_position** – Next available position in the queue
- **mask_max_position** – Mask of the total number of positions in the queue

5.4.11.7.5.1 doca_gpu_dev_rdma_write_*

To RDMA write data onto a remote memory location from a CUDA kernel, DOCA GPUNetIO offers strong and weak modes for enqueuing operations on the RDMA queue. For both modes, the scope is the single CUDA thread.

```
__device__ doca_error_t doca_gpu_dev_rdma_write_strong(struct doca_gpu_dev_rdma *rdma,
                                                       uint32_t connection_index,
                                                       struct doca_gpu_buf *remote_buf, uint64_t remote_offset,
                                                       struct doca_gpu_buf *local_buf, uint64_t local_offset,
                                                       size_t length, uint32_t imm,
                                                       const enum doca_gpu_dev_rdma_write_flags flags)
```

- **rdma** – RDMA queue GPU handler
- **connection_index** – In case of RDMA CM, the connection index must be specified. By default, it is 0.
- **remote_buf** – Remote DOCA buffer from a DOCA GPU buffer array to write data to
- **remote_offset** – Offset, in bytes, to write data to in the remote buffer
- **local_buf** – Local DOCA buffer from a DOCA GPU buffer array from which to fetch data to write
- **local_offset** – Offset, in bytes, to fetch data from in the local buffer
- **length** – Number of bytes to write
- **imm** – Immediate value **uint32_t**
- **flags** – One of the flags in the **doca_gpu_dev_rdma_write_flags** enum

```
__device__ doca_error_t doca_gpu_dev_rdma_write_weak(struct doca_gpu_dev_rdma *rdma,
                                                     uint32_t connection_index,
                                                     struct doca_gpu_buf *remote_buf, uint64_t remote_offset,
                                                     struct doca_gpu_buf *local_buf, uint64_t local_offset,
                                                     size_t length, uint32_t imm,
                                                     const enum doca_gpu_dev_rdma_write_flags flags,
                                                     uint32_t position);
```

- **rdma** – RDMA queue GPU handler
- **connection_index** – In case of RDMA CM, the connection index must be specified. By default, it is 0.
- **remote_buf** – Remote DOCA buffer from a DOCA GPU buffer array to write data to
- **remote_offset** – Offset, in bytes, to write data to in the remote buffer
- **local_buf** – Local DOCA buffer from a DOCA GPU buffer array where to fetch data to write
- **local_offset** – Offset, in bytes, to fetch data in the local buffer
- **length** – Number of bytes to write
- **imm** – Immediate value **uint32_t**
- **flags** – One of the flags in the **doca_gpu_dev_rdma_write_flags** enum
- **position** – Position in the queue to place the RDMA operation. Range: 0 - **mask_max_position**.

5.4.11.7.5.2 doca_gpu_dev_rdma_read_*

To RDMA read data onto a remote memory location from a CUDA kernel, DOCA GPUNetIO offers strong and weak modes to enqueue operations on the RDMA queue. For both modes, the scope is the single CUDA thread.

```
__device__ doca_error_t doca_gpu_dev_rdma_read_strong(struct doca_gpu_dev_rdma *rdma,
                                                      uint32_t connection_index,
                                                      struct doca_gpu_buf *remote_buf, uint64_t remote_offset,
                                                      struct doca_gpu_buf *local_buf, uint64_t local_offset,
                                                      size_t length,
                                                      const uint32_t flags_bitmask)
```

- **rdma** – RDMA queue GPU handler
- **connection_index** – In case of RDMA CM, the connection index must be specified. By default, it is 0.
- **remote_buf** – Remote DOCA buffer from a DOCA GPU buffer array where to read data
- **remote_offset** – Offset in bytes to read data to in the remote buffer

- **local_buf** – Local DOCA buffer from a DOCA GPU buffer array where to store remote data
- **local_offset** – Offset in bytes to store data in the local buffer
- **length** – Number of bytes to be read
- **flags_bitmask** – Must be 0; reserved for future use

```
__device__ doca_error_t doca_gpu_dev_rdma_read_weak(struct doca_gpu_dev_rdma *rdma,
                                                    uint32_t connection_index,
                                                    struct doca_gpu_buf *remote_buf, uint64_t remote_offset,
                                                    struct doca_gpu_buf *local_buf, uint64_t local_offset,
                                                    size_t length,
                                                    const uint32_t flags_bitmask,
                                                    uint32_t position);
```

- **rdma** – RDMA queue GPU handler
- **connection_index** – In case of RDMA CM, the connection index must be specified. By default, it is 0.
- **remote_buf** – Remote DOCA buffer from a DOCA GPU buffer array where to read data
- **remote_offset** – Offset in bytes to read data to in the remote buffer
- **local_buf** – Local DOCA buffer from a DOCA GPU buffer array where to store remote data
- **local_offset** – Offset in bytes to store data in the local buffer
- **length** – Number of bytes to be read
- **flags_bitmask** – Must be 0; reserved for future use
- **position** – Position in the queue to place the RDMA operation. Range: 0 - **mask_max_position**.

5.4.11.7.5.3 doca_gpu_dev_rdma_send_*

To RDMA send data from a CUDA kernel, DOCA GPUNetIO offers strong and weak modes for enqueuing operations on the RDMA queue. For both modes, the scope is the single CUDA thread.

```
__device__ doca_error_t doca_gpu_dev_rdma_send_strong(struct doca_gpu_dev_rdma *rdma,
                                                      uint32_t connection_index,
                                                      struct doca_gpu_buf *local_buf, uint64_t local_offset,
                                                      size_t length, uint32_t imm,
                                                      const enum doca_gpu_dev_rdma_write_flags flags)
```

- **rdma** – RDMA queue GPU handler
- **connection_index** – In case of RDMA CM, the connection index must be specified. By default, it is 0.
- **local_buf** – Local DOCA buffer from a DOCA GPU buffer array from which to fetch data to send
- **local_offset** – Offset in bytes to fetch data in the local buffer
- **length** – Number of bytes to send
- **imm** – Immediate value **uint32_t**
- **flags** – One of the flags in the **doca_gpu_dev_rdma_write_flags** enum

```
__device__ doca_error_t doca_gpu_dev_rdma_send_weak(struct doca_gpu_dev_rdma *rdma,
                                                    uint32_t connection_index,
                                                    struct doca_gpu_buf *local_buf, uint64_t local_offset,
                                                    size_t length, uint32_t imm,
                                                    const enum doca_gpu_dev_rdma_write_flags flags,
                                                    uint32_t position);
```

- **rdma** – RDMA queue GPU handler

- **connection_index** – In case of RDMA CM, the connection index must be specified. By default, it is 0.
- **local_buf** – Local DOCA buffer from a DOCA GPU buffer array from which to fetch data to send
- **local_offset** – Offset in bytes to fetch data in the local buffer
- **length** – Number of bytes to send
- **imm** – Immediate value **uint32_t**
- **flags** – One of the flags in the **doca_gpu_dev_rdma_write_flags** enum
- **position** – Position in the queue to place the RDMA operation. Range: 0 - **mask_max_position**.

5.4.11.7.5.4 doca_gpu_dev_rdma_commit_*

Once all RDMA write, send or read requests have been enqueue in the RDMA queue, a synchronization point must be reached to consolidate and execute those requests. Only 1 CUDA thread can invoke this function at a time.

```
__device__ doca_error_t doca_gpu_dev_rdma_commit_strong(struct doca_gpu_dev_rdma *rdma, uint32_t connection_index)
```

- **rdma** – RDMA queue GPU handler
- **connection_index** – In case of RDMA CM, the connection index must be specified. By default, it is 0.

```
__device__ doca_error_t doca_gpu_dev_rdma_commit_weak(struct doca_gpu_dev_rdma *rdma, uint32_t connection_index,
uint32_t num_ops)
```

- **rdma** – RDMA queue GPU handler
- **connection_index** – In case of RDMA CM, the connection index must be specified. By default, it is 0.
- **num_ops** – Number of RDMA requests enqueued since the last commit

5.4.11.7.5.5 doca_gpu_dev_rdma_wait_all

After a commit, RDMA requests are executed by the network card as applications move forward doing other operations. If the application needs to verify all RDMA operations have been done by the network card, this "wait all" function can be used to wait for all previous posted operations. Only 1 CUDA thread can invoke this function at a time.

```
__device__ doca_error_t doca_gpu_dev_rdma_wait_all(struct doca_gpu_dev_rdma *rdma, uint32_t *num_commits)
```

- **rdma** – RDMA queue GPU handler
- **num_commits** – Output parameter; the number of commit operations completed

This function is optional, and it can be used to ensure all the RDMA Send/Write/Read operations have actually been executed before moving forward with the application.

5.4.11.7.5.6 doca_gpu_dev_rdma_recv_*

To receive data from an RDMA send, send with immediate, or write with immediate, the destination peer should post a receive operation. DOCA GPUNetIO RDMA receive operations must be done with a **doca_gpu_dev_rdma_r** handler. This handler can be obtained with the function **doca_gpu_dev_rdma_get_recv**.

> ⚠ All receive operations must use this object.

```
__device__ doca_error_t doca_gpu_dev_rdma_get_recv(struct doca_gpu_dev_rdma *rdma, struct doca_gpu_dev_rdma_r
**rdma_r)
```

- **rdma** – RDMA queue GPU handler
- **rdma_r** – RDMA receive queue GPU handler

Even for the receive side, in this case, DOCA GPUNetIO offers strong and weak modes for enqueuing operations on the RDMA queue. For both modes, the scope is the single CUDA thread.

```
__device__ doca_error_t doca_gpu_dev_rdma_recv_strong(struct doca_gpu_dev_rdma_r *rdma_r,
                                                      struct doca_gpu_buf *recv_buf,
                                                      size_t recv_length,
                                                      uint64_t recv_offset,
                                                      const uint32_t flags_bitmask)
```

- **rdma_r** – RDMA receive queue GPU handler
- **recv_buf** – Local DOCA buffer from a DOCA GPU buffer array from which to fetch data to send
- **recv_length** – Number of bytes to send
- **recv_offset** – Offset in bytes to fetch data in the local buffer
- **flags_bitmask** – Must be 0; reserved for future use

```
__device__ doca_error_t doca_gpu_dev_rdma_recv_weak(struct doca_gpu_dev_rdma_r *rdma_r,
                                                    struct doca_gpu_buf *recv_buf,
                                                    size_t recv_length,
                                                    uint64_t recv_offset,
                                                    const uint32_t flags_bitmask,
                                                    uint32_t position);
```

- **rdma_r** – RDMA receive queue GPU handler
- **recv_buf** – Local DOCA buffer from a DOCA GPU buffer array from which to fetch data to send
- **recv_length** – Number of bytes to send
- **recv_offset** – Offset in bytes to fetch data in the local buffer
- **flags_bitmask** - Must be 0; reserved for future use
- **position** – Position in the queue to place the RDMA operation. Range: 0 - **mask_max_position**.

5.4.11.7.5.7 doca_gpu_dev_rdma_recv_commit_*

After posting several RDMA receive operations, a commit function must be invoked to activate the receive in the queue. Only 1 CUDA thread can invoke this function at a time.

```
__device__ doca_error_t doca_gpu_dev_rdma_recv_commit_strong(struct doca_gpu_dev_rdma_r *rdma_r)
```

- **rdma_r** – RDMA receive queue GPU handler

```
__device__ doca_error_t doca_gpu_dev_rdma_recv_commit_weak(struct doca_gpu_dev_rdma_r *rdma_r, uint32_t num_ops)
```

- **rdma_r** – RDMA receive queue GPU handler
- **num_ops** – Number of RDMA receive requests enqueued since the last commit

5.4.11.7.5.8 doca_gpu_dev_rdma_recv_wait_all

This function waits for the completion of all previously posted RDMA receive operation. Only 1 CUDA thread can invoke this function at a time. It works in blocking or non-blocking mode.

```
 enum doca_gpu_dev_rdma_recv_wait_flags {
    DOCA_GPU_RDMA_RECV_WAIT_FLAG_NB = 0,  /**< Non-Blocking mode: the wait receive function
doca_gpu_dev_rdma_recv_wait
                                           *  checks if the receive operation happened (data has been received)
                                           *  and exit from the function. If nothing has been received,
                                           *  the function doesn't block the execution.
                                           */
    DOCA_GPU_RDMA_RECV_WAIT_FLAG_B  = 1, /**< Blocking mode: the wait receive function doca_gpu_dev_rdma_recv_wait
                                           *  blocks the execution waiting for the receive operations to be
executed.
                                           */
};
```

Function:

```
__device__ doca_error_t doca_gpu_dev_rdma_recv_wait_all(struct doca_gpu_dev_rdma_r *rdma_r, const enum
 doca_gpu_dev_rdma_recv_wait_flags flags, uint32_t *num_ops, uint32_t *imm_val)
```

- **rdma_r** – RDMA receive queue GPU handler
- **flags** – receive flags
- **num_ops** – Output parameter. Function reports number of completed operations.
- **imm_val** – Output parameter. Application-provided buffer where the function can store received immediate values, if any (or 0xFFFFFFFF if no immediate value is received). If **nullptr**, the function ignores this parameter.

5.4.11.7.6 GPU Functions – DMA

This section provides a list of DOCA GPUNetIO functions that can be used on the GPU only within a CUDA kernel to execute DMA operations.

5.4.11.7.6.1 doca_gpu_dev_dma_memcpy

This function allows a CUDA kernel to trigger a DMA memory copy operation through the DMA GPU engine. There is no strong/weak mode here, the DMA is assuming the strong behavior by default.

```
__device__ doca_error_t doca_gpu_dev_dma_memcpy(struct doca_gpu_dma *dma, struct doca_gpu_buf *src_buf, uint64_t
src_offset, struct doca_gpu_buf *dst_buf, uint64_t dst_offset, size_t length);
```

- **dma** – DMA queue GPU handler
- **src_buf** – memcpy source buffer

- **src_offset** – fetch data starting from this source buffer offset
- **dst_buf** – memcpy destination buffer
- **dst_offset** – copy data starting from this destination buffer offset
- **lenght** – number of bytes to copy

5.4.11.7.6.2 doca_gpu_dev_dma_commit

After posting several DMA memory copies, a commit function must be invoked to execute the operations enqueued in the DMA queue. Only 1 CUDA thread can invoke this function at a time.

```
__device__ doca_error_t doca_gpu_dev_dma_commit(struct doca_gpu_dma *dma);
```

- **dma** – DMA queue GPU handler

5.4.11.7.7 GPU Functions – Comch

This section lists the DOCA GPUNetIO functions that can be used within a CUDA kernel to execute DOCA Comch Producer or Consumer operations. For more information on the Producer/Consumer model, refer to the DOCA Comch documentation.

Threading Limitation

Only one CUDA thread can invoke these functions per Producer or Consumer object.

5.4.11.7.7.1 Consumer: doca_dev_gpu_comch_consumer_post_recv

This function allows a DOCA Comch Consumer to post a buffer to receive a message from a remote producer.

```
__device__ doca_error_t doca_dev_gpu_comch_consumer_post_recv(
    struct doca_comch_gpu_consumer *consumer,
    uint32_t mkey,
    uintptr_t addr,
    uint32_t recv_len
);
```

Parameters:

- **consumer** : The Consumer GPU handle.
- **mkey** : The mkey associated with the buffer to be posted.
- **addr** : The address of the buffer to receive the message into.
- **recv_len** : The maximum message size that can be written to the buffer.

5.4.11.7.7.2 Consumer: doca_dev_gpu_comch_consumer_recv_wait

This function allows a DOCA Comch Consumer to wait for a message to be received from a remote producer.

```
__device__ doca_error_t doca_dev_gpu_comch_consumer_recv_wait(
    struct doca_comch_gpu_consumer *consumer,
    uint32_t *mkey,
    uintptr_t *addr,
    uint32_t *recv_len,
    uint8_t **imm_data,
```

```
    uint32_t *imm_data_len,
    const enum doca_gpu_dev_comch_wait_flags flags
);
```

Parameters:

- **con sumer** : The Consumer GPU handle.
- **mkey** [out]: The mkey of the buffer that the message has been received into.
- **addr** [out]: The address of the buffer that the message has been received into.
- **recv_len** [out]: The length of the message that has been received.
- **imm_data** : Pointer to a buffer for any immediate data included with the message.
- **imm_data_len** [in/out]:
  - Input: The maximum size of immediate data that can be received.
  - Output: The actual size of immediate data received.
- **flags** : A **doca_gpu_dev_comch_wait_flags** value (e.g., blocking or non-blocking).

5.4.11.7.7.3 Producer: doca_dev_gpu_comch_producer_send

This function allows a DOCA Comch Producer to send a message to a remote consumer.

```
__device__ doca_error_t doca_dev_gpu_comch_producer_send(
    struct doca_comch_gpu_producer *producer,
    uint32_t mkey,
    uintptr_t addr,
    uint32_t send_len,
    uint8_t *imm_data,
    uint32_t imm_data_len,
    uint32_t consumer_id,
    uint64_t user_msg_id
);
```

Paramters:

- **producer** : The Producer GPU handle.
- **mkey** : The mkey associated with the buffer to be sent.
- **addr** : The address of the buffer to be sent.
- **send_len** : The number of bytes to send from the buffer.
- **imm_data** : Immediate data to include with the message.
- **imm_data_len** : Size of the immediate data to include.
- **consumer_id** : The ID of the remote consumer to send the message to.
- **user_msg_id** : A user-defined ID that will be returned by **doca_dev_gpu_comch_producer_poll** when this message has been successfully sent.

5.4.11.7.7.4 Producer: doca_dev_gpu_comch_producer_poll

This function allows a DOCA Comch Producer to poll for the completion of a send operation.

```
__device__ doca_error_t doca_dev_gpu_comch_producer_poll(
    struct doca_comch_gpu_producer *producer,
    uint64_t *user_msg_id,
    const enum doca_gpu_dev_comch_wait_flags flags
);
```

Parameters:

- **producer** : The Producer GPU handle.

- **user_msg_id** [out]: The **user_msg_id** that was set in the corresponding **doca_dev_gpu_comch_producer_send** function.
- **flags** : A **doca_gpu_dev_comch_wait_flags** value (e.g., blocking or non-blocking).

5.4.11.7.7.5 doca_gpu_dev_comch_wait_flags

This **enum** is used as the **flags** parameter in both the **doca_gpu_dev_comch_consumer_recv_wait** and **doca_gpu_dev_comch_producer_poll** functions. It determines how the function behaves if no operation has completed.

```
enum doca_gpu_dev_comch_wait_flags {
    DOCA_GPU_COMCH_WAIT_FLAG_NB = 0, /**< Non-Blocking mode: the wait function  checks if any of the operations
                                      * have been completed (data has been sent/received) and exit from the
                                      * function. If nothing has been sent/received, the function doesn't block
                                      * the execution.
                                      */
    DOCA_GPU_COMCH_WAIT_FLAG_B = 1,  /**< Blocking mode: the wait function blocks the execution waiting for a
                                      * send/receive operations to be completed.
                                      */
};
```

5.4.11.8 GPUNetIO Sample Guide

This section contains two samples that show how to enable simple GPUNetIO features. Be sure to correctly set the following environment variables:

Build the sample

```
export PATH=${PATH}:/usr/local/cuda/bin
export CPATH="$(echo /usr/local/cuda/targets/{x86_64,sbsa}-linux/include | sed 's/ /:/'):${CPATH}"
export PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/usr/lib/pkgconfig:/opt/mellanox/grpc/lib/{x86_64,aarch64}-linux-gnu/
pkgconfig:/opt/mellanox/dpdk/lib/{x86_64,aarch64}-linux-gnu/pkgconfig:/opt/mellanox/doca/lib/{x86_64,aarch64}-
linux-gnu/pkgconfigexport LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/local/cuda/lib64:/opt/mellanox/gdrcopy/src:/opt/
mellanox/dpdk/lib/{x86_64,aarch64}-linux-gnu:/opt/mellanox/doca/lib/{x86_64,aarch64}-linux-gnu
```

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

GPU architecture

Please ensure the arch of your GPU is included in the **meson.build** file before building the samples (e.g., **sm_80** for Ampere, **sm_89** for L40, **sm_90** for H100, etc).

5.4.11.8.1 Ethernet Send Wait Time

The sample shows how to enable Accurate Send Scheduling (or wait-on-time) feature in the context of a GPUNetIO application. Accurate Send Scheduling is the ability of an NVIDIA NIC to send packets in the future according to application-provided timestamps.

This feature is supported on ConnectX-7 and later.

This sample demonstrates how to send packets from the GPU using Accurate Send Scheduling by calling the high-level **doca_gpu_dev_eth_txq_wait_send** function with a **BLOCK** execution scope.

> This NVIDIA blog post offers an example for how this feature has been used in 5G networks.

5.4.11.8.1.1 Synchronizing Clocks

Before starting the sample, it is important to properly synchronize the CPU clock with the NIC clock. This way, timestamps provided by the system clock are synchronized with the time in the NIC.

For this purpose, at least the **phc2sys** service must be used. To install it on an Ubuntu system:

phc2sys

```
sudo apt install linuxptp
```

To start the **phc2sys** service properly, a config file must be created in **/lib/systemd/system/phc2sys.service**. Assuming the network interface is **ens6f0** :

phc2sys

```
[Unit]
Description=Synchronize system clock or PTP hardware clock (PHC)
Documentation=man:phc2sys

[Service]
Restart=always
RestartSec=5s
Type=simple
ExecStart=/bin/sh -c "taskset -c 15 /usr/sbin/phc2sys -s /dev/ptp$(ethtool -T ens6f0 | grep PTP | awk '{print $4}')
-c CLOCK_REALTIME -n 24 -O 0 -R 256 -u 256"

[Install]
WantedBy=multi-user.target
```

Now **phc2sys** service can be started:

phc2sys

```
sudo systemctl stop systemd-timesyncd
sudo systemctl disable systemd-timesyncd
sudo systemctl daemon-reload
sudo systemctl start phc2sys.service
```

To check the status of **phc2sys**:

phc2sys

```
$ sudo systemctl status phc2sys.service
```

Output:

phc2sys

```
● phc2sys.service - Synchronize system clock or PTP hardware clock (PHC)
     Loaded: loaded (/lib/systemd/system/phc2sys.service; disabled; vendor preset: enabled)
     Active: active (running) since Mon 2023-04-03 10:59:13 UTC; 2 days ago
       Docs: man:phc2sys
   Main PID: 337824 (sh)
      Tasks: 2 (limit: 303788)
     Memory: 560.0K
        CPU: 52min 8.199s
     CGroup: /system.slice/phc2sys.service
             337824 /bin/sh -c "taskset -c 15 /usr/sbin/phc2sys -s /dev/ptp\$(ethtool -T enp23s0f1np1 | grep PTP
| awk '{print \$4}') -c CLOCK_REALTIME -n 24 -O 0 -R >
             337829 /usr/sbin/phc2sys -s /dev/ptp3 -c CLOCK_REALTIME -n 24 -O 0 -R 256 -u 256

Apr 05 16:35:52 doca-vr-045 phc2sys[337829]: [457395.040] CLOCK_REALTIME rms    8 max   18 freq +110532 +/-  27
delay   770 +/-   3
Apr 05 16:35:53 doca-vr-045 phc2sys[337829]: [457396.071] CLOCK_REALTIME rms    8 max   20 freq +110513 +/-  30
delay   769 +/-   3
Apr 05 16:35:54 doca-vr-045 phc2sys[337829]: [457397.102] CLOCK_REALTIME rms    8 max   18 freq +110527 +/-  30
delay   769 +/-   3
Apr 05 16:35:55 doca-vr-045 phc2sys[337829]: [457398.130] CLOCK_REALTIME rms    8 max   18 freq +110517 +/-  31
delay   769 +/-   3
Apr 05 16:35:56 doca-vr-045 phc2sys[337829]: [457399.159] CLOCK_REALTIME rms    8 max   19 freq +110523 +/-  32
delay   770 +/-   3
Apr 05 16:35:57 doca-vr-045 phc2sys[337829]: [457400.191] CLOCK_REALTIME rms    8 max   20 freq +110528 +/-  33
delay   770 +/-   3
Apr 05 16:35:58 doca-vr-045 phc2sys[337829]: [457401.221] CLOCK_REALTIME rms    8 max   19 freq +110512 +/-  38
delay   770 +/-   3
Apr 05 16:35:59 doca-vr-045 phc2sys[337829]: [457402.253] CLOCK_REALTIME rms    9 max   20 freq +110538 +/-  47
delay   770 +/-   4
Apr 05 16:36:00 doca-vr-045 phc2sys[337829]: [457403.281] CLOCK_REALTIME rms    8 max   21 freq +110517 +/-  38
delay   769 +/-   3
Apr 05 16:36:01 doca-vr-045 phc2sys[337829]: [457404.311] CLOCK_REALTIME rms    8 max   17 freq +110526 +/-  26
delay   769 +/-   3
...
```

At this point, the system and NIC clocks are synchronized so timestamps provided by the CPU are correctly interpreted by the NIC.

The timestamps you get may not reflect the real time and day. To get that, you must properly set the **ptp4l** service with an external grand master on the system. Doing that is out of the scope of this sample.

5.4.11.8.1.2 Running the Sample

To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

phc2sys

```
# Ensure DOCA is in the pkgconfig environment variable
cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_send_wait_time
meson build
ninja -C build
```

The sample sends 8 bursts of 32 raw Ethernet packets or 1kB to a dummy Ethernet address, **10:11:12:13:14:15**, in a timed way. Program the NIC to send every **t** nanoseconds (command line option **-t**).

The following example programs a system with GPU PCIe address **ca:00.0** and NIC PCIe address **17:00.0** to send 32 packets every 5 milliseconds:

Run

```
# Ensure DOCA and DPDK are in the LD_LIBRARY_PATH environment variable
```

```
$ sudo ./build/doca_gpunetio_send_wait_time -n 17:00.0 -g ca:00.0 -t 5000000[09:22:54:165778][1316878][DOCA][INF]
[gpunetio_send_wait_time_main.c:195][main] Starting the sample
[09:22:54:438260][1316878][DOCA][INF][gpunetio_send_wait_time_main.c:224][main] Sample configuration:
        GPU ca:00.0
        NIC 17:00.0
        Timeout 5000000ns
EAL: Detected CPU lcores: 128
...
EAL: Probe PCI driver: mlx5_pci (15b3:a2d6) device: 0000:17:00.0 (socket 0)
[09:22:54:819996][1316878][DOCA][INF][gpunetio_send_wait_time_sample.c:607][gpunetio_send_wait_time] Wait on time
 supported mode: DPDK
EAL: Probe PCI driver: gpu_cuda (10de:20b5) device: 0000:ca:00.0 (socket 1)
[09:22:54:830212][1316878][DOCA][INF][gpunetio_send_wait_time_sample.c:252][create_tx_buf] Mapping send queue
buffer (0x0x7f48e32a0000 size 262144B) with legacy nvidia-peermem mode
[09:22:54:832462][1316878][DOCA][INF][gpunetio_send_wait_time_sample.c:657][gpunetio_send_wait_time] Launching CUDA
kernel to send packets
[09:22:54:842945][1316878][DOCA][INF][gpunetio_send_wait_time_sample.c:664][gpunetio_send_wait_time] Waiting 10 sec
for 256 packets to be sent
[09:23:04:883309][1316878][DOCA][INF][gpunetio_send_wait_time_sample.c:684][gpunetio_send_wait_time] Sample
finished successfully
[09:23:04:883339][1316878][DOCA][INF][gpunetio_send_wait_time_main.c:239][main] Sample finished successfully
```

To verify that packets are actually sent at the right time, use a packet sniffer on the other side (e.g., **tcpdump**):

phc2sys

```
$ sudo tcpdump -i enp23s0f1np1 -A -s 64

17:12:23.480318 IP5 (invalid)
Sent from DOCA GPUNetIO.........................
....
17:12:23.480368 IP5 (invalid)
Sent from DOCA GPUNetIO.........................
# end of first burst of 32 packets, bump to +5ms
17:12:23.485321 IP5 (invalid)
Sent from DOCA GPUNetIO.........................
...
17:12:23.485369 IP5 (invalid)
Sent from DOCA GPUNetIO.........................
# end of second burst of 32 packets, bump to +5ms
17:12:23.490278 IP5 (invalid)
Sent from DOCA GPUNetIO.........................
...
```

The output should show a jump of approximately 5 milliseconds every 32 packets.

⚠ **tcpdump** may increase latency in sniffing packets and reporting the receive timestamp, so the difference between bursts of 32 packets reported may be less than expected, especially with small interval times like 500 microseconds (**-t 500000**).

5.4.11.8.2 Ethernet Simple Receive

This sample application demonstrates the fundamental steps to build a DOCA GPUNetIO receiver application. It creates one queue for UDP packets and uses a single CUDA kernel to receive those packets from the GPU.

The sample uses the high-level **doca_gpu_dev_eth_rxq_recv** function, and its execution scope (thread, warp, or block) can be set at runtime using the **-e** command-line parameter.

printf in CUDA Kernels

Invoking **printf** from a CUDA kernel is not good practice for release software as it slows down the kernel's overall execution. It should only be used for debugging. To enable packet info printing in this sample, the **DOCA_GPUNETIO_SIMPLE_RECEIVE_DEBUG** macro must be set to **1** .

5.4.11.8.2.1 Build Instructions

Build the sample

```
# Ensure DOCA is in the pkgconfig environment variable
cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_simple_receive
meson build
ninja -C build
```

5.4.11.8.2.2 Execution and Testing

This guide assumes a two-machine setup:

- Receiver Machine: Runs the **doca_gpunetio_simple_receive** application.
- Packet Generator Machine: Uses an application like **nping** to send UDP packets.

Packet Generator Machine Example

On the packet generator machine, use **nping** to send 10 UDP packets to the receiver's IP address (assumed to be **192.168.1.1** in this example).

Command:

nping generator

```
$ nping --udp -c 10 -p 2090 192.168.1.1 --data-length 1024 --delay 500ms
```

Output:

```
Starting Nping 0.7.80 ( https://nmap.org/nping ) at 2023-11-20 11:05 UTC
SENT (0.0018s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (0.5018s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (1.0025s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (1.5025s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (2.0032s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (2.5033s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (3.0040s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (3.5040s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (4.0047s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (4.5048s) UDP packet with 1024 bytes to 192.168.1.1:2090

Max rtt: N/A | Min rtt: N/A | Avg rtt: N/A
UDP packets sent: 10 | Rcvd: 0 | Lost: 10 (100.00%)
Nping done: 1 IP address pinged in 5.50 seconds
```

Receiver Machine Example

On the receiver machine, run the sample with the appropriate PCI addresses for the NIC and GPU. This example uses **BLOCK** scope (**-e 2**).

Command:

nping generator

```
# Ensure DOCA is in the LD_LIBRARY_PATH environment variable
$ sudo ./doca_gpunetio_simple_receive -n 9f:00.0 -g 8a:00.0 -e 2
```

Output:

```
nping generator
```

```
[2025-10-27 00:52:32:387590][3382972416][DOCA][INF][doca_log.cpp:633] DOCA version 3.2.0111
[2025-10-27 00:52:32:387627][3382972416][DOCA][INF][gpunetio_simple_receive_main.c:198][main] Starting the sample
[2025-10-27 00:52:32:681807][3382972416][DOCA][INF][gpunetio_simple_receive_main.c:240][main] Sample configuration:
        GPU 8a:00.0
        NIC 9f:00.0
        Shared QP exec scope Block
[2025-10-27 00:52:32:687128][3382972416][DOCA][WRN][engine_model.c:90] adapting queue depth to 128.
[2025-10-27 00:52:32:753758][3382972416][DOCA][WRN][hws_port.c:864] ARGUMENT_256B resource doens't exist, skip
creating NAT64 actions
[2025-10-27 00:52:32:755527][3382972416][DOCA][INF][gpunetio_simple_receive_sample.c:479][create_rxq] Creating
Sample Eth Rxq
[2025-10-27 00:52:32:755713][3382972416][DOCA][INF][gpunetio_simple_receive_sample.c:544][create_rxq] Mapping
receive queue buffer (0x0x7eef8e000000 size 33554432B dmabuf fd 43) with dmabuf mode
[2025-10-27 00:52:32:795823][3382972416][DOCA][INF][gpunetio_simple_receive_sample.c:717][gpunetio_simple_receive]
Launching CUDA kernel to receive packets
[2025-10-27 00:52:32:799403][3382972416][DOCA][INF][gpunetio_simple_receive_sample.c:721][gpunetio_simple_receive]
Waiting for termination

# Type Ctrl+C to kill the sample

[2025-10-27 00:53:35:034046][3382972416][DOCA][INF][gpunetio_simple_receive_sample.c:61][signal_handler] Signal 2
received, preparing to exit!
Exiting from simple receive sample. Total number of received packets: 10
[2025-10-27 00:53:35:034322][3382972416][DOCA][INF][gpunetio_simple_receive_sample.c:395][destroy_rxq] Destroying
Rxq
[2AcC-10-27 00:53:35:061065][3382972416][DOCA][WRN][hws_group_pool.c:85] group_pool has 1 used groups
[2025-10-27 00:53:35:845568][3382972416][DOCA][INF][gpunetio_simple_receive_sample.c:738][gpunetio_simple_receive]
Sample finished successfully
[2025-10-27 00:53:35:845583][3382972416][DOCA][INF][gpunetio_simple_receive_main.c:259][main] Sample finished
successfully
```

5.4.11.8.3 Ethernet Simple Send

This sample implements a simple GPU Ethernet packet generator that constantly sends a flow of raw Ethernet packets. It demonstrates two different implementation "flavors" for sending packets: one using the low-level API and one using the high-level API.

The behavior is controlled by the `-q` command-line option:

- `-q 0` (Low-level API): Disables the shared queue feature and executes using the low-level send functions.
- `-q 1` (High-level API): Enables the shared queue feature. When this is set, the execution scope (thread, warp, or block) can also be chosen using the `-e` option.

5.4.11.8.3.1 Build Instructions

```
DOCA Simple Receive
```

```
# Ensure DOCA is in the pkgconfig environment variable
cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_simple_send
meson build
ninja -C build
```

5.4.11.8.3.2 Execution Example

Assuming a system with a NIC at PCIe address `9f:00.0` and a GPU at `8a:00.0`, the following command launches the sample to send 1024-byte packets (`-s 1024`) using 1024 CUDA threads (`-t 1024`). It enables the high-level API (`-q 1`) with a `BLOCK` execution scope (`-e 2`).

Command:

DOCA Simple Receive

```
# Ensure DOCA is in the LD_LIBRARY_PATH environment variable
$ sudo ./doca_gpunetio_simple_send -n 9f:00.0 -g 8a:00.0 -s 1024 -t 1024 -q 1 -e 2
```

Output:

```
[2025-10-28 04:38:10:548387][961593344][DOCA][INF][doca_log.cpp:645] DOCA version 3.3.0010
[2025-10-28 04:38:10:548420][961593344][DOCA][INF][gpunetio_simple_send_main.c:363][main] Starting the sample
[2025-10-28 04:38:10:839382][961593344][DOCA][INF][gpunetio_simple_send_main.c:433][main] Sample configuration:
        GPU 8a:00.0
        NIC 9f:00.0
        Packet size 1024
        CUDA threads 1024
        CPU Proxy No
        Shared QP Yes
        Shared QP exec scope Block

[2025-10-28 04:38:10:911219][961593344][DOCA][INF][gpunetio_simple_send_sample.c:291][create_txq] Creating Sample
Eth Txq
[2025-10-28 04:38:10:916434][961593344][DOCA][INF][gpunetio_simple_send_sample.c:429][create_txq] Mapping send
queue buffer (0x0x7fd718800000 size 1048576B dmabuf fd 45) with dmabuf mode
[2025-10-28 04:38:10:917479][961593344][DOCA][INF][gpunetio_simple_send_sample.c:580][gpunetio_simple_send]
Launching CUDA kernel to send packets.
[2025-10-28 04:38:10:920972][961593344][DOCA][INF][gpunetio_simple_send_sample.c:584][gpunetio_simple_send] Waiting
for ctrl+c termination

# Type Ctrl+C to kill the sample

[2025-10-28 04:38:22:681160][961593344][DOCA][INF][gpunetio_simple_send_sample.c:67][signal_handler] Signal 2
received, preparing to exit!
[2025-10-28 04:38:22:681176][961593344][DOCA][INF][gpunetio_simple_send_sample.c:590][gpunetio_simple_send] Exiting
from sample
[2025-10-28 04:38:22:681387][961593344][DOCA][INF][gpunetio_simple_send_sample.c:204][destroy_txq] Destroying Txq
[2025-10-28 04:38:22:990964][961593344][DOCA][INF][gpunetio_simple_send_sample.c:612][gpunetio_simple_send] Sample
finished successfully
[2025-10-28 04:38:22:990980][961593344][DOCA][INF][gpunetio_simple_send_main.c:457][main] Sample finished
successfully
```

Performance Verification

To verify that packets are being sent, you can check the traffic throughput on the same machine using the `mlnx_perf` command. (Assuming the NIC at `9f:00.0` has the interface name `ens6f0np0`):

nping generator

```
$ mlnx_perf -i ens6f0np0
      tx_vport_unicast_packets: 21,033,677
        tx_vport_unicast_bytes: 21,538,485,248 Bps   = 172,307.88 Mbps
                tx_packets_phy: 21,033,286
                  tx_bytes_phy: 21,622,256,208 Bps   = 172,978.4 Mbps
                tx_prio0_bytes: 21,617,084,940 Bps   = 172,936.67 Mbps
              tx_prio0_packets: 21,028,292
                          UP 0: 172,936.67           Mbps = 100.00%
                          UP 0: 21,028,292           Tran/sec = 100.00%
```

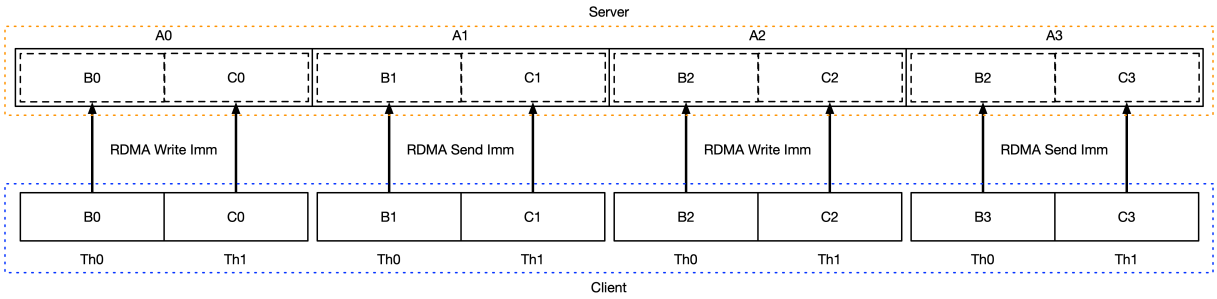
> **Note on Performance**
>
> Throughput will vary between different systems based on factors like the PCIe connection between the GPU and NIC, the NIC model, and the GPU model. This sample can be used to evaluate your system's send performance by varying the packet size and number of CUDA threads.

5.4.11.8.4 RDMA Client Server

This sample exhibits how to use the GPUNetIO RDMA API to receive and send/write with immediate using a single RDMA queue.

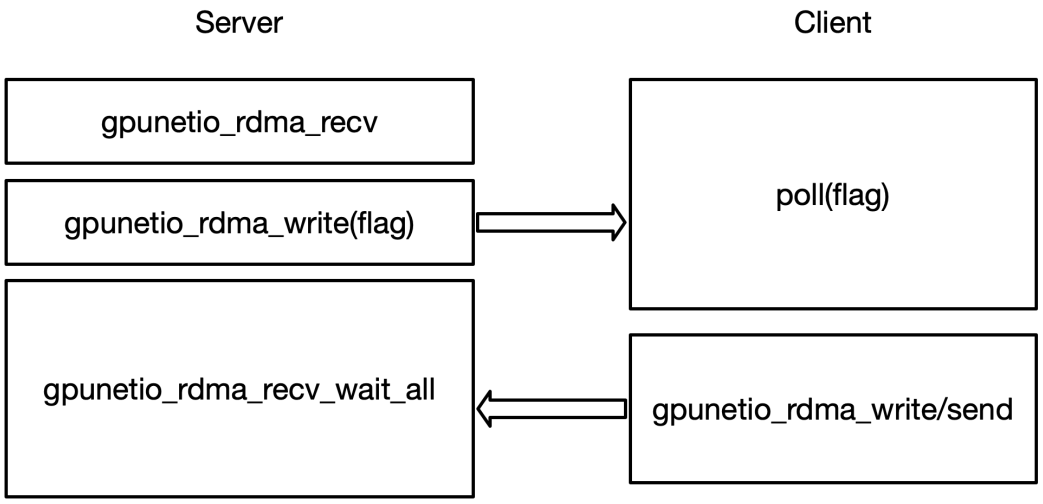
The server has a GPU buffer array A composed by **GPU_BUF_NUM doca_gpu_buf** elements, each 1kB in size. The client has two GPU buffer arrays, B and C, each composed by **GPU_BUF_NUM doca_gpu_buf** elements, each 512B in size.

The goal is for the client to fill a single server buffer of 1kB with two GPU buffers of 512B as illustrated in the following figure:



To show how to use RDMA write and send, even buffers are sent from the client with write immediate, while odd buffers are sent with send immediate. In both cases, the server must pre-post the RDMA receive operations.

For each buffer, the CUDA kernel code repeats the handshake:



Once all buffers are filled, the server double checks that all values are valid. The server output should be as follows:

DOCA RDMA Server side

```
# Ensure DOCA and DPDK are in the LD_LIBRARY_PATH environment variable
$ cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_rdma_client_server_write
$ ./build/doca_gpunetio_rdma_client_server_write -gpu 17:00.0 -d mlx5_0

[14:11:43:000930][1173110][DOCA][INF][gpunetio_rdma_client_server_write_main.c:250][main] Starting the sample
...
```

```
[14:11:43:686610][1173110][DOCA][INF][rdma_common.c:91][oob_connection_server_setup] Listening for incoming
connections
[14:11:45:681523][1173110][DOCA][INF][rdma_common.c:105][oob_connection_server_setup] Client connected at IP:
192.168.2.28 and port: 46274
...
[14:11:45:771807][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:644][rdma_write_server] Before
launching CUDA kernel, buffer array A is:
[14:11:45:771822][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:646][rdma_write_server] Buffer 0
-> offset 0: 1111 | offset 128: 1111
[14:11:45:771837][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:646][rdma_write_server] Buffer 1
-> offset 0: 1111 | offset 128: 1111
[14:11:45:771851][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:646][rdma_write_server] Buffer 2
-> offset 0: 1111 | offset 128: 1111
[14:11:45:771864][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:646][rdma_write_server] Buffer 3
-> offset 0: 1111 | offset 128: 1111
RDMA Recv 2 ops completed with immediate values 0 and 1!
RDMA Recv 2 ops completed with immediate values 1 and 2!
RDMA Recv 2 ops completed with immediate values 2 and 3!
RDMA Recv 2 ops completed with immediate values 3 and 4!
[14:11:45:781561][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:671][rdma_write_server] After
launching CUDA kernel, buffer array A is:
[14:11:45:781574][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:673][rdma_write_server] Buffer 0
-> offset 0: 2222 | offset 128: 3333
[14:11:45:781583][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:673][rdma_write_server] Buffer 1
-> offset 0: 2222 | offset 128: 3333
[14:11:45:781593][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:673][rdma_write_server] Buffer 2
-> offset 0: 2222 | offset 128: 3333
[14:11:45:781602][1173110][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:673][rdma_write_server] Buffer 3
-> offset 0: 2222 | offset 128: 3333
[14:11:45:781640][1173110][DOCA][INF][gpunetio_rdma_client_server_write_main.c:294][main] Sample finished
successfully
```

On the other side, assuming the server is at IP address **192.168.2.28**, the client output should be as follows:

DOCA RDMA Client side

```
# Ensure DOCA and DPDK are in the LD_LIBRARY_PATH environment variable

$ cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_rdma_client_server_write
$ ./build/doca_gpunetio_rdma_client_server_write -gpu 17:00.0 -d mlx5_0 -c 192.168.2.28

[16:08:22:335744][160913][DOCA][INF][gpunetio_rdma_client_server_write_main.c:197][main] Starting the sample
...
[16:08:25:753316][160913][DOCA][INF][rdma_common.c:147][oob_connection_client_setup] Connected with server
successfully
......
Client waiting on flag 7f6596735000 for server to post RDMA Recvs
Thread 0 post rdma write imm 0
Thread 1 post rdma write imm 0
Client waiting on flag 7f6596735001 for server to post RDMA Recvs
Thread 0 post rdma send imm 1
Thread 1 post rdma send imm 1
Client waiting on flag 7f6596735002 for server to post RDMA Recvs
Thread 0 post rdma write imm 2
Thread 1 post rdma write imm 2
Client waiting on flag 7f6596735003 for server to post RDMA Recvs
Thread 0 post rdma send imm 3
Thread 1 post rdma send imm 3
[16:08:25:853454][160913][DOCA][INF][gpunetio_rdma_client_server_write_main.c:241][main] Sample finished
successfully
```

With RDMA, the network device must be specified by name (e.g., **mlx5_0**) instead of the PCIe address (as is the case for Ethernet).

It is also possible to enable the RDMA CM mode, establishing two connections with the same RDMA GPU handler. An example on the client side:

DOCA RDMA Client side with CM

```
# Ensure DOCA and DPDK are in the LD_LIBRARY_PATH environment variable

$ cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_rdma_client_server_write
$ ./build/samples/doca_gpunetio_rdma_client_server_write -d mlx5_0 -gpu 17:00.0 -gid 3 -c 10.137.189.28 -cm --
server-addr-type ipv4 --server-addr 192.168.2.28

[11:30:34:489781][3853018][DOCA][INF][gpunetio_rdma_client_server_write_main.c:461][main] Starting the sample
...
[11:30:35:038828][3853018][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:950][rdma_write_client] Client is
waiting for a connection establishment
```

```
[11:30:35:082039][3853018][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:963][rdma_write_client] Client -
Connection 1 is established
...
[11:30:35:095282][3853018][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:1006][rdma_write_client]
Establishing connection 2..
[11:30:35:097521][3853018][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:1016][rdma_write_client] Client is
waiting for a connection establishment
[11:30:35:102718][3853018][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:1029][rdma_write_client] Client -
Connection 2 is established
[11:30:35:102783][3853018][DOCA][INF][gpunetio_rdma_client_server_write_sample.c:1046][rdma_write_client] Client,
terminate kernels
Client waiting on flag 7f16067b5000 for server to post RDMA Recvs
Thread 0 post rdma write imm 0
Thread 1 post rdma write imm 1
Client waiting on flag 7f16067b5001 for server to post RDMA Recvs
Thread 0 post rdma send imm 1
Thread 1 post rdma send imm 2
Client waiting on flag 7f16067b5002 for server to post RDMA Recvs
Thread 0 post rdma write imm 2
Thread 1 post rdma write imm 3
Client waiting on flag 7f16067b5003 for server to post RDMA Recvs
Thread 0 post rdma send imm 3
Thread 1 post rdma send imm 4
Client posted and completed 4 RDMA commits on connection 0. Waiting on the exit flag.
Client waiting on flag 7f16067b5000 for server to post RDMA Recvs
Thread 0 post rdma write imm 0
Thread 1 post rdma write imm 1
Client waiting on flag 7f16067b5001 for server to post RDMA Recvs
Thread 0 post rdma send imm 1
Thread 1 post rdma send imm 2
Client waiting on flag 7f16067b5002 for server to post RDMA Recvs
Thread 0 post rdma write imm 2
Thread 1 post rdma write imm 3
Client waiting on flag 7f16067b5003 for server to post RDMA Recvs
Thread 0 post rdma send imm 3
Thread 1 post rdma send imm 4
Client posted and completed 4 RDMA commits on connection 1. Waiting on the exit flag.
[11:30:35:122448][3853018][DOCA][INF][gpunetio_rdma_client_server_write_main.c:512][main] Sample finished
successfully
```

In case of RDMA CM, the command option **-cm** must be specified on the server side.

Printing from a CUDA kernel is not recommended for performance. It may make sense for debugging purposes and for simple samples like this one.

5.4.11.8.5 Verbs Samples

The `doca_gpunetio_verbs_*` examples demonstrate how to use the GPUNetIO Verbs API in various scenarios. These samples require a client-server setup, with the client needing the `-c <server IP>` parameter.

The following parameters are supported by all Verbs samples:

- `-n`: Network card handler type (0: AUTO, 1: CPU Proxy, 2: GPU DB). Default is 0 (AUTO).
- `-e`: Execution mode for shared QP (0: per-thread, 1: per-warp). Default is 0 (per-thread).
- `-d`: Network card device name.
- `-g`: GPU device PCIe address.
- `-gid`: GID index for DOCA RDMA (optional).
- `-i`: Number of iterations (optional).
- `-t`: Number of CUDA threads (optional).

Please read the specific sample section below to check for extra parameters.

For the examples in this guide, it is assumed that the GPU PCIe address is `8A:00.0` and the network card PCIe address is specified accordingly.

Running with RoCE

If the samples are running on a RoCE connection (e.g. the ConnectX/BlueField is set in Ethernet mode instead of InfiniBand mode) you may get this (or similar) error: **FW failed to modify object, status=BAD_PARAM_ERR (0x3), syndrome=0x1f3b5d** .

To fix it, please remove the **doca_verbs_ah_attr_set_dlid** function from the **connect_verbs_qp** function in the **samples/doca_gpunetio/verbs_common.c** file.

Performance Tuning: MTU Optimization

For optimal bandwidth performance, the default MTU in **verbs_common.cpp** is set to 4KB:

```
doca_verbs_qp_attr_set_path_mtu(verbs_qp_attr, DOCA_MTU_SIZE_4K_BYTES);
```

To accommodate this, you must ensure your network interface is configured to support an MTU of at least 4200 bytes. You can configure this using the following command:

```
$ ifconfig <interface_name> mtu 4200
```

If your network infrastructure does not support 4KB MTUs, you must modify the sample code to use a 1KB MTU (**DOCA_MTU_SIZE_1K_BYTES**).

5.4.11.8.5.1 Bandwidth Samples

The samples ending with `_bw` measure the bandwidth of specific GPUNetIO Verbs API functions. In these samples, the client prepares and sends data from a CUDA kernel, while the server waits on the CPU to receive the data, validate it upon receiving a Ctrl+C signal, and reports the outcome.

Key Characteristics

- The client outputs the MB/s achieved for preparing and sending messages of various sizes.
- The server outputs a message indicating the execution outcome.
- All bandwidth samples support the previously listed command-line parameters.

Simplifying QP/CQ/UAR Creation

To simplify the integration of DOCA RDMA Verbs with GPUNetIO, high-level functions like `doca_gpu_verbs_create_qp_hl()` and `doca_gpu_verbs_create_qp_group_hl()` are provided in `samples/doca_gpunetio/verbs_high_level.cpp`. These functions encapsulate the necessary steps for creating QP/CQ/UAR, making it easier for developers to combine DOCA RDMA Verbs and GPUNetIO in their applications.

doca_gpunetio_verbs_write_bw

The `doca_gpunetio_verbs_write_bw` test measures the bandwidth of RDMA Write operations using the GPUNetIO Verbs API. It launches a CUDA kernel with 1 block and 512 threads by default, where all threads post RDMA Write WQEs in different positions and the last thread submits them. The test then polls the CQE corresponding to the last WQE to ensure all previous WQEs have been executed correctly.

Example command lines:

- Server:

```
doca_gpunetio_verbs_write_bw -g 8A:00.0 -d mlx5_0
```

- Client (additional command-line options can be added):

```
doca_gpunetio_verbs_write_bw -g 8A:00.0 -d mlx5_0 -c 192.168.1.63
```

doca_gpunetio_verbs_put_bw

The `doca_gpunetio_verbs_put_bw` test measures the bandwidth of RDMA Write operations (referred to as "Put") using the GPUNetIO Verbs API with the shared QP feature. It launches a CUDA kernel with 2 blocks, each containing 256 threads. The test can also measure individual function latencies by setting the `KERNEL_DEBUG_TIMES` macro to 1.

Example command lines:

- Server:

```
doca_gpunetio_verbs_put_bw -g 8A:00.0 -d mlx5_0
```

- Client (additional command-line options can be added):

```
doca_gpunetio_verbs_put_bw -g 8A:00.0 -d mlx5_0 -c 192.168.1.63
```

doca_gpunetio_verbs_put_signal_bw

The `doca_gpunetio_verbs_put_signal_bw` test measures the bandwidth of Put + Signal operations (RDMA Write + RDMA Atomic with shared QP) using the GPUNetIO Verbs API. It launches a CUDA kernel with 2 blocks, each containing 256 threads.

Example command lines:

- Server:

```
doca_gpunetio_verbs_put_signal_bw -g 8A:00.0 -d mlx5_0
```

- Client (additional command-line options can be added):

```
doca_gpunetio_verbs_put_signal_bw -g 8A:00.0 -d mlx5_0 -c 192.168.1.63
```

Note on CUDA Kernel Loop

The sample works correctly, but there is a minor mistake in the CUDA kernel loop. The corrected loop should be:

```
do {
    final_val = doca_gpu_dev_verbs_atomic_read<uint64_t,
DOCA_GPUNETIO_VERBS_RESOURCE_SHARING_MODE_GPU>(&prev_flag_buf[tidx]);
    doca_gpu_dev_verbs_fence_acquire<DOCA_GPUNETIO_VERBS_SYNC_SCOPE_SYS>();
} while((final_val != (iter_thread - 1)) && (final_val != ((iter_thread * 2) - 1)));
```

doca_gpunetio_verbs_put_counter_bw

The `doca_gpunetio_verbs_put_counter_bw` test measures the bandwidth of Put + Counter operations (RDMA Write + Wait WQE + RDMA Atomic with shared QP) using the GPUNetIO Verbs API. It launches a CUDA kernel with 2 blocks, each containing 256 threads.

Example command lines:

- Server:

```
doca_gpunetio_verbs_put_counter_bw -g 8A:00.0 -d mlx5_0
```

- Client (additional command-line options can be added):

```
doca_gpunetio_verbs_put_counter_bw -g 8A:00.0 -d mlx5_0 -c 192.168.1.63
```

Note on CUDA Kernel Loop

The sample works correctly, but there is a minor mistake in the CUDA kernel loop. The corrected loop should be:

```
do {
    final_val = doca_gpu_dev_verbs_atomic_read<uint64_t,
DOCA_GPUNETIO_VERBS_RESOURCE_SHARING_MODE_GPU>(&prev_flag_buf[tidx]);
    doca_gpu_dev_verbs_fence_acquire<DOCA_GPUNETIO_VERBS_SYNC_SCOPE_SYS>();
} while((final_val != (iter_thread - 1)) && (final_val != ((iter_thread * 2) - 1)));
```

doca_gpunetio_verbs_twosided_bw

The `doca_gpunetio_verbs_twosided_bw` test measures the bandwidth of client-server data exchange via Send/Recv operations using the GPUNetIO Verbs API with the shared QP feature. It launches a CUDA kernel with 2 blocks, each containing 256 threads.

Example command lines:

- Server:

```
doca_gpunetio_verbs_twosided_bw -g 8A:00.0 -d mlx5_0
```

- Client (additional command-line options can be added):

```
doca_gpunetio_verbs_twosided_bw -g 8A:00.0 -d mlx5_0 -c 192.168.1.63
```

doca_gpunetio_verbs_get_bw

This sample application measures the bandwidth of one-sided RDMA Read ("Get") operations. It uses the GPUNetIO Verbs API with the shared QP feature.

The application launches a CUDA kernel configured with 2 blocks, each containing 256 threads.

Example usage:

- Server:

```
doca_gpunetio_verbs_get_bw -g 8A:00.0 -d mlx5_0
```

- Client (connects to the server's IP, e.g., **192.168.1.63**):

```
doca_gpunetio_verbs_get_bw -g 8A:00.0 -d mlx5_0 -c 192.168.1.63
```

Additional CLI options can be added.

5.4.11.8.5.2 Latency Samples

The samples ending with `_lat` measure the latency of specific GPUNetIO Verbs API functions by performing a ping-pong exchange between client and server. These tests launch a CUDA kernel with a single CUDA thread and do not support the `-e` and `-t` command-line options.

Both client and server output the round-trip time (RTT) latency (half and full) in microseconds for preparing and exchanging messages of different sizes. The server also outputs a message indicating the execution outcome.

doca_gpunetio_verbs_write_lat

The `doca_gpunetio_verbs_write_lat` test measures the latency of RDMA Write operations using the GPUNetIO Verbs API without the shared QP feature. It is similar to perftest.

This sample support the reliable doorbell feature with command-line option **`-r <0|1>`** .

Example command lines:

- Server:

```
doca_gpunetio_verbs_write_lat -g 8A:00.0 -d mlx5_0
```

- Client (additional command-line options can be added):

```
doca_gpunetio_verbs_write_lat -g 8A:00.0 -d mlx5_0 -c 192.168.1.63
```

doca_gpunetio_verbs_put_signal_lat

The `doca_gpunetio_verbs_put_signal_lat` test measures the latency of Put and Signal operations (RDMA Write + RDMA Atomic with shared QP) using the GPUNetIO Verbs API. It uses high-level API functions from `doca_gpunetio_dev_verbs_onesided.cuh` and launches a CUDA kernel with a single CUDA thread.

Example command lines:

- Server:

```
doca_gpunetio_verbs_put_signal_lat -g 8A:00.0 -d mlx5_0
```

- Client (additional command-line options can be added):

```
doca_gpunetio_verbs_put_signal_lat -g 8A:00.0 -d mlx5_0 -c 192.168.1.63
```

doca_gpunetio_verbs_put_counter_lat

The `doca_gpunetio_verbs_put_counter_lat` test measures the latency of Put + Counter operations (RDMA Write + Wait WQE + RDMA Atomic with shared QP) using the GPUNetIO Verbs API. It utilizes high-level API functions from `doca_gpunetio_dev_verbs_counter.cuh` and the Core Direct counter feature, even with a single CUDA thread.

This sample support the reliable doorbell feature with command-line option `-r <0|1>` .

Example command lines:

- Server:

```
doca_gpunetio_verbs_put_counter_lat -g 8A:00.0 -d mlx5_0
```

- Client: (additional command-line options can be added):

```
doca_gpunetio_verbs_put_counter_lat -g 8A:00.0 -d mlx5_0 -c 192.168.1.63
```

5.4.11.8.6 GPU DMA Copy

This sample exhibits how to use the DOCA DMA and DOCA GPUNetIO libraries to DMA copy a memory buffer from the CPU to the GPU (with DOCA DMA CPU functions) and from the GPU to the CPU (with DOCA GPUNetIO DMA device functions) from a CUDA kernel. This sample requires a DPU as it uses the DMA engine on it.

DOCA RDMA Client side

```
$ cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_dma_memcpy

# Build the sample and then execute

$ ./build/doca_gpunetio_dma_memcpy -g 17:00.0 -n ca:00.0
[15:44:04:189462][862197][DOCA][INF][gpunetio_dma_memcpy_main.c:164][main] Starting the sample
EAL: Detected CPU lcores: 64
EAL: Detected NUMA nodes: 2
EAL: Detected shared linkage of DPDK
EAL: Selected IOVA mode 'VA'
EAL: No free 2048 kB hugepages reported on node 0
EAL: No free 2048 kB hugepages reported on node 1
EAL: VFIO support initialized
TELEMETRY: No legacy callbacks, legacy socket not created
EAL: Probe PCI driver: gpu_cuda (10de:2331) device: 0000:17:00.0 (socket 0)
[15:44:04:857251][862197][DOCA][INF][gpunetio_dma_memcpy_sample.c:211][init_sample_mem_objs] The CPU source buffer
value to be copied to GPU memory: This is a sample piece of text from CPU
[15:44:04:857359][862197][DOCA][WRN][doca_mmap.cpp:1743][doca_mmap_set_memrange] Mmap 0x55aec6206140: Memory range
isn't cache-line aligned - addr=0x55aec52ceb10. For best performance align address to 64B
[15:44:04:858839][862197][DOCA][INF][gpunetio_dma_memcpy_sample.c:158][init_sample_mem_objs] The GPU source buffer
value to be copied to CPU memory: This is a sample piece of text from GPU
[15:44:04:921702][862197][DOCA][INF][gpunetio_dma_memcpy_sample.c:570][submit_dma_memcpy_task] Success, DMA memcpy
job done successfully
CUDA KERNEL INFO: The GPU destination buffer value after the memcpy: This is a sample piece of text from CPU
CPU received message from GPU: This is a sample piece of text from GPU
[15:44:04:930087][862197][DOCA][INF][gpunetio_dma_memcpy_sample.c:364][gpu_dma_cleanup] Cleanup DMA ctx with GPU
data path
[15:44:04:932658][862197][DOCA][INF][gpunetio_dma_memcpy_sample.c:404][gpu_dma_cleanup] Cleanup DMA ctx with CPU
data path
[15:44:04:954156][862197][DOCA][INF][gpunetio_dma_memcpy_main.c:197][main] Sample finished successfully
```

5.4.12 DOCA App Shield

This guide provides instructions on using the DOCA App Shield API.

5.4.12.1 Introduction

> ⚠ The quality status of DOCA libraries is listed here.

DOCA App Shield API offers a solution for strong intrusion detection capabilities using the DPU services to collect and analyze data from the host's (or a VM on the host) memory in real time. This solution provides intrusion detection and forensics investigation in a way that is:

- Robust against attacks on a host machine
- Able to detect a wide range of attacks (including zero-day attacks)
- Least disruptive to the execution of host application (where current detection solutions hinder the performance of host applications)
- Transparent to the host, such that the host does not need to install anything (other than providing some files obtained from the **doca_apsh_config.py** tool)

App Shield uses a DMA device to access the host's memory and analyze it.

The App Shield API provides multiple functions that help with gathering data extracted from system's memory (e.g., processes list, modules list, connections). This data helps with detecting attacks on critical services or processes in a system (e.g., services that enforce integrity or privacy of the execution of different applications).

5.4.12.2 Prerequisites

1. Configure the NVIDIA® BlueField® networking platform's (DPU or SuperNIC) firmware.
   a. On BlueField, configure the PF base address register and NVMe emulation. Run:

   ```
   dpu> mlxconfig -d /dev/mst/mt41686_pciconf0 s PF_BAR2_SIZE=2 PF_BAR2_ENABLE=1
   ```

   If working with VFs, configure NVME emulation, SR-IOV, and number of VFs. Run:

   ```
   dpu> mlxconfig -d /dev/mst/mt41686_pciconf0 s NVME_EMULATION_ENABLE=1 SRIOV_EN=1 NUM_OF_VFS=<vf-number>
   ```

   b. Perform graceful shutdown and a cold boot from the host.

   > These configurations can be checked using the following command:
   >
   > ```
   > dpu> mlxconfig -d /dev/mst/mt41686_pciconf0 q | grep -E "NVME|BAR|SRIOV|NUM_OF_VFS"
   > ```

2. Download target system (host/VM) symbols.
   - For Ubuntu:

   ```
   host> sudo tee /etc/apt/sources.list.d/ddebs.list << EOF
   deb http://ddebs.ubuntu.com/ $(lsb_release -cs) main restricted universe multiverse
   deb http://ddebs.ubuntu.com/ $(lsb_release -cs)-updates main restricted universe multiverse
   deb http://ddebs.ubuntu.com/ $(lsb_release -cs)-proposed main restricted universe multiverse
   EOF
   host> sudo apt install ubuntu-dbgsym-keyring
   host> sudo apt-get update
   host> sudo apt-get install linux-image-$(uname -r)-dbgsym
   ```

   - For CentOS:

   ```
   host> yum install --enablerepo=base-debuginfo kernel-devel-$(uname -r) kernel-debuginfo-$(uname -r) kernel-debuginfo-common-$(uname -m)-$(uname -r)
   ```

   - No action is needed for Windows
3. Perform IOMMU passthrough. This stage is only necessary if IOMMU is not enabled by default (e.g., when the host is using an AMD CPU).

   > Skip this step if you are not sure whether it is needed. Return to it only if DMA fails with a message similar to the following in **dmesg**:

```
host> dmesg
[ 3839.822897] mlx5_core 0000:81:00.0: AMD-Vi: Event logged [IO_PAGE_FAULT domain=0x0047
address=0x2a0aff8 flags=0x0000]
```

a. Locate your OS's `grub` file (most likely **/boot/grub/grub.conf**, **/boot/grub2/grub.cfg**, or **/etc/default/grub**) and open it for editing. Run:

```
host> vim /etc/default/grub
```

b. Search for the line defining **GRUB_CMDLINE_LINUX_DEFAULT** and add the argument **iommu=pt**. For example:

```
GRUB_CMDLINE_LINUX_DEFAULT="iommu=pt <intel/amd>_iommu=on"
```

c. Run:

⚠ Prior to performing a power cycle, make sure to do a graceful shutdown.

- For Ubuntu:

```
host> sudo update-grub
```

- For CentOS:

```
host> grub2-mkconfig -o /boot/grub2/grub.cfg
```

4. BlueField currently supports up to 4-level virtual memory layers. Therefore, it is necessary to check if the target system uses a 5-level virtual memory layer and to adjust the configuration accordingly:
   a. Check the virtual memory layer:

```
grep la57 /proc/cpuinfo
```

   Verify the **la57** flag in the output. If something like this appears:

```
flags : ... la57 ...
```

   This indicates that the target system is using a 5-level virtual memory layer and that 5-level paging must be deactivated.
   b. To deactivate 5-level paging, modify the kernel parameters using GRUB. Add the following line to your GRUB configuration:

```
GRUB_CMDLINE_LINUX_DEFAULT="no5lvl"
```

   c. Update GRUB configuration:

```
update-grub
```

   d. Reboot the system to apply the changes.
5. Prepare target:
   a. Install DOCA on the target system.

b. Create the ZIP and JSON files. Run:

```
target-system> cd /opt/mellanox/doca/tools/
target-system> python3 doca_apsh_config.py --pid <pid-of-process-to-monitor> --os <windows/linux>
--path <path to dwarf2json executable  or pdbparse-to-json.py>
target-system> cp /opt/mellanox/doca/tools/*.* <shared-folder-with-baremetal>
dpu> scp <shared-folder-with-baremetal>/* <path-to-app-shield-binary>
```

If the target system does not have DOCA installed, the script can be copied from BlueField.
The required **dwaf2json** and **pdbparse-to-json.py** are not provided with DOCA.

> If the kernel and process **.exe** have not changed, there is no need to redo this step.

5.4.12.3 Dependencies

The library requires firmware version 24.32.1010 or higher.

5.4.12.4 API

For the library API reference, refer to the DOCA APSH API documentation in the NVIDIA DOCA Library APIs.

> The pkg-config (***.pc** file) for the APSH library is **doca-apsh** .

The following subsections provide more details about the library API.

5.4.12.4.1 doca_apsh_dma_dev_set

To attach a DOCA DMA device to App Shield, calling this function is mandatory and must be done before calling **doca_apsh_start** .

```
doca_apsh_dma_dev_set(doca_apsh_ctx, doca_dev)
```

- **doca_apsh_ctx [in]** – App Shield opaque context struct
- **doca_dev [in]** – struct for DOCA Device with DMA capabilities

5.4.12.4.2 Capabilities Per System

For each initialized system, App Shield retrieves an array of the requested object according to the getter's name:

Getter Function Name	Functions Information	Functions Signature	Return Type
Get modules	Returns an array with information about the system modules (drivers) loaded into the kernel of the OS.	`doca_error_t doca_apsh_modules_get(struct doca_apsh_system *system, struct doca_apsh_module ***modules, int *modules_size);`	• Array of **struct doca_apsh_module** • **int** – size of the returned array • **doca_error** status
Get processes	Returns an array with information about each process running on the system.	`doca_error_t doca_apsh_processes_get(struct doca_apsh_system *system, struct doca_apsh_process ***processes, int *processes_size);`	• Array of **struct doca_apsh_process** • **int** – size of the returned array • **doca_error** status
Get library	For a specified process, this function returns an array with information about each library loaded into this process.	`doca_error_t doca_apsh_libs_get(struct doca_apsh_process *process, struct doca_apsh_lib ***libs, int *libs_size);`	• Array of **struct doca_apsh_lib** • **int** – size of the returned array • **doca_error** status
Get threads	For a specified process, this function returns an array with information about each thread running within this process.	`doca_error_t doca_apsh_threads_get(struct doca_apsh_process *process, struct doca_apsh_thread ***threads, int *threads_size);`	• Array of **struct doca_apsh_thread** • **int** – size of the returned array • **doca_error** status

Getter Function Name	Functions Information	Functions Signature	Return Type
Get virtual memory areas/ virtual address description	For a specified process, this function returns an array with information about each virtual memory area within this process.	`doca_error_t doca_apsh_vads_get(struct doca_apsh_process *process, struct doca_apsh_vad ***vads, int *vads_size);`	• Array of **struct doca_apsh_vma** • **int** – size of the returned array • **doca_error** status
Get privileges	For a specified process, this function returns an array with information about each possible privilege for this process, as described here. ⚠ Available on a Windows host only.	`doca_error_t doca_apsh_privileges_get(struct doca_apsh_process *process, struct doca_apsh_privilege ***privileges, int *privileges_size);`	• Array of **struct doca_apsh_privilege** • **int** – size of the returned array • **doca_error** status
Get environment variables	For a specified process, this function returns an array with information about each environment variable within this process. ⚠ Available on a Windows host only.	`doca_error_t doca_apsh_envars_get(struct doca_apsh_process *process, struct doca_apsh_envar ***envars, int *envars_size);`	• Array of **struct doca_apsh_envar** • **int** – size of the returned array • **doca_error** status
Get handles	For a specified process, this function returns an array with information about each handle this process holds. ⚠ Available on a Windows host only.	`doca_error_t doca_apsh_handles_get(struct doca_apsh_process *process, struct doca_apsh_handle ***handles, int *handles_size);`	• Array of **struct doca_apsh_handle** • **int** – size of the returned array • **doca_error** status

Getter Function Name	Functions Information	Functions Signature	Return Type
Get LDR modules	For a specified process, this function returns an array with information about each loaded module within this process. ⚠ Available on a Windows host only.	`doca_error_t doca_apsh_ldrmodules_get(struct doca_apsh_process *process, struct doca_apsh_ldrmodule ***ldrmodules, int *ldrmodules_size);`	• Array of **struct doca_apsh_ldrmodule** • **int** – size of the returned array • **doca_error** status
Process attestation	For a specified process, this function attests the memory pages of the process according to a precomputed golden hash file given as an input. ⚠ Single-threaded processes are supported at beta level.	`doca_error_t doca_apsh_attestation_get(struct doca_apsh_process *process, const char *exec_hash_map_path, struct doca_apsh_attestation ***attestation, int *attestation_size);`	• Array of **struct doca_apsh_attestation** • **int** – size of the returned array • **doca_error** status
Attestation refresh	Refreshes a single attestation handler of a process with a new snapshot.	`doca_error_t doca_apsh_attst_refresh(struct doca_apsh_attestation ***attestation, int *attestation_size);`	• Array of **struct doca_apsh_attestation** • **int** – size of the returned array • **doca_error** status

<table>
<tr><th>Getter Function Name</th><th>Functions Information</th><th>Functions Signature</th><th>Return Type</th></tr>
<tr>
<td>Get NetScan</td>
<td>This function scans the system's physical memory (Windows) or kernel memory space (Linux) and returns an array with information about each socket that resides in the memory.

⚠ Only available on hosts with either Linux OS or one of the following Windows 10 OS builds:
<table>
<tr><th>Arch</th><th>Build No.</th></tr>
<tr><td>x86</td><td>10240</td></tr>
<tr><td></td><td>10586</td></tr>
<tr><td></td><td>14393</td></tr>
<tr><td></td><td>15063</td></tr>
<tr><td></td><td>17134</td></tr>
<tr><td></td><td>19041</td></tr>
<tr><td>x64</td><td>15063</td></tr>
<tr><td></td><td>16299</td></tr>
<tr><td></td><td>17134</td></tr>
<tr><td></td><td>17763</td></tr>
<tr><td></td><td>18362</td></tr>
<tr><td></td><td>18363</td></tr>
<tr><td></td><td>19041</td></tr>
</table>
⚠ This feature is currently supported at beta level.</td>
<td><pre>doca_error_t doca_apsh_netscan_get(struct doca_apsh_system *system, struct doca_apsh_netscan ***connections, int *connections_size);</pre></td>
<td>• Array of struct doca_apsh_netscan
• int – size of the returned array
• doca_error status</td>
</tr>
</table>

Getter Function Name	Functions Information	Functions Signature	Return Type
Get process parameters	For a specified process, this function returns a struct object (not an array) with information about the process' parameters (ones not included in the "get processes" capability). ⚠ Available on a Windows host only. ⚠ This feature is currently supported at beta level.	`doca_error_t doca_apsh_process_parameters_get(struct doca_apsh_process *process, struct doca_apsh_process_parameters **process_parameters);`	• An object of **struct doca_apsh_process_parameters** • **doca_error** status
Get security identifier (SID)	For a specified process, this function returns an array with information about each SID (security identifier) included in the process's security context. ⚠ Available on a Windows host only.	`doca_error_t doca_apsh_sids_get(struct doca_apsh_process *process, struct doca_apsh_sid ***sids, int *sids_size);`	• Array of **struct doca_apsh_sid** • **int** – size of the returned array • **doca_error** status
Perform Yara scan	For a specified process, this function returns an array with information about each Yara rule match found in the process' memory. ⚠ Available on a Windows host and Ubuntu 22.04 DPU.	`doca_error_t doca_apsh_yara_get(struct doca_apsh_process *process, enum doca_apsh_yara_rule *yara_rules_arr, uint32_t yara_rules_arr_size, uint64_t scan_type, struct doca_apsh_yara ***yara_matches, int *yara_matches_size);` ⚠ To get a better understanding of the arguments, refer to documentation in **doca_apsh.h**.	• Array of **struct doca_apsh_yara** • **int** – size of the returned array • **doca_error** status
Get containers	Returns an array with information about each container running on the system. ⚠ Available on a Linux host only. ⚠ Only available for containers on the following runtimes: • runc • containerd	`doca_error_t doca_apsh_containers_get(struct doca_apsh_system *system, struct doca_apsh_container ***containers, int *containers_size);`	• Array of **struct doca_apsh_container** • **int** – size of the returned array • **doca_error** status

Getter Function Name	Functions Information	Functions Signature	Return Type
Get container's processes	For a specified container, this function returns an array with information about each process running within this container. ⚠ Available on a Linux host only. ⚠ Only available for containers on the following runtimes: • runc • containerd	`doca_error_t doca_apsh_container_processes_get(struct doca_apsh_container *container, struct doca_apsh_process ***processes, int *processes_size);`	• Array of **struct doca_apsh_process** • **int** – size of the returned array • **doca_error** status

<table>
<tr><th>Getter Function Name</th><th>Functions Information</th><th>Functions Signature</th><th>Return Type</th></tr>
<tr>
<td>Get Process NetScan</td>
<td>This function scans the system's physical memory (Windows) or kernel memory space (Linux) and returns an array with information about each socket that resides in the memory.

⚠ Only available on hosts with either Linux OS or one of the following Windows 10 OS builds:
<table>
<tr><th>Arch</th><th>Build No.</th></tr>
<tr><td>x86</td><td>10240</td></tr>
<tr><td></td><td>10586</td></tr>
<tr><td></td><td>14393</td></tr>
<tr><td></td><td>15063</td></tr>
<tr><td></td><td>17134</td></tr>
<tr><td></td><td>19041</td></tr>
<tr><td>x64</td><td>15063</td></tr>
<tr><td></td><td>16299</td></tr>
<tr><td></td><td>17134</td></tr>
<tr><td></td><td>17763</td></tr>
<tr><td></td><td>18362</td></tr>
<tr><td></td><td>18363</td></tr>
<tr><td></td><td>19041</td></tr>
</table>
⚠ This feature is currently supported at beta level.</td>
<td><pre>doca_error_t
doca_apsh_process_netscan_get(struct
doca_apsh_process *process, struct
doca_apsh_netscan ***connections, int
 *connections_size);</pre></td>
<td>• Array of struct doca_apsh_netscan
• int – size of the returned array
• doca_error status</td>
</tr>
<tr>
<td>Get interfaces</td>
<td>Returns an array with information about each interface running on the system.

⚠ Available on a Linux host only.</td>
<td><pre>doca_error_t
doca_apsh_interfaces_get(struct
doca_apsh_system *system,
struct doca_apsh_interface ***interfaces,
int *interfaces_size);</pre></td>
<td>• Array of struct doca_apsh_interface
• int – size of the returned array
• doca_error status</td>
</tr>
</table>

The following attribute getters return a specific attribute of an object, obtained from the array returned from the getter functions listed above, depending on the requested attribute:

```
doca_apsh_process_info_get(struct doca_apsh_proccess *process, enum doca_apsh_process_attr attr);
doca_apsh_module_info_get(struct doca_apsh_module *module, enum doca_apsh_module_attr attr);
doca_apsh_lib_info_get(struct doca_apsh_lib *lib, enum doca_apsh_lib_attr attr);
doca_apsh_thread_info_get(struct doca_apsh_thread *thread, enum doca_apsh_lib_attr attr);
doca_apsh_vad_info_get(struct doca_apsh_vad *vad, enum doca_apsh_vad_attr attr);
doca_apsh_privilege_info_get(struct doca_apsh_privilege *privilege, enum doca_apsh_privilege_attr attr);
doca_apsh_envar_info_get(struct doca_apsh_envar *envar, enum doca_apsh_envar_attr attr);
doca_apsh_handle_info_get(struct doca_apsh_handle *handle, enum doca_apsh_handle_attr attr);
doca_apsh_ldrmodule_info_get(struct doca_apsh_ldrmodule *ldrmodule, enum doca_apsh_ldrmodule_attr attr);
doca_apsh_attst_info_get(struct doca_apsh_attestation *attestation, enum doca_apsh_attestation_attr attr);
doca_apsh_netscan_info_get(struct doca_apsh_netscan *connection, enum doca_apsh_netscan_attr attr)
doca_apsh_process_parameters_info_get(struct doca_apsh_process_parameters *process_parameters, enum
 doca_apsh_process_parameters_attr attr);
doca_apsh_sid_info_get(struct doca_apsh_sid *sid, enum doca_apsh_sid_attr attr);
doca_apsh_yara_info_get(struct doca_apsh_yara *yara, enum doca_apsh_yara_attr attr);
doca_apsh_container_info_get(struct doca_apsh_container *container, enum doca_apsh_container_attr attr);
```

The return type of the attribute getter can be found in **doca_apsh_attr.h**.

Usage example:

```
const uint pid = doca_apsh_process_info_get(processes[i], DOCA_APSH_PROCESS_PID);
const char *proc_name = doca_apsh_process_info_get(processes[i], DOCA_APSH_PROCESS_COMM);
```

5.4.12.5 App Shield Initialization and Teardown

To use App Shield, users must initialize and configure two main structs. This section presents these structs and explains how to interact with them.

5.4.12.5.1 doca_apsh_ctx

doca_apsh_ctx is the basic struct used by App Shield which defines the DMA device used to perform the memory forensics techniques required to run App Shield.

> ⚠ The same **doca_apsh_ctx** struct may be used to run multiple App Shield instances over different systems (e.g., two different VMs on the host).

1. To acquire an instance of the **doca_apsh_ctx** struct, use the following function:

   ```
   struct doca_apsh_ctx *doca_apsh_create(void);
   ```

2. To configure the **doca_apsh_ctx** instance with DMA device to use:

   ```
   doca_error_t doca_apsh_dma_dev_set(struct doca_apsh_ctx *ctx, struct doca_dev *dma_dev);
   ```

3. To start the **doca_apsh_ctx** instance, call the following function:

   ```
   doca_error_t doca_apsh_start(struct doca_apsh_ctx *ctx);
   ```

4. To destroy the **doca_apsh_ctx** instance when it is no longer needed, call:

   ```
   void doca_apsh_destroy(struct doca_apsh_ctx *ctx);
   ```

5.4.12.5.2 doca_apsh_system

The **doca_apsh_system** struct is built on the **doca_apsh_ctx** instance. This struct is created per system running App Shield. **doca_apsh_system** defines multiple attributes used by App Shield to perform memory analysis over the specific system successfully.

1. To acquire an instance of the **doca_apsh_system** struct, use the following function:

```
const uint pid = doca_apsh_process_info_get(processes[i], DOCA_APSH_PROCESS_PID);
const char *proc_name = doca_apsh_process_info_get(processes[i], DOCA_APSH_PROCESS_COMM);
```

2. To configure different attributes for the system instance:
   - OS type – specifies the system's OS type.

     ```
     doca_error_t doca_apsh_sys_os_type_set(struct doca_apsh_system *ctx, enum doca_apsh_system_os
     os_type);
     ```

     ⚠ Currently supported types: Windows or Linux.

   - System representor – specifies the representor of the device connected to the system for App Shield to run on (which can be a representor of VF/PF). For information on querying the DOCA device, refer to the DOCA Core.
     After acquiring the DOCA device, use the following function to configure it into the system instance:

     ```
     doca_error_t doca_apsh_sys_dev_set(struct doca_apsh_system *system, struct doca_dev_rep *dev);
     ```

   - System symbols map – includes information about the OS that App Shield is attempting to run on (e.g., Window 10 Build 18363) and the size and fields of the OS structures, which helps App Shield with the memory forensic techniques it uses to access and analyze these structures in the system's memory. This can be obtained by running the **doca_apsh_config.py** on the system machine.
     After obtaining it, there are two options to provide the file:
     - The first option is to explicitly specify the file path:

       ```
       doca_error_t doca_apsh_sys_os_symbol_map_set(struct doca_apsh_system *system, const char
       *system_os_symbol_map_path);
       ```

     - The second option is to provide a folder where the file is located:

       ```
       doca_error_t doca_apsh_sys_os_symbol_map_folder_set(struct doca_apsh_system *system, const
       char *system_os_symbol_folder_path);
       ```

       ⚠ In this option, the file name must be maintained as **doca_apsh_config.py**.

   - Memory regions – includes the physical addresses of the memory regions which are mapped for system memory RAM. This is needed to prevent App Shield from

accessing other memory regions, such as memory mapped I/O regions. This can be obtained by running the **doca_apsh_config.py** tool on the system machine.
After obtaining it, run:

```
doca_error_t doca_apsh_sys_mem_region_set(struct doca_apsh_system *system, const char
*system_mem_region_path);
```

- KPGD file (optional and relevant only for Linux OS) – contains the KPGD physical address and the virtual address of **init_task**. This information is required since App Shield extracts data from the kernel struct in the physical memory. Thus, the kernel page directory table must translate the virtual addresses of these structs. This can be obtained by running the **doca_apsh_config.py** tool on the system machine with the flag **find_kpgd=1**. Since setting this attribute is optional, App Shield can work without it, but providing it speeds up App Shield's initialization process.
  After obtaining it, run:

```
doca_error_t doca_apsh_sys_kpgd_file_set(struct doca_apsh_system *system, const char
*system_kpgd_file_path);
```

3. To start the **doca_apsh_system**:

```
doca_error_t doca_apsh_system_start(struct doca_apsh_system *system);
```

4. To destroy the **doca_apsh_system** instance when it is no longer needed, call:

```
void doca_apsh_system_destroy(struct doca_apsh_system *system);
```

5.4.12.5.3 doca_apsh_config.py Tool

The **doca_apsh_config.py** tool is a python3 script which can be used to obtain all the attributes needed to run **doca_apsh_system** instance.

The following parameters are necessary to use the tool:

Parameter	Description
pid (optional)	The process ID of the process we want to run attestation capability on
os (mandatory)	The OS type of the machine (i.e., Linux or Windows)
find_kpgd (optional)	Relevant for Linux OS only, AS flag to enable/disable creating **kpgd_file.conf**. Default 0.
files (mandatory)	A list of files for the tool to create. File options: **hash**, **symbols**, **memregions**, **kpgd_file** (only relevant for Linux). ⚠ Make sure that the value set is appropriate for your setup.

Parameter	Description
path (mandatory)	• Linux – path to the **dwarf2json** executable. Default **./dwarf2json**. This file can be obtained by compiling the following project using Go. • Windows – path to **pdbparse-to-json.py**. Default **./pdbparse-to-json.py**. This file can be found here. ⚠ Make sure that the value set is appropriate for your setup.

The tool creates the following files:

- Symbol map – this file changes once the system kernel is updated or a kernel module is installed. The file does not change on system reboot.
- Memory regions – this file changes when adding or removing hardware or drivers that affect the system's memory map (e.g., when adding register addresses). The file does not change on system reboot.
- **hash.zip** – this file is required for attestation but is unnecessary for all other capabilities. The ZIP file contains the required data to attest to a single process. The file changes on library or executable update.
- **kpgd_file.conf** (relevant for Linux OS only) – helps with faster initialization of the library. The file changes on system reboot.

5.4.12.6 DOCA App Shield Samples

This section provides DOCA App Shield library sample implementations on top of BlueField.

ℹ All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

5.4.12.6.1 Sample Prerequisites

Follow the guidelines in section "Prerequisites" then copy the generated JSON files, **symbols.json** and **mem_regions.json**, to the **/tmp/** directory.

5.4.12.6.2 Running the Sample

1. Refer to the following documents:
   - DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - DOCA Troubleshooting for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

```
cd /opt/mellanox/doca/samples/doca_apsh/<sample_name>
meson /tmp/build
ninja -C /tmp/build
```

> The binary **doca_<sample_name>** will be created under **/tmp/build/**.

3. Sample (e.g., **apsh_libs_get**) usage:

```
Usage: doca_apsh_libs_get [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -p, --pid                         Process ID of process to be analyzed
  -f, --vuid                        VUID of the System device
  -d, --dma                         DMA device name
  -s, --osty <windows|linux>        System OS type - windows/linux
```

4. For additional information per sample, use the **-h** option:

```
/tmp/build/doca_<sample_name> -h
```

5.4.12.6.3 Samples

> These sample are also https://github.com/NVIDIA-DOCA/doca-samples/tree/master/samples/doca_apsh.

5.4.12.6.3.1 Apsh Libs Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of loadable libraries of a specific process.

The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCIe device via given vendor unique identifier (VUID).
5. Creating DOCA Apsh system handler.
6. Setting fields and starting Apsh system handler.
7. Getting the list of system process using Apsh API and searching for a specific process with the given PID.
8. Getting the list of process-loadable libraries using **doca_apsh_libs_get** Apsh API call.
9. Querying the libraries for 3 selected fields using **doca_apsh_lib_info_get** Apsh API call.
10. Printing libraries' attributes to the terminal.
11. Cleaning up.

References:

- **/opt/mellanox/doca/samples/doca_apsh/apsh_libs_get/apsh_libs_get_sample.c**
- **/opt/mellanox/doca/samples/doca_apsh/apsh_libs_get/apsh_libs_get_main.c**

- **/opt/mellanox/doca/samples/doca_apsh/apsh_libs_get/meson.build**
- **/opt/mellanox/doca/samples/doca_apsh/apsh_common.c** ; **/opt/mellanox/doca/samples/doca_apsh/apsh_common.h**

5.4.12.6.3.2 Apsh Modules Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of installed modules on a monitored system.
The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCIe device via given VUID.
5. Creating DOCA Apsh system handler.
6. Setting fields and start Apsh system handler.
7. Getting the the list of system-installed modules using **doca_apsh_modules_get** Apsh API call.
8. Querying the names of modules using **doca_apsh_module_info_get** Apsh API call.
9. Printing the attributes of up to 5 modules attributes to the terminal.
10. Cleaning up.

References:

- **/opt/mellanox/doca/samples/doca_apsh/apsh_modules_get/apsh_modules_get_sample.c**
- **/opt/mellanox/doca/samples/doca_apsh/apsh_modules_get/apsh_modules_get_main.c**
- **/opt/mellanox/doca/samples/doca_apsh/apsh_modules_get/meson.build**
- **/opt/mellanox/doca/samples/doca_apsh/apsh_common.c** ; **/opt/mellanox/doca/samples/doca_apsh/apsh_common.h**

5.4.12.6.3.3 Apsh Pslist

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of running processes on a monitored system.
The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCIe device via given VUID.
5. Creating DOCA Apsh system handler.
6. Setting fields and starting Apsh system handler.
7. Getting the list of processes running on the system using **doca_apsh_processes_get** Apsh API call.
8. Querying the processes for 4 chosen attributes using **doca_apsh_proc_info_get** Apsh API call.
9. Printing the attributes of up to 5 processes to the terminal.
10. Cleaning up.

References:

- **/opt/mellanox/doca/samples/doca_apsh/apsh_pslist/apsh_pslist_sample.c**
- **/opt/mellanox/doca/samples/doca_apsh/apsh_pslist/apsh_pslist_main.c**
- **/opt/mellanox/doca/samples/doca_apsh/apsh_pslist/meson.build**
- **/opt/mellanox/doca/samples/doca_apsh/apsh_common.c** ; **/opt/mellanox/doca/samples/doca_apsh/apsh_common.h**

5.4.12.6.3.4 Apsh Threads Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of threads of a specific process.
The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCIe device via given VUID.
5. Creating DOCA Apsh system handler.
6. Setting fields and starting Apsh system handler.
7. Getting the list of system processes using Apsh API and searching for a specific process with the given PID.
8. Getting the list of process threads using **doca_apsh_threads_get** Apsh API call.
9. Querying the threads for up to 3 selected fields using **doca_apsh_thread_info_get** Apsh API call.
10. Printing thread attributes to the terminal.
11. Cleaning up.

References:

- **/opt/mellanox/doca/samples/doca_apsh/apsh_threads_get/apsh_threads_get_sample.c**
- **/opt/mellanox/doca/samples/doca_apsh/apsh_threads_get/apsh_threads_get_main.c**
- **/opt/mellanox/doca/samples/doca_apsh/apsh_threads_get/meson.build**
- **/opt/mellanox/doca/samples/doca_apsh/apsh_common.c** ; **/opt/mellanox/doca/samples/doca_apsh/apsh_common.h**

5.4.12.6.3.5 Apsh Vads Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of virtual address descriptors (VADs) of a specific process.
The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and start the Apsh context.
4. Opening DOCA remote PCIe device via given VUID.
5. Creating DOCA Apsh system handler.
6. Setting fields and starting Apsh system handler.
7. Getting the list of system processes using Apsh API and searching for a specific process with the given PID.

8. Getting the list of process VADs using `doca_apsh_vads_get` Apsh API call.
9. Querying the VADs for 3 selected fields using `doca_apsh_vad_info_get` Apsh API call.
10. Printing the attributes of up to 5 VADs to the terminal.
11. Cleaning up.

References:

- `/opt/mellanox/doca/samples/doca_apsh/apsh_vads_get/apsh_vads_get_sample.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_vads_get/apsh_vads_get_main.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_vads_get/meson.build`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_common.c` ; `/opt/mellanox/doca/samples/doca_apsh/apsh_common.h`

5.4.12.6.3.6 Apsh Envars Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of environment variables of a specific process.

> ⚠ This sample works only on target systems with Windows OS.

The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCIe device via given VUID.
5. Creating DOCA Apsh system handler.
6. Setting fields and starting Apsh system handler.
7. Getting the list of system processes using Apsh API and searching for a specific process with the given PID.
8. Getting the list of process envars using `doca_apsh_envars_get` Apsh API call.
9. Querying the envars for 2 selected fields using `doca_apsh_envar_info_get` Apsh API call.
10. Printing the envars attributes to the terminal.
11. Cleaning up.

References:

- `/opt/mellanox/doca/samples/doca_apsh/apsh_envars_get/apsh_envars_get_sample.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_envars_get/apsh_envars_get_main.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_envars_get/meson.build`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_common.c` ; `/opt/mellanox/doca/samples/doca_apsh/apsh_common.h`

5.4.12.6.3.7 Apsh Privileges Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of privileges of a specific process.

⚠ This sample works only on target systems with Windows OS.

The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCIe device via given VUID.
5. Creating DOCA Apsh system handler.
6. Setting fields and starting Apsh system handler.
7. Getting the list of system processes using Apsh API and searching for a specific process with the given PID.
8. Getting the list of process privileges using the **doca_apsh_privileges_get** Apsh API call.
9. Querying the privileges for 5 selected fields using the **doca_apsh_privilege_info_get** Apsh API call.
10. Printing the privileges attributes to the terminal.
11. Cleaning up.

References:

- **/opt/mellanox/doca/samples/doca_apsh/apsh_privileges_get/apsh_privileges_get_sample.c**
- **/opt/mellanox/doca/samples/doca_apsh/apsh_privileges_get/apsh_privileges_get_main.c**
- **/opt/mellanox/doca/samples/doca_apsh/apsh_privileges_get/meson.build**
- **/opt/mellanox/doca/samples/doca_apsh/apsh_common.c** ; **/opt/mellanox/doca/samples/doca_apsh/apsh_common.h**

5.4.12.6.3.8 Apsh Containers Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of running containers on a monitored system, as well as getting a list of processes for each container.

⚠ This sample works only on target systems with Linux OS.

The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCIe device using specific VUID.
5. Creating DOCA Apsh system handler.
6. Setting fields and starting Apsh system handler.
7. Getting the list of containers running on the system using **doca_apsh_containers_get** Apsh API call.
8. Querying the containers for container ID attribute using **doca_apsh_container_info_get** Apsh API call.
9. Getting list of processes for each container using **doca_apsh_container_processes_get** Apsh API call.

10. Printing the attributes of up to 5 processes to the terminal.
11. Cleaning up.

References:

- **/opt/mellanox/doca/samples/doca_apsh/apsh_containers_get/apsh_containers_get_sample.c**
- **/opt/mellanox/doca/samples/doca_apsh/apsh_containers_get/apsh_containers_get_main.c**
- **/opt/mellanox/doca/samples/doca_apsh/apsh_containers_get/meson.build**
- **/opt/mellanox/doca/samples/doca_apsh/apsh_common.c** ; **/opt/mellanox/doca/samples/doca_apsh/apsh_common.h**

5.4.12.6.3.9 Apsh Netscan Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of open network connections on a monitored system.

This sample works only on target systems with Linux OS and specific Windows OS builds.

The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCIe device using specific VUID.
5. Creating DOCA Apsh system handler.
6. Setting fields and starting Apsh system handler.
7. Getting the list of open connections on the system using **doca_apsh_netscan_get** Apsh API call.
8. Querying the connections for some of the connection attributes using **doca_apsh_netscan_info_get** Apsh API call.
9. Printing the attributes of all connection to the terminal.
10. Cleaning up.

References:

- **/opt/mellanox/doca/samples/doca_apsh/apsh_netscan_get/apsh_netscan_get_sample.c**
- **/opt/mellanox/doca/samples/doca_apsh/apsh_netscan_get/apsh_netscan_get_main.c**
- **/opt/mellanox/doca/samples/doca_apsh/apsh_netscan_get/meson.build**
- **/opt/mellanox/doca/samples/doca_apsh/apsh_common.c** ; **/opt/mellanox/doca/samples/doca_apsh/apsh_common.h**

5.4.12.6.3.10 Apsh Process Netscan Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of open network connections of a specific process.

This sample works only on target systems with Linux OS and specific Windows OS builds.

The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCIe device via given VUID.
5. Creating DOCA Apsh system handler.
6. Setting fields and starting Apsh system handler.
7. Getting the list of system processes using Apsh API and searching for a specific process with the given PID.
8. Getting the list of process connections using `doca_apsh_process_netscan_get` Apsh API call.
9. Querying the connections for some of the connection attributes using `doca_apsh_netscan_info_get` Apsh API call.
10. Printing connection attributes to the terminal.
11. Cleaning up.

References:

- `/opt/mellanox/doca/samples/doca_apsh/apsh_process_netscan_get/apsh_process_netscan_get_sample.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_process_netscan_get/apsh_process_netscan_get_main.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_process_netscan_get/meson.build`

`/opt/mellanox/doca/samples/doca_apsh/apsh_common.c` ; `/opt/mellanox/doca/samples/doca_apsh/apsh_common.h`

5.4.12.6.3.11 Apsh Interfaces Get

This sample illustrates how to properly initialize DOCA App Shield and use its API to get the list of interfaces in the system.

> ⚠ This sample works only on target systems with Linux OS.

The sample logic includes:

1. Opening DOCA device with DMA ability.
2. Creating DOCA Apsh context.
3. Setting and starting the Apsh context.
4. Opening DOCA remote PCIe device via given VUID.
5. Creating DOCA Apsh system handler.
6. Setting fields and starting Apsh system handler.
7. Getting the list of system interfaces using `doca_apsh_interfaces_get` Apsh API call.
8. Printing interfaces attributes to the terminal.
9. Cleaning up.

References:

- `/opt/mellanox/doca/samples/doca_apsh/apsh_interfaces_get/apsh_interfaces_get_sample.c`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_interfaces_get/apsh_interfaces_get_main.c`

- `/opt/mellanox/doca/samples/doca_apsh/apsh_interfaces_get/meson.build`
- `/opt/mellanox/doca/samples/doca_apsh/apsh_common.c` ; `/opt/mellanox/doca/samples/doca_apsh/apsh_common.h`

5.4.13 DOCA Compress

This guide provides instructions on how to use the DOCA Compress API.

5.4.13.1 Introduction

> ⚠ The quality status of DOCA libraries is listed here.

DOCA Compress library provides an API to compress and decompress data using hardware acceleration, supporting both host and NVIDIA® BlueField® DPU memory regions.

The library provides an API for executing compress operations on DOCA buffers, where these buffers reside in either the DPU memory or host memory.

Using DOCA Compress, compress and decompress memory operations can be easily executed in an optimized, hardware-accelerated manner.

This document is intended for software developers wishing to accelerate their application's compress memory operations.

5.4.13.2 Prerequisites

The DOCA Compress library follows the architecture of a DOCA Core Context. It is recommended to read the following sections before proceeding:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

5.4.13.3 Changes From Previous Releases

5.4.13.3.1 Changes in 2.9.0

N/A

5.4.13.4 Environment

DOCA Compress-based applications can run either on the host machine or on the BlueField DPU target.

Compress can only be run with a DPU configured with DPU mode as described in BlueField Modes of Operation.

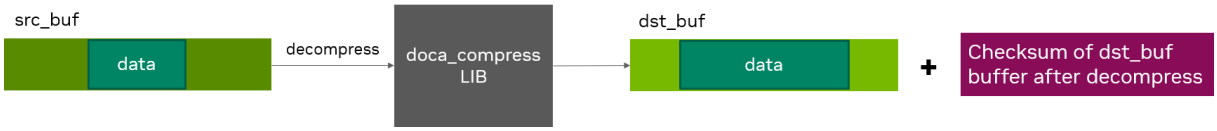
5.4.13.5 Architecture

DOCA Compress is a DOCA Context as defined by DOCA Core. See NVIDA DOCA Core Context for more information.

DOCA Compress leverages DOCA Core architecture to expose asynchronous tasks that are offloaded to hardware.

Compress operation:

Decompress operation:



5.4.13.5.1 Supported Compress/Decompress Algorithms

For BlueField-2 devices, this library supports:

- Compress operation using the deflate algorithm
- Decompress operation using the deflate algorithm

For BlueField-3 devices, this library supports:

- Decompress operation using the deflate algorithm
- Decompress operation using the LZ4 algorithm

5.4.13.5.2 Supported Checksum Methods

Depending on the task type, the following checksum methods are produced and may be retrieved using the relevant getter functions:

- Adler – produced by the deflate compress and decompress tasks
- CRC – produced by all tasks
- xxHash – produced by the LZ4 decompress tasks

Refer to "Tasks" section for more information.

5.4.13.5.3 Objects

5.4.13.5.3.1 Device and Device Representor

The library requires a DOCA device to operate, the device is used to access memory and perform the actual copy. See DOCA Core Device Discovery for information.

For same BlueField DPU, it does not matter which device is used (PF/VF/SF), as all these devices utilize the same hardware component. If there are multiple DPUs, it is possible to create a Compress instance per DPU, providing each instance with a device from a different DPU.

To access memory that is not local (from the host to the DPU or vice versa), then the DPU side of the application must pick a device with an appropriate representor. See DOCA Core Device Representor Discovery.

The device must stay valid as long as the Compress instance is not destroyed.

5.4.13.5.3.2 Memory Buffers

All compress/decompress tasks require two DOCA buffers containing the destination and the source. Depending on the allocation pattern of the buffers, refer to the Inventory Types table.

Buffers must not be modified or read during the compress/decompress operation.

5.4.13.5.4 Source and Destination Location

DOCA Compress can process DOCA buffers that reside on the host, the DPU, or both.

5.4.13.5.4.1 Local Host

Source and destination buffers reside on the host and the compress library runs on the host.

5.4.13.5.4.2 Local DPU

Source and destination buffers reside on the DPU and the compress library runs on the DPU.

5.4.13.5.4.3 Remote

Source at Host, Destination at DPU

- The source resides on the host and is exported (DOCA mmap export) to the DPU
- The destination resides on the DPU
- The compress library runs on the DPU and compresses/decompresses the host source to the DPU destination

Source at DPU, Destination at Host

- The source resides on the DPU
- The destination resides on the host and is exported (DOCA mmap export) to the DPU
- Compress library runs on the DPU and compresses/decompresses the DPU source to the host destination

5.4.13.6 Configuration Phase

To start using the library, the user must go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context, to allow execution of tasks and retrieval of events.

5.4.13.6.1 Configurations

The context can be configured to match the use case of the application.

To find if a configuration is supported or what its min/max value is, refer to Device Support.

5.4.13.6.1.1 Mandatory Configurations

The following configurations must be set by the application before attempting to start the context:

- At least one task/event type must be configured. See configuration of Tasks.
- A device with appropriate support must be provided upon creation

5.4.13.6.2 Device Support

DOCA Compress requires a device to operate. To pick a device, see DOCA Core Device Discovery.

As device capabilities may change in the future (see DOCA Core Device Support), it is recommended to select your device using the following APIs:

5.4.13.6.2.1 Supported Tasks

- **doca_compress_cap_task_compress_deflate_is_supported**
- **doca_compress_cap_task_decompress_deflate_is_supported**
- **doca_compress_cap_task_decompress_lz4_stream_is_supported**
- **doca_compress_cap_task_decompress_lz4_block_is_supported**

5.4.13.6.2.2 Supported Buffer Size

- **doca_compress_cap_task_compress_deflate_get_max_buf_size**
- **doca_compress_cap_task_decompress_deflate_get_max_buf_size**
- **doca_compress_cap_task_decompress_lz4_stream_get_max_buf_size**
- **doca_compress_cap_task_decompress_lz4_block_get_max_buf_size**

5.4.13.6.3 Buffer Support

Tasks support buffers with the following features:

Buffer Type	Source Buffer	Destination Buffer
Linked List Buffer	Yes	No
Local mmap Buffer	Yes	Yes

Buffer Type	Source Buffer	Destination Buffer
mmap From PCI Export Buffer	Yes	Yes
mmap From RDMA Export Buffer	No	No

5.4.13.7 Execution Phase

This section describes execution on CPU or DPU using DOCA Core Progress Engine.

5.4.13.7.1 Tasks

5.4.13.7.1.1 Task Batching

DOCA Compress supports task batching mode, which is a task submit mode of work that allows aggregating multiple DOCA tasks of the same type and handling them as a single unit.

> For more information on task batching, refer to DOCA Core Task.

DOCA Compress supports only the `DOCA_TASK_SUBMIT_FLAG_NONE`, `DOCA_TASK_SUBMIT_FLAG_FLUSH` flag.

5.4.13.7.1.2 Compress Deflate Task

This task facilitates compressing memory, with the deflate algorithm, using buffers as described in section "Buffer Support".

> DOCA compress returns only the payload. To create a compressed file, (e.g., gzip), the developer must add a gzip header/trailer.

Task Configuration

Description	API to set the configuration	API to query support
Enable the task	`doca_compress_task_compress_deflate_set_conf`	`doca_compress_cap_task_compress_deflate_is_supported`
Number of tasks	`doca_compress_task_compress_deflate_set_conf`	`doca_compress_get_max_num_tasks (max total num tasks)`
Maximal buffer size	–	`doca_compress_cap_task_compress_deflate_get_max_buf_size`
Maximum buffer list size	–	`doca_compress_cap_task_compress_deflate_get_max_buf_list_len`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Buffer pointing to the memory to be compressed	Only the data residing in the data segment is compressed
Destination buffer	Buffer pointing to where compressed memory will be stored	The data is compressed to the tail segment extending the data segment

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The source data is compressed to destination
- The destination buffer data segment is extended to include the compressed data
- Adler can be retrieved by calling `doca_compress_task_compress_deflate_get_adler_cs`
- CRC can be retrieved by calling `doca_compress_task_compress_deflate_get_crc_cs`

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the source and destination should not be read/written to
- Source and destination must not overlap
- Other limitations are described in DOCA Core Task

5.4.13.7.1.3 Decompress Deflate Task

This task facilitates decompressing memory, with the deflate algorithm, using buffers as described in section "Buffer Support".

> ⚠ DOCA decompress expects the payload alone. To decompress a file (e.g. gzip), the developer must strip the header/trailer.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_compress_task_decompress_deflate_set_conf`	`doca_compress_cap_task_decompress_deflate_is_supported`

Description	API to Set the Configuration	API to Query Support
Number of tasks	`doca_compress_task_decompress_deflate_set_conf`	`doca_compress_get_max_num_tasks (max-total-num-tasks)`
Maximal buffer size	–	`doca_compress_cap_task_decompress_deflate_get_max_buf_size`
Maximum buffer list size	–	`doca_compress_cap_task_decompress_deflate_get_max_buf_list_len`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
source buffer	Buffer pointing to the memory to be decompressed	Only the data residing in the data segment is decompressed
destination buffer	Buffer pointing to where decompressed memory will be stored	The data is decompressed to the tail segment extending the data segment

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The source data is decompressed to destination
- The destination buffer data segment is extended to include the decompressed data
- Adler can be retrieved by calling `doca_compress_task_decompress_deflate_get_adler_cs`
- CRC can be retrieved by calling `doca_compress_task_decompress_deflate_get_crc_cs`

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the source and destination should not be read/written to
- Source and destination must not overlap
- Other limitations are described in DOCA Core Task

5.4.13.7.1.4 Decompress LZ4 Tasks

These tasks facilitate decompressing memory with the LZ4 algorithm, using buffers as described in section "Buffer Support".

The main differences between the tasks is the input data format –

- The decompress LZ4 stream task expects a stream of one or more blocks, without the frame (i.e., the magic number, frame descriptor, and content checksum)
- The decompress LZ4 block task expects a single, compressed, data-only block (i.e., without block size or block checksum)

Decompress LZ4 Stream Task

This task facilitates decompressing memory with the LZ4 algorithm, using buffers as described in section "Buffer Support".

> ⚠ The decompress LZ4 stream task expects a stream of one or more blocks without the frame (i.e., the magic number, frame descriptor, and content checksum).

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_compress_task_decompress_lz4_stream_set_conf`	`doca_compress_cap_task_decompress_lz4_stream_is_supported`
Number of tasks	`doca_compress_task_decompress_lz4_stream_set_conf`	`doca_compress_get_max_num_tasks (max total num tasks)`
Maximal buffer size	–	`doca_compress_cap_task_decompress_lz4_stream_get_max_buf_size`
Maximum buffer list size	–	`doca_compress_cap_task_decompress_lz4_stream_get_max_buf_list_len`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Has block checksum Flag	A flag to indicate whether or not the blocks in the stream have a checksum	1 if the task should expect blocks in the stream to have a checksum; 0 otherwise
Are blocks independent flag	A flag to indicate whether or not each block depends on previous blocks in the stream	1 the the task should expect blocks to be independent; 0 otherwise (dependent blocks)
Source buffer	Buffer pointing to the memory to be decompressed	Only the data residing in the data segment is decompressed
Destination buffer	Buffer pointing to where decompressed memory will be stored	The data is decompressed to the tail segment extending the data segment

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully:

- The source data is decompressed to destination
- The destination buffer data segment is extended to include the decompressed data
- CRC can be retrieved by calling `doca_compress_task_decompress_lz4_stream_get_crc_cs`
- xxHash can be retrieved by calling `doca_compress_task_decompress_lz4_stream_get_xxh_cs`

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the source and destination should not be read/written to
- Source and destination must not overlap
- Other limitations are described in DOCA Core Task

Decompress LZ4 Block Task

This task facilitates decompressing memory with the LZ4 algorithm, using buffers as described in section "Buffer Support".

> ⚠ The decompress LZ4 block task expects a single, compressed, data-only block (i.e., without block size or block checksum).

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_compress_task_decompress_lz4_block_set_conf`	`doca_compress_cap_task_decompress_lz4_block_is_supported`
Number of tasks	`doca_compress_task_decompress_lz4_block_set_conf`	`doca_compress_get_max_num_tasks (max total num tasks)`
Maximal buffer size	–	`doca_compress_cap_task_decompress_lz4_block_get_max_buf_size`
Maximum buffer list size	–	`doca_compress_cap_task_decompress_lz4_block_get_max_buf_list_len`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Buffer pointing to the memory to be decompressed	Only the data residing in the data segment will be decompressed
Destination buffer	Buffer pointing to where decompressed memory will be stored	The data is decompressed to the tail segment extending the data segment

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully:

- The source data is decompressed to destination
- The destination buffer data segment is extended to include the decompressed data
- CRC can be retrieved by calling **doca_compress_task_decompress_lz4_block_get_crc_cs**
- xxHash can be retrieved by calling **doca_compress_task_decompress_lz4_bloxk_get_xxh_cs**

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination **doca_buf** objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the source and destination should not be read/written to
- Source and destination must not overlap
- Other limitations are described in DOCA Core Task

5.4.13.7.2 Events

DOCA Compress exposes asynchronous events to notify about changes that happen unexpectedly according to DOCA Core architecture.

The only events DOCA Compress expose are common events (DOCA CTX state changed). See more info in DOCA Core Event.

5.4.13.8 State Machine

The DOCA Compress library follows the Context state machine described in DOCA Core Context State Machine.

This section describes how to move states and what is allowed in each state.

5.4.13.8.1 States

5.4.13.8.1.1 Idle

In this state, it is expected that application:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to Configurations
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop after making sure all tasks have been freed
Stopping	Call progress until all tasks are completed and freed

5.4.13.8.1.2 Starting

This state cannot be reached.

5.4.13.8.1.3 Running

In this state, it is expected that application:

- Allocates and submit tasks
- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocate previously configured task
- Submit a task
- Call stop

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

5.4.13.8.1.4 Stopping

In this state, it is expected that application:

- Calls progress to complete all inflight tasks (tasks will complete with failure)

- Frees any completed tasks

Allowed operations:

- Call progress

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call progress and fatal error occurs
Running	Call stop without freeing all tasks

5.4.13.9 Alternative Datapath Options

DOCA Compress only supports datapath on CPU, see Execution Phase.

5.4.13.10 DOCA Compress Samples

The following samples illustrate how to use the DOCA Compress API to compress and decompress files.

> ⚠ DOCA Compress handles payload only unless the **zc** flag is used (available only for deflate samples). In that case, a zlib header and trailer are added in compression and it is considered as part of the input when decompressing.

> ℹ All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

5.4.13.10.1 Running the Sample

1. Refer to the following documents:
   - DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - DOCA Troubleshooting for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

   ```
   cd /opt/mellanox/doca/samples/doca_compress/<sample_name>
   meson /tmp/build
   ninja -C /tmp/build
   ```

   > ℹ The binary **doca_<sample_name>** is created under **/tmp/build/**.

3. Sample (e.g., **doca_compress_deflate**) usage:
   - Common arguments

```
Usage: doca_<sample_name> [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -p, --pci-addr                    DOCA device PCI device address
  -f, --file                        Input file to compress/decompress
  -o, --output                      Output file
  -c, --output-checksum             Output checksum
```

- Sample-specific arguments

Sample	Argument	Description
Compress/ Decompress Deflate	**-wf** , **-with-frame**	Write/read a file with a frame, compatible with default zlib settings
Decompress LZ4 Stream	**-bc** , **--has-block-checksum**	Flag to indicate if blocks have a checksum
	-bi , **--are-blocks-independent**	Flag to indicate if blocks are independent
	-wf , **-with-frame**	Read a file compatible with an LZ4 frame

4. For additional information per sample, use the **-h** option:

```
/tmp/build/doca_<sample_name> -h
```

5.4.13.10.2 Samples

These samples are also available on GitHub.

5.4.13.10.2.1 Compress/Decompress Deflate

This sample illustrates how to use DOCA Compress library to compress or decompress a file.

The sample logic includes:

1. Locating a DOCA device.
2. Initializing the required DOCA Core structures.
3. Populating DOCA memory map with two relevant buffers; one for the source data and one for the result.
4. Allocating elements in DOCA buffer inventory for each buffer.
5. Allocating and initializing a DOCA Compress deflate task or a DOCA Decompress deflate task.
6. Submitting the task.
7. Running the progress engine until the task is completed.
8. Writing the result into an output file, **out.txt** .

9. Destroying all DOCA Compress and DOCA Core structures.

References:

- **/opt/mellanox/doca/samples/doca_compress/compress_deflate/compress_deflate_sample.c**
- **/opt/mellanox/doca/samples/doca_compress/compress_deflate/compress_deflate_main.c**
- **/opt/mellanox/doca/samples/doca_compress/compress_deflate/meson.build**
- **/opt/mellanox/doca/samples/doca_compress/decompress_deflate/decompress_deflate_sample.c**
- **/opt/mellanox/doca/samples/doca_compress/decompress_deflate/decompress_deflate_main.c**
- **/opt/mellanox/doca/samples/doca_compress/decompress_deflate/meson.build**
- **/opt/mellanox/doca/samples/doca_compress/compress_common.h**
- **/opt/mellanox/doca/samples/doca_compress/compress_common.c**

5.4.13.10.2.2 Decompress LZ4 Stream

This sample illustrates how to use DOCA Compress library to decompress a file using the LZ4 stream decompress task.

The sample logic includes:

1. Locating a DOCA device.
2. Initializing the required DOCA Core structures.
3. Populating DOCA memory map with two relevant buffers; one for the source data and one for the result.
4. Allocating elements in DOCA buffer inventory for each buffer.
5. Allocating and initializing an DOCA Decompress LZ4 stream task.
6. Submitting the task.
7. Running the progress engine until the task is completed.
8. Writing the result into an output file, **out.txt** .
9. Destroying all DOCA Compress and DOCA Core structures.

References:

- **/opt/mellanox/doca/samples/doca_compress/decompress_lz4_stream/decompress_lz4_stream_sample.c**
- **/opt/mellanox/doca/samples/doca_compress/decompress_lz4_stream/decompress_lz4_stream_main.c**
- **/opt/mellanox/doca/samples/doca_compress/decompress_lz4_stream/meson.build**
- **/opt/mellanox/doca/samples/doca_compress/compress_common.h**
- **/opt/mellanox/doca/samples/doca_compress/compress_common.c**

5.4.13.10.3 Backward Compatibility

5.4.13.10.3.1 Decompress LZ4 Task

The decompress LZ4 task has been removed. To facilitates decompressing memory with the LZ4 algorithm, use the decompress LZ4 stream task or the decompress LZ4 block task instead.

5.4.14 DOCA SHA

This guide provides instructions on building and developing applications that calculate message digest using the SHA1, SHA2-256, or SHA2-512 algorithms.

5.4.14.1 Introduction

> ⚠ The quality status of DOCA libraries is listed here.

The library provides an API for executing SHA operations on DOCA buffers, where the buffers reside in either local memory (i.e., within the same host) or host memory accessible by the NVIDIA® BlueField®-2 device (remote memory). Using DOCA SHA, complex cryptographic hash operations can be easily executed in an optimized, hardware-accelerated manner.

> ⚠ NVIDIA BlueField-3 does not support this library because it has no SHA acceleration engine.

This document is intended for software developers wishing to accelerate their applications' SHA calculations typically used in digital signature schemes or hash-based message authentication code calculations.

5.4.14.2 Prerequisites

This library follows the architecture of the DOCA Core context:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

5.4.14.3 Environment

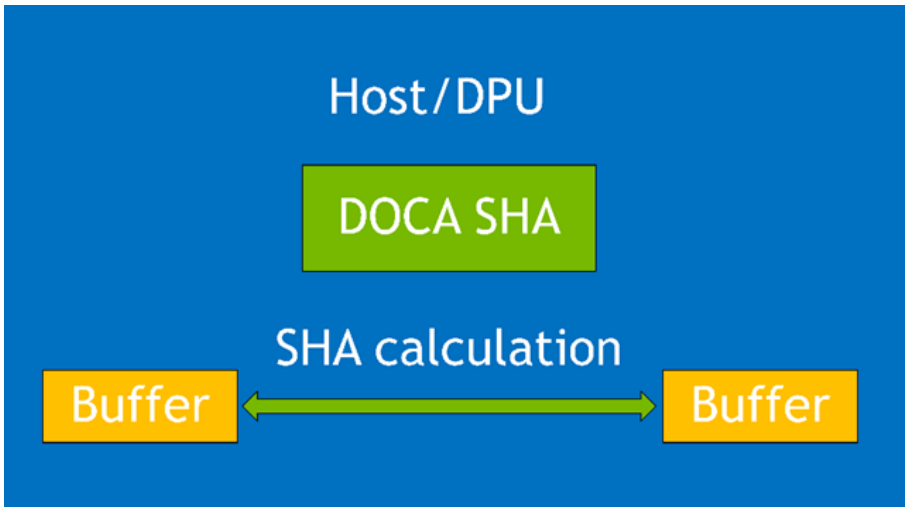
DOCA SHA-based applications can run either on the host machine or on the BlueField-2 DPU target. You must configure the DPU in DPU mode to run DOCA SHA calculations from the host to BlueField (or from BlueField to host).

5.4.14.4 Architecture

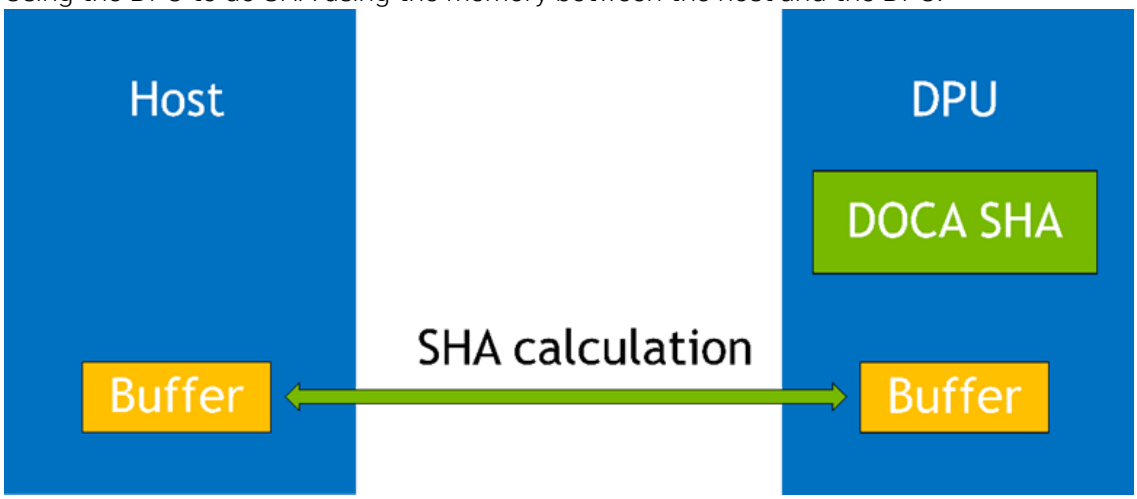
DOCA SHA is a DOCA Core Context. This library leverages the DOCA Core architecture to expose asynchronous tasks or events offloaded to hardware.

SHA can be used to calculate message digest as illustrated in the following diagrams:

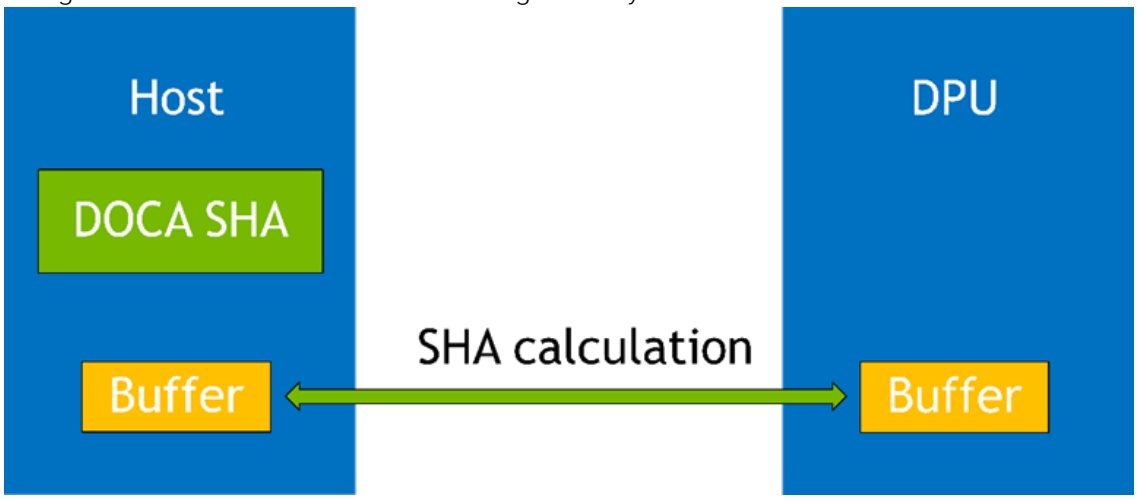
- SHA from local memory to local memory:



- Using the DPU to do SHA using the memory between the host and the DPU:



- Using the host to do SHA calculation using memory between the host and the DPU:



5.4.14.4.1 Objects

5.4.14.4.1.1 Device and Representor

The library requires a DOCA device to operate. The device is used to access memory and perform the actual SHA calculation. See DOCA Core Device Discovery.

For the same BlueField DPU, it does not matter which device is used (i.e., PF/VF/SF) as these devices utilize the same hardware component. If there are multiple DPUs, then it is possible to create a SHA instance per DPU, providing each instance with a device from a different DPU.

To access non-local memory (i.e., from the host to DPU or vice versa), the DPU side of the application must choose a device with an appropriate representor (see DOCA Core Device Representor Discovery). The device must stay valid for as long as the SHA instance is not destroyed.

5.4.14.4.1.2 Memory Buffers

The SHA task requires at least two DOCA buffers containing the destination and the source.

The destination is always a single **doca_buf**. The source can be a single **doca_buf** or a **linked_list** of **doca_buf**. All destination and source **doca_buf**s can be allocated from **doca_buf_inventory**.

> For the usage of **doca_buf_inventory**, please refer to the DOCA Core Inventory Types table.

Buffers must not be modified or read during the SHA operation. For information on what kind of memory is supported, refer to the table in section "Buffer Support".

5.4.14.5 Configuration Phase

To start using the library, users must go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context to allow the execution of tasks and retrieval of events.

5.4.14.5.1 Configurations

The context can be configured to match the application use case.

To find if a configuration is supported or its min/max value, refer to Device Support.

5.4.14.5.1.1 Mandatory Configurations

These configurations must be set by the application before attempting to start the context:

- You must configure at least one task/event type. See configuration of tasks and/or events in Tasks and Events respectively for information.
- A device with appropriate support must be provided upon creation.

5.4.14.5.2 Device Support

DOCA SHA requires a device to operate. For information on choosing a device, see DOCA Core Device Discovery.

As device capabilities may change in the future (see DOCA Core Device Support), NVIDIA recommends selecting your device using the following methods:

- `doca_sha_cap_task_hash_get_supported`
- `doca_sha_cap_task_partial_hash_get_supported`

Some devices allow for different capabilities, for example:

- The maximum number of tasks
- The maximum source buffer size
- The minimum destination buffer size
- The maximum supported number of elements in DOCA linked-list buffer
- Check whether SHA1, SHA2-256 or SHA2-512 is supported

5.4.14.5.3 Buffer Support

Tasks support buffers with the following features:

Buffer Type	Source Buffer	Destination Buffer
Local mmap buffer	Yes	Yes
Mmap from PCIe export buffer	Yes	Yes
Mmap from RDMA export buffer	No	No
Linked list buffer	Yes	No

5.4.14.6 Execution Phase

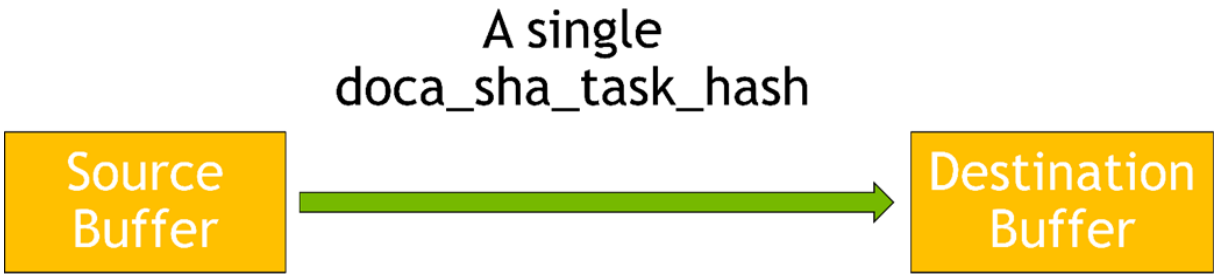
This section describes execution on the CPU using DOCA Core Progress Engine.

5.4.14.6.1 Tasks

DOCA SHA exposes asynchronous tasks that leverage DPU hardware according to DOCA Core architecture.

5.4.14.6.1.1 SHA Task

The SHA task `doca_sha_task_hash` allows one-shot SHA calculation using buffers as described in Buffer Support. One-shot means that the source buffer is used as a whole input. Therefore, the SHA operation is completed after this task completion event arrives.



Task Configuration

Description	API to Set Configuration	API to Query Support
Enable the task	`doca_sha_task_hash_set_conf`	`doca_sha_cap_task_hash_get_supported`
Number of tasks	`doca_sha_task_hash_set_conf`	`doca_sha_cap_get_max_num_tasks`
Maximal source buffer size	–	`doca_sha_cap_get_max_src_buf_size`
Maximum source buffer list size	–	`doca_sha_cap_get_max_list_buf_num_elem`
Minimum destination buffer size	–	`doca_sha_cap_get_min_dst_buf_size`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Buffer pointing to the memory to be used for SHA calculation	Only the data residing in the data segment is to be used
Destination buffer	Buffer pointing to the memory used for writing the SHA calculation result	The SHA result is appended to the tail segment
SHA algorithm type	SHA algorithm to be used in SHA calculation	Must be one of `DOCA_SHA_ALGORITHM_SHA1`, `DOCA_SHA_ALGORITHM_SHA256`, `DOCA_SHA_ALGORITHM_SHA512`

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The SHA calculation of data from the source buffer is successfully completed and the result is written to the destination buffer
- The destination buffer data segment is extended to include the SHA result data

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

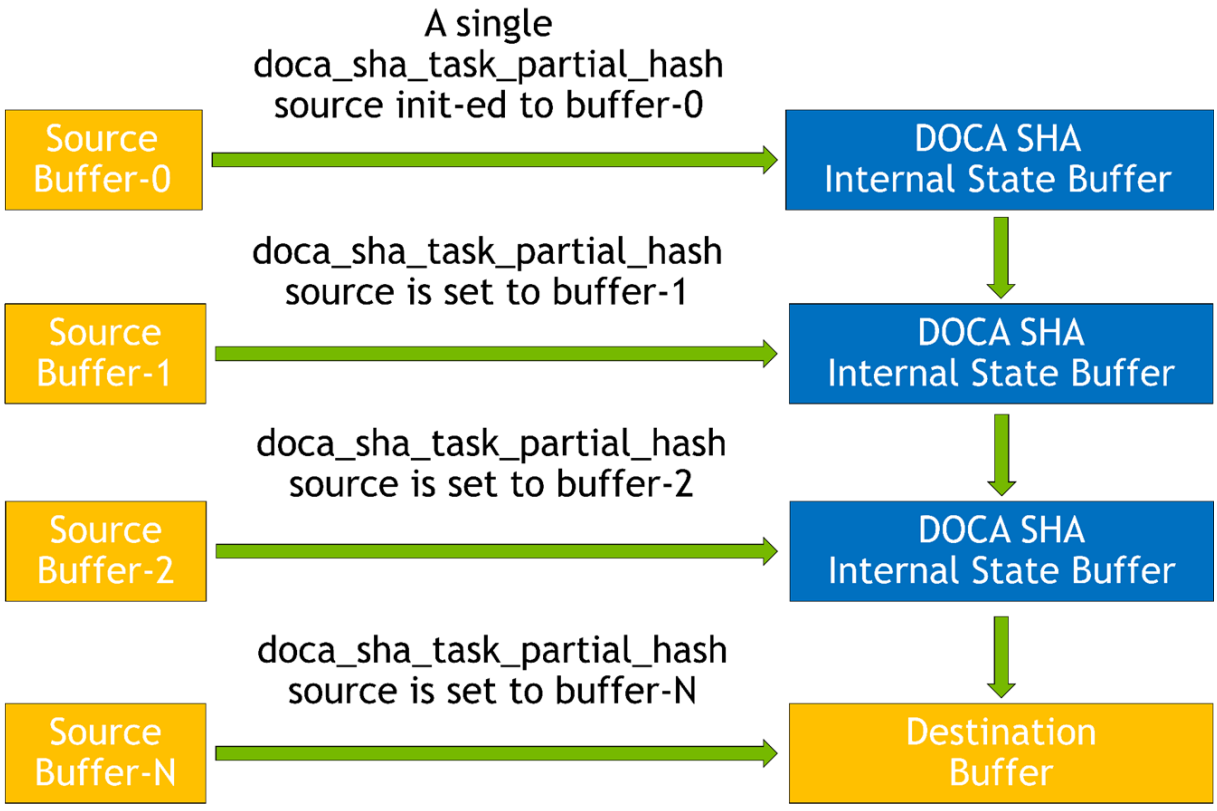
Task Limitations

- The operation is not atomic
- Once the task is submitted, the source and destination should not be read/written to
- Other limitations are described in DOCA Core Task

5.4.14.6.1.2 Partial-SHA Task

The partial-SHA task `doca_sha_task_partial_hash` allows stateful SHA calculation for a collection of messages using buffers.

Stateful means that the input data is composed of many segments (may be spatial or timely non-consecutive), therefore, its SHA calculation requires more than one one-shot SHA operation to finish. During any stateful operation, other independent SHA tasks can also be executed.



Task Configuration

Description	API to Set Configuration	API to Query Support
Enable the task	`doca_sha_task_partial_hash_set_conf`	`doca_sha_cap_task_partial_hash_get_supported`
Number of tasks	`doca_sha_task_partial_hash_set_conf`	`doca_sha_cap_get_max_num_tasks`
Maximal source buffer size	–	`doca_sha_cap_get_max_src_buf_size`
Maximum source buffer list size	–	`doca_sha_cap_get_max_list_buf_num_elem`
Minimum destination buffer size	–	`doca_sha_cap_get_min_dst_buf_size`
SHA block size		`doca_sha_cap_get_partial_hash_block_size`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Buffer pointing to the memory to be used for SHA calculation	Only the data residing in the data segment is to be used. And the data length for the non-last data segment must be multiple of the SHA block size queried by `doca_sha_cap_get_partial_hash_block_size`
Destination buffer	Buffer pointing to the memory is used for writing the SHA calculation result	The SHA result is appended to the tail segment. During the whole calculation process, this buffer cannot be modified.
SHA algorithm type	SHA algorithm to be used in SHA calculation	Must be one of `DOCA_SHA_ALGORITHM_SHA1`, `DOCA_SHA_ALGORITHM_SHA256`, `DOCA_SHA_ALGORITHM_SHA512`
Whether the current source buffer is the last segment	Indicate whether the current source Buffer is the last segment data to be used for partial-SHA calculation	Use `doca_sha_task_partial_hash_set_is_final_buf` to set this property
Set source buffer	Use to set the subsequent source segment buffer after the initial `doca_sha_task_partial_hash` task is allocated	`doca_sha_task_partial_hash_set_src`

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The SHA calculation of data from the source buffer is successfully completed and the result is written to the destination buffer

- The destination buffer data segment is extended to include the SHA result data

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects is not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task is submitted, the source and destination should not be read/written to
- Other limitations are described in DOCA Core Task

5.4.14.6.2 Events

DOCA SHA exposes asynchronous events to notify about changes that happen unexpectedly according to the DOCA Core architecture.

The only events SHA exposes are common events as described in DOCA Core Event.

5.4.14.7 State Machine

The DOCA SHA library follows the context state machine as described in DOCA Core Context State Machine.

The following section describes moving states and what is allowed in each state.

5.4.14.7.1 Idle

In this state, it is expected that the application either:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop after making sure all tasks have been freed
Stopping	Call progress until all tasks are completed and freed

5.4.14.7.2 Starting

This state cannot be reached.

5.4.14.7.3 Running

In this state, it is expected that the application:

- Allocates and submits tasks
- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocating previously configured task
- Submitting a task
- Calling stop

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

5.4.14.7.4 Stopping

In this state, it is expected that the application:

- Calls progress to complete all inflight tasks (tasks complete with failure)
- Frees any completed tasks

Allowed operations:

- Calling progress

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call progress and fatal error occurs
Running	Call stop without freeing all tasks

5.4.14.8 Alternative Datapath Options

DOCA SHA only supports datapath on the CPU. See Execution Phase for details.

5.4.14.9 DOCA SHA Samples

This section describes DOCA SHA samples based on the DOCA SHA library. The samples in this section illustrate how to use the DOCA SHA API to do the following:

- Calculate SHA of contents of a buffer, and write the result to another buffer

- Segment the the contents of a buffer, and do partial-SHA calculation of this collection of segments, then write the result to another buffer

> The DOCA samples described in this section are governed under the BSD-3 software license agreement.

5.4.14.9.1 Running the Samples

1. Refer to the following documents:
   - DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - DOCA Troubleshooting for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

```
cd /opt/mellanox/doca/samples/doca_sha/<sample_name>
meson/tmp/build
ninja -C/tmp/build
```

> The binary **doca_<sample_name>** is created under **/tmp/build/**.

3. The following sample displays **doca_sha_create** usage:

```
Usage: doca_sha_create [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -d, --data                        user data
```

4. For additional information per sample, use the **-h** option:

```
/tmp/build/doca_<sample_name>-h
```

5.4.14.9.2 Samples

> These samples are also available on GitHub.

5.4.14.9.2.1 SHA Create

This sample illustrates how to perform an SHA calculation with DOCA SHA.

The sample logic includes:

1. Locating DOCA device.

2. Initializing required DOCA Core structures.
3. Setting the **task_pool** configuration for **doca_sha_task_hash**.
4. Populating DOCA memory map with two relevant buffers.
5. Allocating element in DOCA buffer inventory for each buffer.
6. Allocating and initializing a **doca_sha_task_hash**.
7. Submitting the task.
8. Retrieving task result once it is done.

Reference:

- **/opt/mellanox/doca/samples/doca_sha/sha_create/sha_create_sample.c**
- **/opt/mellanox/doca/samples/doca_sha/sha_create/sha_create_main.c**
- **/opt/mellanox/doca/samples/doca_sha/sha_create/meson.build**

5.4.14.9.2.2 SHA-Partial Create

This sample illustrates how to perform a partial-SHA calculation for a collection of data segments with DOCA SHA.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the required DOCA Core structures.
3. Setting the **task_pool** configuration for **doca_sha_task_partial_hash**.
4. Chopping the source data into a collection of data segments according to the selected SHA algorithm's block size
5. Populating DOCA memory map with needed buffers for all source data segments and destination buffer.
6. Allocating element in DOCA buffer inventory for the first source buffer and destination buffer.
7. Allocating and initializing a **doca_sha_task_partial_hash** with the first source buffer and the destination buffer.
8. Iteratively repeating the following sub-steps until all data segments are consumed:
    a. Submitting the **doca_sha_task_partial_hash**.
    b. Waiting for the submitted task to finish.
    c. Allocating a **doca_buf** for the next source segment and use **doca_sha_task_partial_hash_set_src** to set it as source buffer of the above allocated task.
    d. If it is the final segment, use **doca_sha_task_partial_hash_set_is_final_buf** to mark it in the allocate task.
9. Retrieving the result of the final iteration in the destination buffer as the full partial-SHA calculation result.
10. Destroying all SHA and DOCA Core structures.

Reference:

- **/opt/mellanox/doca/samples/doca_sha/sha_partial_create/sha_partial_create_sample.c**
- **/opt/mellanox/doca/samples/doca_sha/sha_partial_create/sha_partial_create_main.c**
- **/opt/mellanox/doca/samples/doca_sha/sha_partial_create/meson.build**

5.4.15 DOCA Erasure Coding

This guide provides instructions on how to use the DOCA Erasure Coding API.

5.4.15.1 Introduction

> ⚠ The quality status of DOCA libraries is listed here.

The DOCA Erasure Coding (known also as forward error correction or FEC) library provides an API to encode and decode data using hardware acceleration, supporting both host and NVIDIA® BlueField®-3 (and higher) DPU memory regions.

DOCA Erasure Coding recovers lost data fragments by creating generic redundancy fragments (backup). Each redundancy block that the library creates can help recover any block in the original data should a total loss of fragment occur. This increases data redundancy and reduces data overhead.

The library provides an API for executing erasure coding (EC) operations on DOCA buffers residing in either the DPU or host memory.

This document is intended for software developers wishing to accelerate their application's EC memory operations.

5.4.15.1.1 Glossary

The following terms aid in the understanding of the information on this page:

Term	Definition
Data	Original data, original blocks, blocks of original data to be protected/preserved
Coding matrix	Coefficients, the matrix used to generate the redundancy blocks and recovery
Redundancy blocks	Codes; encoded data; the extra blocks that help recover data loss
Encoding	The process of creating the redundancy blocks. Encoded data is referred to as the original blocks or redundancy blocks.
Decoding	The process of recovering the data. Decoded data is referred to as the original blocks alone.

5.4.15.2 Prerequisites

DOCA Erasure Coding library follows the architecture of a DOCA Core Context, it is recommended read the following sections before:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

5.4.15.3 Changes From Previous Releases

5.4.15.3.1 Changes in 2.10.0

N/A

5.4.15.4 Environment

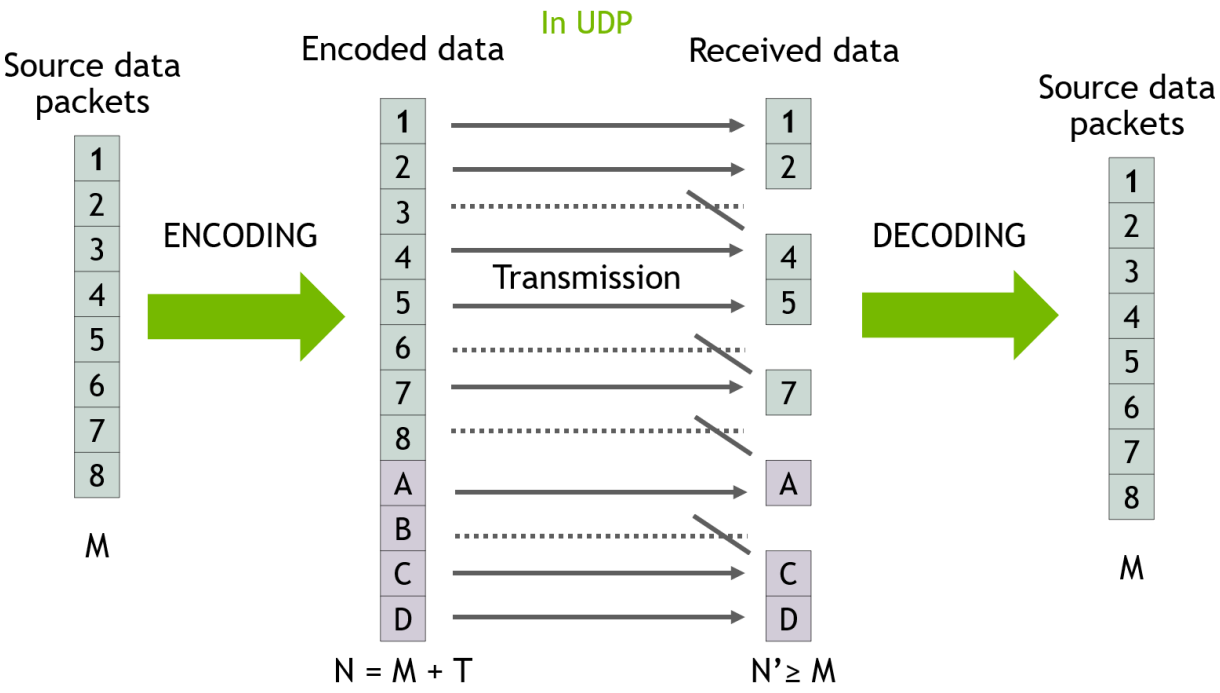
DOCA Erasure Coding-based applications can run either on the host machine or on the BlueField target (NVIDIA® BlueField®-3 and above).

Erasure Coding can be run with BlueField configured in either NIC or DPU mode (as described in BlueField Modes of Operation).

5.4.15.5 Architecture

DOCA Erasure Coding is a DOCA Context as defined by DOCA Core. This library leverages the DOCA Core architecture to expose asynchronous tasks/events that are offloaded to hardware.

The following diagram presents a high-level view of the EC transmission flow:



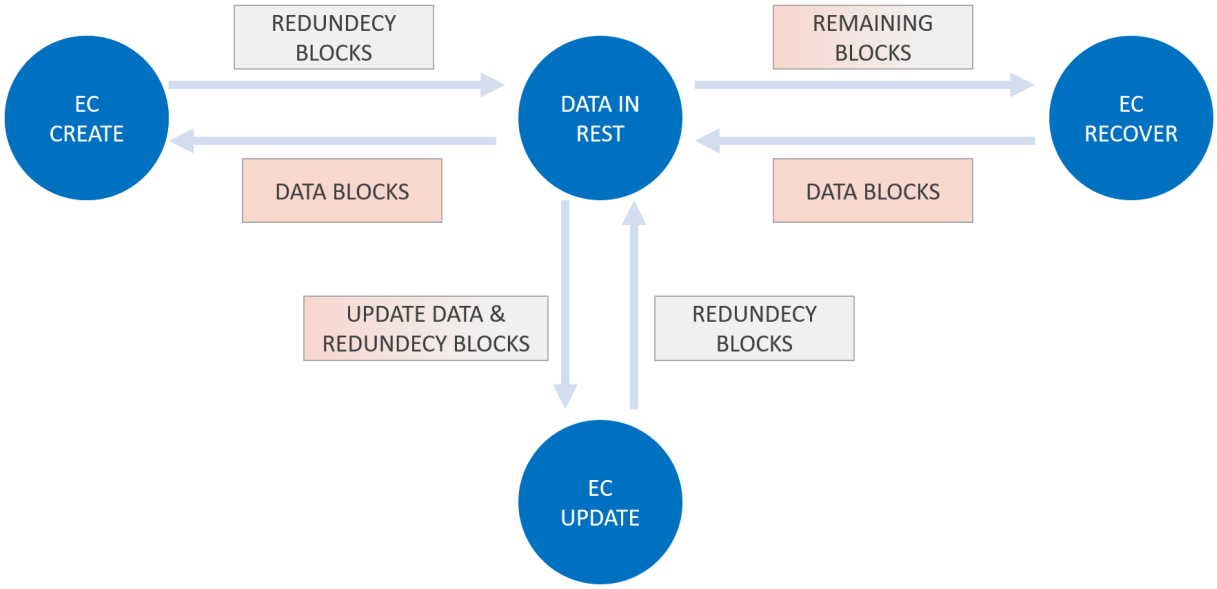
1. M packets are sent from the source (8 in this case).
2. Before the source send them, the source encode the data by adding to it T redundancy packets (4 in this case).
3. The packets are transmitted to the destination in UDP protocol. Some packets are lost and N' packets are received (in this case 4 packets are lost and 8 are received).

4. The destination decodes the data using all the packets available (both original data in green and redundancy data in red) and gets back the M original data packets.

5.4.15.5.1 Flows

Regular EC flow consists of the following elements:

1. Creating redundancy blocks from data (EC create).
2. Updating redundancy blocks from updated data (EC update).
3. Recovering data blocks from redundancy blocks (EC recover).



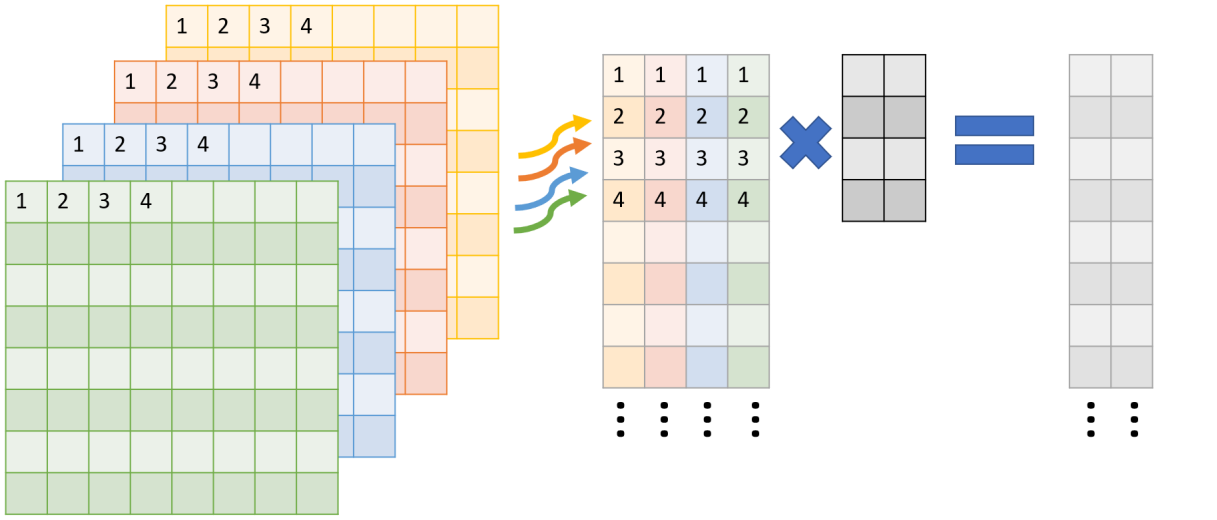
The following sections examine an M:K (where M is the original data and K is redundancy) EC.

5.4.15.5.2 Create Redundancy Blocks

The user must perform the following:

1. Input M data blocks via **doca_buf** (filled with data, each block size B).
2. Output K empty blocks via **doca_buf** (each block size B).
3. Use DOCA Erasure Coding to create a coding matrix of M by K via **doca_buf**.
4. Use DOCA Erasure Coding Create task to get the K output redundancy blocks.

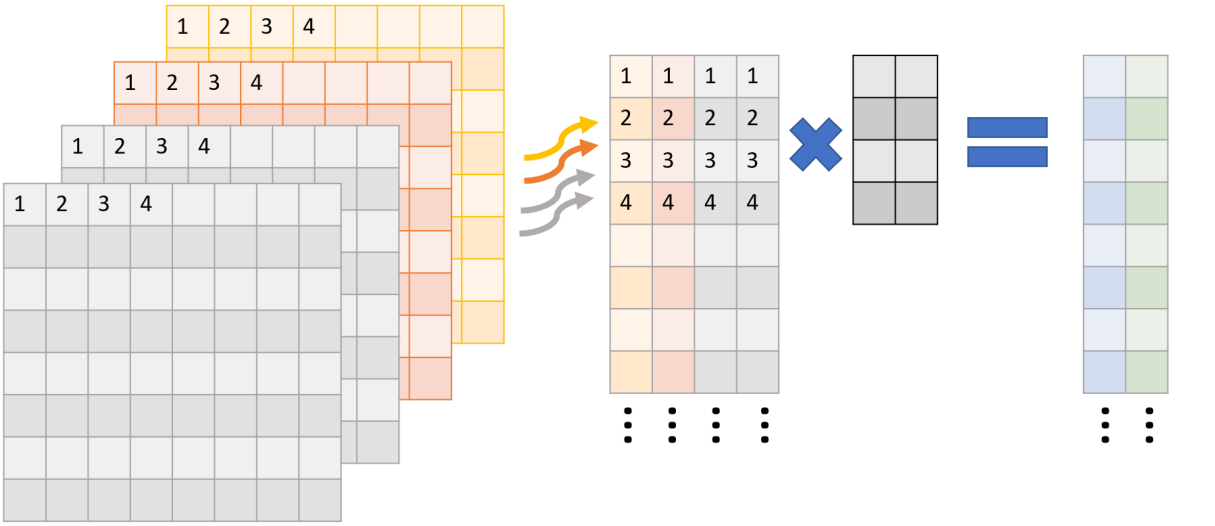
> ⚠ This step can be repeated in a stream use case, as the DPU would not be the recovery or update point.



5.4.15.5.3 Recover Block

The user must perform the following:

1. Input M-L original blocks via **doca_buf** (blocks that were not impaired).
2. Input L≤K (any) redundancy blocks via **doca_buf** (redundancy blocks originating from create/update tasks).
3. Input bitmask or array, indicating which blocks to recover.
4. Output L empty blocks via **doca_buf** (same size of data block).
5. Use DOCA Erasure Coding to create a recover coding matrix of M by L via **doca_buf** (unique per bitmask).
6. Use DOCA Erasure Coding Recover task to get the L output recovered data blocks.



5.4.15.5.4 Objects

5.4.15.5.4.1 Device and Device Representor

The DOCA Erasure Coding library requires a DOCA device to operate. The device is used to access memory and perform the encoding and decoding operations. See DOCA Core Device Discovery.

For same Bluefield card, it does not matter which device is used (PF/VF/SF), as all these devices utilize the same HW component. If there are multiple DPUs, then it is possible to create an EC instance per DPU, providing each instance with a device from a different DPU. To access memory that is not local (from the host to the DPU and vice versa), the DPU side of the application must pick a device with an appropriate representor. See DOCA Core Representor Device Discovery.

The device must stay valid until the EC instance is destroyed.

5.4.15.5.4.2 Memory Buffers

Executing any DOCA EC task requires two DOCA buffers, a source buffer and a destination buffer.

Depending on the allocation pattern of the buffers, refer to the Inventory Types table.

Buffers must not be modified or read during the execution of any task.

5.4.15.6 Configuration Phase

To start using the library, first, you need to go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context, to allow execution of tasks and retrieval of events.

5.4.15.6.1 Configurations

The context can be configured to match the application use case.

To find if a configuration is supported, or what the min/max value, please refer to Device Support.

5.4.15.6.1.1 Mandatory Configurations

These configurations are mandatory and must be set by the application before attempting to start the context:

- At least 1 task/event type needs to be configured. See configuration of Tasks.
- A device with appropriate support must be provided on creation.

5.4.15.6.2 Device Support

DOCA Erasure Coding needs a device to operate. For picking a device, see DOCA Core Device Discovery.

Erasure Coding can be used in BlueField-3 with some limitations (see architecture). Any device can be used PF/VF/SF.

As device capabilities may change in the future, it is recommended to choose your device using the following methods:

- **doca_ec_cap_task_galois_mul_is_supported**
- **doca_ec_cap_task_create_is_supported**
- **doca_ec_cap_task_update_is_supported**
- **doca_ec_cap_task_recover_is_supported**

Some devices can allow different capabilities as follows:

- The maximum buffer list length
- The maximum block size

> ⚠ Current BlueField-3 limitations:
>
> - Data block count range: 1-128
> - Redundancy block count: 1-32
> - Block size: 64B-128MB

5.4.15.6.3 Buffer Support

Tasks support buffers with the following features:

Buffer Type	Source Buffer	Destination Buffer
Linked list buffer	Depends on the device; check the **max_buf_list_len** capability	Depends on the device; check the **max_buf_list_len** capability
Local mmap buffer	Yes	Yes

Buffer Type	Source Buffer	Destination Buffer
Mmap from PCIe export buffer	Yes	Yes
Mmap from RDMA export buffer	No	No

5.4.15.7 Execution Phase

This section describes execution on CPU or DPU using the DOCA Core Progress Engine.

5.4.15.7.1 Matrix Generate

All tasks require a coding matrix.

5.4.15.7.1.1 Matrix Type

DOCA EC provides 2 matrix types which are elaborated on in the following subsections.

Cauchy

Cauchy encoding matrix is constructed so that

We don't have a way to export this macro.

Where:

- We don't have a way to export this macro.
- We don't have a way to export this macro.
- We don't have a way to export this macro.
- We don't have a way to export this macro.

Vandermonde

Vandermonde encoding matrix is constructed so that

We don't have a way to export this macro.

Where:

- We don't have a way to export this macro.

- We don't have a way to export this macro.

Vandermonde matrix does not guarantee that every submatrix is invertible (i.e., the decode task may fail in some settings).

5.4.15.7.1.2 Matrix Functionality

Create

An encoding matrix is necessary for executing the create task, to create redundancy blocks.

The matrices used for updates and recovery are based on an encoding matrix.

The following subsections describe the available options for creating matrices.

Generic

Generic creation, with the `doca_ec_matrix_create()` function, is used for simple setup using one of matrix types provided by the library.

Input:

Name	Description
Type	One of matrix types provided by the library
Data block count	The number of original data blocks
Redundancy block count	The number of redundancy blocks

Custom

Custom creation, with the `doca_ec_matrix_create_from_raw()` function, is used if the desired type of matrix is not provided by the library.

Input:

Name	Description	Notes
Data	The data of a coding matrix	The size of the data should be `data_block_count` * `rdnc_block_count`
Data block count	The number of original data blocks	–
Redundancy block count	The number of redundancy blocks	–

Update

This matrix is necessary for executing the update task, to update the redundancy blocks after a change in the data blocks.

The matrix is created using the `doca_ec_matrix_create_update()` function.

Input:

Name	Description	Notes
Coding matrix	A coding matrix created by `doca_ec_matrix_create()` or `doca_ec_matrix_create_from_raw()`	–
Update indices	An array specifying the indices of the updated data blocks	• The indices must be in ascending order • The indices should match the order of the data blocks in the matrix creation function
Number of updates	The number of updated blocks. The length of the update indices array.	–

Recover

This matrix is necessary for executing the recover task, to recover original data blocks.

The matrix is created using the `doca_ec_matrix_create_recover()` function.

Input:

Name	Description	Notes
Coding matrix	A coding matrix created by `doca_ec_matrix_create()` or `doca_ec_matrix_create_from_raw()`	–
Missing indices	An array specifying the indices of the missing data blocks	• The indices must be in ascending order • The indices should match the order of the data blocks in the matrix creation function
Number of missing	The number of updated blocks. The length of the update indices array.	–

5.4.15.7.2 Tasks

5.4.15.7.2.1 Task Batching

DOCA Erasure Coding supports task batching mode, which is a task submit mode of work that allows aggregating multiple DOCA tasks of the same type and handling them as a single unit.

> For more information on task batching, refer to DOCA Core Task.

DOCA Erasure Coding supports the flags `DOCA_TASK_SUBMIT_FLAG_NONE`, `DOCA_TASK_SUBMIT_FLAG_FLUSH` and `DOCA_TASK_SUBMIT_FLAG_OPTIMIZE_REPORTS`.

5.4.15.7.2.2 Galois Mul Task

This task executes Galois multiplication between the original blocks and the coding matrix.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_ec_task_galois_mul_set_conf`	`doca_ec_cap_task_galois_mul_is_supported`
Maximum block size	–	`doca_ec_cap_get_max_block_size`
Maximum buffer list length	–	`doca_ec_cap_get_max_buf_list_len`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
coding matrix	A coding matrix as created by `doca_ec_matrix_create()` or `doca_ec_matrix_create_from_raw()`	–
source buffer	Source original data buffer, holding a sequence containing all original blocks (e.g., `block_1`, `block_2`, etc.); the order matters	• The data length of `src_buf` should be a multiplication of the block size • The data length should also be aligned to 64B and with a minimum size of 64B
destination buffer	A destination buffer for the multiplication outcome blocks. The sequence containing all multiplication outcome blocks (`dst_block_1`, `dst_block_2`, etc.) is written to it upon successful completion of the task.	• The data is written to the tail segment extending the data segment • The minimal available memory in `dst_buf` should be the number of redundancy blocks * the block size, aligned to 64B and, in any case, at least 64B.

Example for required buffer length

If a Galois multiplication task matrix is 10x4 (i.e., 10 original blocks, 4 multiplication outcome blocks), and the block size is 64KB:

- `src_buf` data length should be 10x64KB = 640KB
- The available memory for writing in `dst_buf` should be at least 4x64KB = 256KB

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The destination buffer holds a sequence containing all multiplication outcome blocks (e.g., `dst_block_1` , `dst_block_2` , etc.)
- The destination buffer data segment is extended to include the outcome blocks

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the source and destination buffer should not be read from/written to
- Source and destination buffers must not overlap
- Other limitations are described in DOCA Core Task

5.4.15.7.2.3 Create Task

This task creates redundancy blocks for the given original data blocks using a given coding matrix.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_ec_task_create_set_conf`	`doca_ec_cap_task_create_is_supported`
Maximum block size	–	`doca_ec_cap_get_max_block_size`
Maximum buffer list length	–	`doca_ec_cap_get_max_buf_list_len`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
coding matrix	A coding matrix created by `doca_ec_matrix_create()` or `doca_ec_matrix_create_from_raw()`	–
original data blocks	Source original data buffer, holding a sequence containing all original blocks (`block_1` , `block_2` , etc.); the order matters	• The data length of `original_data_blocks` should be a multiplication of the block size • The data length should also be aligned to 64B and with a minimum size of 64B

Name	Description	Notes
redundancy blocks	A destination buffer for the redundancy blocks. The sequence containing all redundancy blocks (**rdnc_block_1** , **rdnc_block_2** , etc.) is written to it upon successful completion of the task.	• The data will be written to the tail segment extending the data segment • The minimal available memory in **rdnc_blocks** should be the number of redundancy blocks * the block size, aligned to 64B and, in any case, at least 64B

Example for required buffer lengths

If a create task matrix is 10x4 (i.e., 10 original blocks, 4 redundancy blocks), and the block size is 64KB:

- **original_data_blocks** data length should be 10x64KB = 640KB
- The available memory for writing in **redundancy_blocks** should be at least 4x64KB = 256KB

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The destination buffer holds a sequence containing all redundancy blocks (**rdnc_block_1** , **rdnc_block_2** , etc.)
- The destination buffer data segment is extended to include the redundancy blocks

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination **doca_buf** objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task is submitted, the source and destination buffers should not be read from/written to
- Source and destination buffers must not overlap
- Other limitations are described in DOCA Core Task

5.4.15.7.2.4 Update Task

This task executes updates the redundancy blocks for the given original data blocks, using an update coding matrix.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_ec_task_update_set_conf`	`doca_ec_cap_task_update_is_supported`
Maximum block size	–	`doca_ec_cap_get_max_block_size`
Maximum buffer list length	–	`doca_ec_cap_get_max_buf_list_len`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
update matrix	An update coding matrix created by `doca_ec_matrix_create_update()` or `doca_ec_matrix_create_from_raw()`	-
original updated and RDNC blocks	A source buffer with data, holding a sequence containing the original data block and its updated data block, for each block that was updated, followed by the old redundancy blocks (`old_data_block_i` , `updated_data_block_i` , `old_data_block_j` , `updated_data_block_j` , ..., `rdnc_block_1` , `rdnc_block_2` , etc.)	• The data length of `original_updated_and_rdnc_blocks` should be a multiplication of the block size • The data length should also be aligned to 64B and with a minimum size of 64B
updated RDNC blocks	A destination buffer for the updated redundancy blocks. The sequence containing the updated redundancy blocks (`rdnc_block_1` , `rdnc_block_2` , etc.) is written to it upon successful completion of the task	• The data is written to the tail segment extending the data segment • The minimal available memory in `updated_rdnc_blocks` should be the number of redundancy blocks * the block size, aligned to 64B and, in any case, at least 64B

Example for required buffer lengths

using an update task matrix, in which 3 data block were updated and there are 4 redundancy blocks, and the block size is 64KB:

- `original_updated_and_rdnc_blocks` data length should be (3+3+4=10)x64KB = 640KB
- The available memory for writing in `updated_rdnc_blocks` should be at least 4x64KB = 256KB

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The destination buffer holds a sequence containing the updated redundancy blocks (`rdnc_block_1`, `rdnc_block_2`, etc.)
- The destination buffer data segment is extended to include the updated redundancy blocks

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects is not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, the source and destination buffers should not be read from/written to
- Source and destination buffers must not overlap
- Other limitations described in DOCA Core Task

5.4.15.7.2.5 Recover Task

This task executes recovers data blocks for, using given available original data blocks and redundancy blocks and a given coding matrix.

Task Configuration

Description	API to Set the Configuration	API to Query Support
Enable the task	`doca_ec_task_recover_set_conf`	`doca_ec_cap_task_recover_is_supported`
Maximum block size	–	`doca_ec_cap_get_max_block_size`
Maximum buffer list length	–	`doca_ec_cap_get_max_buf_list_len`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
recover matrix	A coding matrix create by `doca_ec_matrix_create()` or `doca_ec_matrix_create_from_raw()`	–

Name	Description	Notes
available blocks	A source buffer with data, holding a sequence containing available data blocks and redundancy blocks (`data_block_a`, `data_block_b`, `data_block_c`, ..., `rdnc_block_x`, `rdnc_block_y`, etc.)	• The total number of blocks given should be equal to the number of original data blocks • The data length of `available_blocks` should be a multiplication of the block size • The data length should also be aligned to 64B and with a minimum size of 64B
recovered data blocks	A destination buffer for the recovered data blocks. The sequence containing the recovered data blocks (`data_block_i`, `data_block_j`, etc.) is written to it upon successful completion of the task	• The data is written to the tail segment extending the data segment • The minimal available memory in `recovered_data_blocks` should be the number of missing data blocks * the block size, aligned to 64B and, in any case, at least 64B.

Example for required buffer lengths

Using a recover task matrix, based on an original 10x4 coding matrix (i.e., 10 original blocks, 4 redundancy blocks), and a block size of 64KB:

- 10 available blocks should be given in total (e.g., 7 data blocks and 3 redundancy blocks)
- `available_blocks` data length should be 10x64KB = 640KB
- The available memory for writing in `recovered_data_blocks` should be at least 3x64KB = 192KB

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task is completed successfully the data is transformed to destination.

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task is submitted, the source and destination buffers should not be read from/written to
- Source and destination must not overlap

- The number of blocks that can be recovered is limited to the number of redundancy blocks created
- Other limitations are described in DOCA Core Task

..

5.4.15.8 DOCA Erasure Coding Samples

This section provides DOCA Erasure Coding sample implementation on top of the BlueField-3 DPU (and higher).

> All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

5.4.15.8.1 Sample Prerequisites

N/A

5.4.15.8.2 Running the Sample

1. Refer to the following documents:
   - DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - DOCA Troubleshooting for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

```
cd /opt/mellanox/doca/samples/doca_erasure_coding/<sample_name>
meson /tmp/build
ninja -C /tmp/build
```

> The binary **doca_<sample_name>** is created under **/tmp/build/**.

3. Sample (e.g., **doca_erasure_coding_recover**) usage:

```
Usage: doca_erasure_coding_recover [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -p, --pci-addr                    DOCA device PCI device address - default: 03:00.0
  -i, --input                       Input file/folder to ec - default: self
  -o, --output                      Output file/folder to ec - default: /tmp
  -b, --both                        Do both (encode & decode) - default: false
  -x, --matrix                      Matrix - {cauchy, vandermonde} - default: cauchy
  -t, --data                        Data block count - default: 2
  -r, --rdnc                        Redundancy block count - default: 2
  -d, --delete-index                Indices of data blocks to delete; comma-separated (i.e., 0,3,4) -
default: 0
  -ns, --num-src-buf                Number of doca_buf for source buffer
  -nd, --num-dst-buf                Number of doca_buf for destination buffer
```

Current BlueField-3 limitations:

- Data block count range – 1-128
- Redundancy block count – 1-32
- Block size – 64B-128MB
- The command line option "--num-src-buf" and "--num-dst-buf" are used to show linked-list usage of doca-buf. The maximum supported number of elements is 64.

4. For additional information per sample, use the `-h` option:

```
/tmp/build/doca_<sample_name> -h
```

5.4.15.8.3 Samples

These samples are also available on GitHub.

5.4.15.8.3.1 Erasure Coding Recover

This sample illustrates how to use DOCA Erasure Coding (EC) library to encode and decode a file block (and entire file).

The sample logic includes 3 steps:

1. Encoding – create redundancy.
2. Deleting – simulating disaster.
3. Decoding – recovering data.

The encode logic includes:

1. Locating a DOCA device.
2. Initializing the required DOCA Core structures, such as the progress engine (PE), memory maps, and buffer inventory.
3. Reading source original data file and splitting it to a specified number of blocks, **<data block count>**, specified for the sample to the output directory.
4. Populating two DOCA memory maps with a memory range, one for the source data and one for the result.
5. Allocating buffers from DOCA buffer inventory for each memory range.
6. Creating an EC object.
7. Connecting the EC context to the PE.
8. Setting a state change callback function for the PE, with the following logic:
    - Printing a log with every state change
    - Indicating that the user may stop progress the PE once it is back in idle state
9. Setting the configuration to the EC create task, including setting callback functions as follows:
    - Successful completion callback:
        i. Writing the resulting redundancy blocks to the output directory (count is specified by **<redundancy block count>**).

ii. Freeing the task.
iii. Saving the result of the task and the callback. If there was an error in step a., the relevant error value is saved.
iv. Stopping the context.

- Failed completion callback:
  i. Saving the result of the task and the callback.
  ii. Freeing the task.
  iii. Stopping the context.

10. Creating EC encoding matrix by the matrix type specified to the sample.
11. Allocating and submitting an EC create task.
12. Progressing the PE until the context returns to idle state, either as a result of a successful run in which all tasks have been successfully completed, or as a result of a fatal error.
13. Destroying all EC and DOCA Core structures.

The delete logic includes:

1. Deleting the block files specified with **<indices of data blocks to delete>**.

The decode logic includes:

1. Locating a DOCA device.
2. Initializing the required DOCA Core structures, such as the PE, memory maps, and buffer inventory.
3. Reading the output directory (source remaining data) and determining the block size and which blocks are missing (needing recovery).
4. Populating two DOCA memory maps with a memory range, one for the source data and one for the result.
5. Allocating buffers from DOCA buffer inventory for each memory range.
6. Creating an EC object.
7. Connecting the EC context to the PE.
8. Setting a state change callback function for the PE, with the following logic:
   - Printing a log with every state change
   - Indicating that the user may stop progress the PE once it is back in idle state
9. Setting the configuration to the EC recover task, including setting callback functions as following:
   - Successful completion callback:
     i. Writing the resulting recovered blocks to the output directory.
     ii. Writing the recovered file to the output path.
     iii. Freeing the task.
     iv. Saving the result of the task and the callback. If there was an error in step a., the relevant error value is saved.
     v. Stopping the context.
   - Failed completion callback:
     i. Saving the result of the task and the callback.
     ii. Freeing the task.
     iii. Stopping the context.
10. Creating EC encoding matrix by the matrix type specified to the sample.
11. Creating EC decoding matrix, with **doca_ec_matrix_create_recover()**, using the encoding matrix.

12. Allocating and submitting an EC recover task.
13. Progressing the PE until the context returns to idle state, either as a result of a successful run in which all tasks have been successfully completed, or as a result of a fatal error.
14. Destroying all DOCA EC and DOCA Core structures.

References:

- **/opt/mellanox/doca/samples/doca_erasure_coding/doca_erasure_coding_recover/erasure_coding_recover_sample.c**
- **/opt/mellanox/doca/samples/doca_erasure_coding/doca_erasure_coding_recover/erasure_coding_recover_main.c**
- **/opt/mellanox/doca/samples/doca_erasure_coding/doca_erasure_coding_recover/meson.build**

5.4.16 DOCA AES-GCM

This guide provides instructions on building and developing applications that require data encryption and decryption using the AES-GCM algorithm.

5.4.16.1 Introduction

The quality status of DOCA libraries is listed here.

The DOCA AES-GCM library is supported at alpha level and on NVIDIA BlueField-3 devices or higher.

The library provides an API for executing AES-GCM operations on DOCA buffers, where the buffers reside in either local memory (i.e., within the same host) or host memory accessible by the DPU (remote memory). Use DOCA AES-GCM to execute complex encrypt/decrypt operations in an optimized, hardware-accelerated manner.

5.4.16.2 Prerequisites

This library follows the architecture of a DOCA Core context, it is recommended to read the following sections before:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

5.4.16.3 Changes From Previous Releases

5.4.16.3.1 Changes in 2.9.0

N/A

5.4.16.4 Environment

DOCA AES-GCM-based applications can run either on the host machine or DPU target.

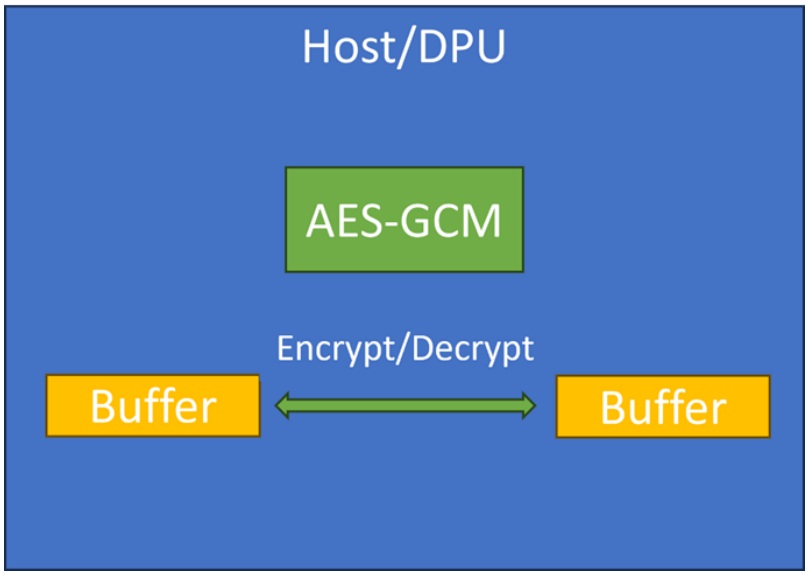
Encrypting/decrypting from the host to DPU and vice versa can only be run when the DPU is configured in DPU mode.

5.4.16.5 Architecture

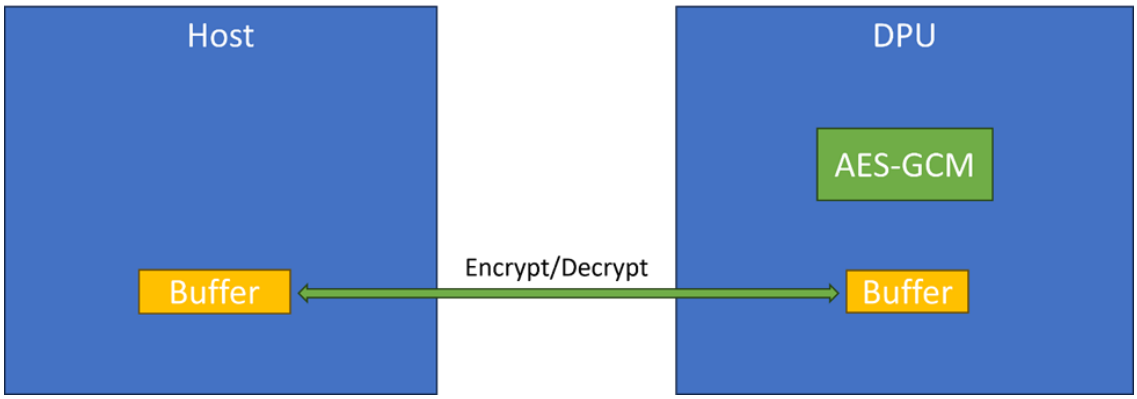
DOCA AES-GCM is a DOCA Core Context. This library leverages the DOCA Core architecture to expose asynchronous tasks/events that are offloaded to hardware.

AES-GCM can be used to encrypt/decrypt data as illustrated in the following diagrams:

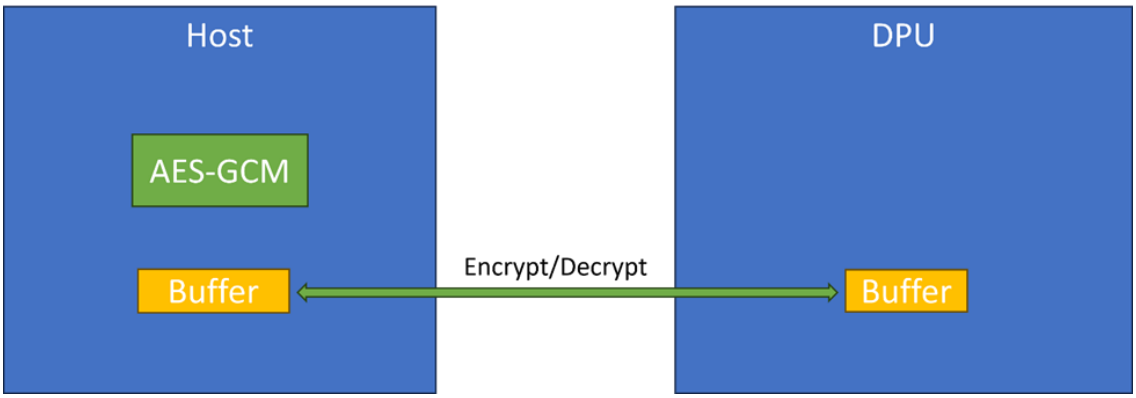
- Encrypt/decrypt from local memory to local memory:



- Using the DPU to copy memory between the host and the DPU:



- Using the host to copy memory between the host and the DPU:



5.4.16.5.1 Objects

5.4.16.5.1.1 Device and Representor

The library requires a DOCA device to operate. The device is used to access memory and perform the actual encrypt/decrypt. See DOCA Core Device Discovery.

For the same BlueField DPU, it does not matter which device is used (i.e., PF/VF/SF) as all these devices utilize the same hardware component. If there are multiple DPUs, then it is possible to create a AES-GCM instance per DPU, providing each instance with a device from a different DPU.

To access memory that is not local (i.e., from the host to DPU or vice versa), the DPU side of the application must pick a device with an appropriate representor (see DOCA Core Device Representor Discovery). The device must stay valid as long as AES-GCM instance is not destroyed.

5.4.16.5.1.2 Memory Buffers

The encrypt/decrypt task requires two DOCA buffers containing the destination and the source.

Depending on the allocation pattern of the buffers, consider the DOCA Core Inventory Types table.

To find what kind of memory is supported, refer to the buffer support table.

⚠ Buffers must not be modified or read during the encrypt/decrypt operation.

5.4.16.6 Configuration Phase

To start using the library users must go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context to allow execution of tasks and retrieval of events.

5.4.16.6.1 Configurations

The context can be configured to match the application use case.

To find if a configuration is supported or its min/max value, refer to Device Support.

5.4.16.6.1.1 Mandatory Configurations

These configurations must be set by the application before attempting to start the context:

- At least one task/event type must be configured. See configuration of Tasks and/or Events.
- A device with appropriate support must be provided upon creation

5.4.16.6.2 Device Support

DOCA AES-GCM requires a device to operate. For picking a device, see DOCA Core Device Discovery.

As device capabilities may change in the future (see DOCA Core Device Support) it is recommended to select your device using the following method:

- `doca_aes_gcm_cap_task_encrypt_is_supported`
- `doca_aes_gcm_cap_task_decrypt_is_supported`

Some devices can allow different capabilities as follows:

- The maximum number of tasks
- The maximum buffer size
- The maximum supported number of elements in DOCA linked-list buffer
- The maximum initialization vector length
- Check if authentication tag of size 96-bit is supported
- Check if authentication tag of size 128-bit is supported
- Check if a given AES-GCM key type is supported

5.4.16.6.3 Buffer Support

Tasks support buffers with the following features:

Buffer Type	Source Buffer	Destination Buffer
Local mmap buffer	Yes	Yes
Mmap from PCIe export buffer	Yes	Yes
Mmap from RDMA export buffer	No	No
Linked list buffer	Yes	Yes

5.4.16.7 Execution Phase

This section describes execution on the CPU using DOCA Core Progress Engine.

5.4.16.7.1 Tasks

DOCA AES-GCM exposes asynchronous tasks that leverage DPU hardware according to the DOCA Core architecture.

5.4.16.7.1.1 Encrypt Task

The encrypt task allows data encryption using buffers as described in Buffer Support.

Task Configuration

Description	API to Set Configuration	API to Query Support
Enable the task	`doca_aes_gcm_task_encrypt_set_conf`	`doca_aes_gcm_cap_task_encrypt_is_supported`
Number of tasks	`doca_aes_gcm_task_encrypt_set_conf`	`doca_aes_gcm_cap_get_max_num_tasks`
Maximal buffer size	–	`doca_aes_gcm_cap_task_encrypt_get_max_buf_size`
Maximum buffer list size	–	`doca_aes_gcm_cap_task_encrypt_get_max_list_buf_num_elem`
Maximum initialization vector length	–	`doca_aes_gcm_cap_task_encrypt_get_max_iv_length`
Enable authentication tag size	–	`doca_aes_gcm_cap_task_encrypt_is_tag_96_supported` `doca_aes_gcm_cap_task_encrypt_is_tag_128_supported`
Enable key type	–	`doca_aes_gcm_cap_task_encrypt_is_key_type_supported`

Task Input

Common input as described in DOCA Core Task

Name	Description	Notes
source buffer	Buffer pointing to the memory to be encrypted	Only the data residing in the data segment is encrypted
destination buffer	Buffer pointing to where memory is encrypted to	The encrypted data is appended to the tail segment
key	Key to encrypt the data	Created by the function `doca_aes_gcm_key_create` Users should use the same key to encrypt and decrypt the data
initialization vector (IV)	Initialization vector to be used by the AES-GCM algorithm	Users should use the same IV to encrypt and decrypt the data
initialization vector length	Initialization vector length that must be supplied for the AES-GCM algorithm	Represented in bytes, 0B-12B values are supported
authentication tag size	Authentication tag size to be supplied for the AES-GCM algorithm. The tag is automatically calculated and appended to the result buffer.	Represented in bytes, only 12B and 16B values are supported

Name	Description	Notes
additional authenticated data size	Additional authenticated data size to be supplied for the AES-GCM algorithm. This data, which should be present at the beginning of the source buffer, is will not encrypted but is authenticated.	Represented in bytes

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The data from the source buffer is encrypted and written to the destination buffer
- The destination buffer data segment is extended to include the encrypted data

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination **doca_buf** objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task is submitted, the source and destination should not be read/written to
- Other limitations are described in DOCA Core Task

5.4.16.7.1.2 Decrypt Task

The decrypt task allows data decryption. Using buffers as described in Buffer Support.

Task Configuration

Description	API to Set Configuration	API to Query Support
Enable the task	**doca_aes_gcm_task_decrypt_set_conf**	**doca_aes_gcm_cap_task_decrypt_is_supported**
Number of tasks	**doca_aes_gcm_task_decrypt_set_conf**	**doca_aes_gcm_cap_get_max_num_tasks**
Maximal buffer size	–	**doca_aes_gcm_cap_task_decrypt_get_max_buf_size**
Maximum buffer list size	–	**doca_aes_gcm_cap_task_decrypt_get_max_list_buf_num_elem**
Maximum initialization vector length	–	**doca_aes_gcm_cap_task_decrypt_get_max_iv_length**

Description	API to Set Configuration	API to Query Support
Enable authentication tag size	–	`doca_aes_gcm_cap_task_decrypt_is_tag_96_supported` `doca_aes_gcm_cap_task_decrypt_is_tag_128_supported`
Enable key type	–	`doca_aes_gcm_cap_task_decrypt_is_key_type_supported`

Task Input

Common input as described in DOCA Core Task.

Name	Description	Notes
Source buffer	Buffer pointing to the memory to be decrypted	Only the data residing in the data segment is decrypted
Destination buffer	Buffer pointing to where memory is decrypted to	The decrypted data is appended to the tail segment extending the data segment
Key	Key to decrypt the data	Created by the function `doca_aes_gcm_key_create` The user should use the same key to encrypt and decrypt the data
Initialization vector (IV)	Initialization vector to be used by the AES-GCM algorithm	The user should use the same IV to encrypt and decrypt the data
Initialization vector length	Initialization vector length that must be supplied for the AES-GCM algorithm	Represented in bytes, 0B-12B values are supported
Authentication tag size	Authentication tag size to be supplied for the AES-GCM algorithm. The tag, present at the end of the source buffer, is verified and is not present in the destination buffer.	Represented in bytes, only 12B and 16B values are supported
Additional authenticated data size	Additional authenticated data size to be supplied for the AES-GCM algorithm. This data, present at the beginning of the source buffer, is not encrypted but is authenticated.	Represented in bytes

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The data from the source buffer is decrypted and written to the destination buffer
- The destination buffer data segment is extended to include the decrypted data

Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs

- The source and destination **doca_buf** objects is not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task is submitted, the source and destination should not be read/written to
- Other limitations are described in DOCA Core Task

5.4.16.7.2 Events

DOCA AES-GCM exposes asynchronous events to notify about changes that happen unexpectedly according to the DOCA Core architecture.

The only events AES-GCM exposes are common events as described in DOCA Core Event.

5.4.16.8 State Machine

The DOCA AES-GCM library follows the Context state machine as described in DOCA Core Context State Machine.

The following section describes moving states and what is allowed in each state.

5.4.16.8.1 Idle

In this state, it is expected that the application either:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to Configurations
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop after making sure all tasks have been freed
Stopping	Call progress until all tasks are completed and freed

5.4.16.8.2 Starting

This state cannot be reached.

5.4.16.8.3 Running

In this state, it is expected that the application:

- Allocates and submits tasks

- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocating previously configured task
- Submitting a task
- Calling stop

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

5.4.16.8.4 Stopping

In this state, it is expected that the application:

- Calls progress to complete all inflight tasks (tasks complete with failure)
- Frees any completed tasks

Allowed operations:

- Calling progress

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call progress and fatal error occurs
Running	Call stop without freeing all tasks

5.4.16.9 Alternative Datapath Options

DOCA AES-GCM only supports datapath on the CPU. See Execution Phase.

5.4.16.10 DOCA AES-GCM Samples

This section describes DOCA AES-GCM samples based on the DOCA AES-GCM library.

The samples in this section illustrate how to use the DOCA AES-GCM API to do the following:

- Encrypt contents of a buffer to another buffer
- Decrypt contents of a buffer to another buffer

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

5.4.16.10.1 Running the Samples

1. Refer to the following documents:

- DOCA Installation Guide for Linux for details on how to install BlueField-related software.
- DOCA Troubleshooting for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

```
cd /opt/mellanox/doca/samples/doca_aes_gcm/<sample_name>
meson/tmp/build
ninja -C/tmp/build
```

> The binary **doca_<sample_name>** is created under **/tmp/build/**.

3. Sample (e.g., **doca_aes_gcm_encrypt**) usage:

```
Usage: doca_aes_gcm_encrypt [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -p, --pci-addr                    DOCA device PCI device address - default: 03:00.0
  -f, --file                        Input file to encrypt/decrypt
  -o, --output                      Output file - default: /tmp/out.txt
  -k, --key                         Raw key to encrypt/decrypt with, represented in hex format (32
 characters for 128-bit key, and 64 for 256-bit key) - default: 256-bit key, equals to zero
  -i, --iv                          Initialization vector, represented in hex format (0-24 characters for
0-96-bit IV) - default: 96-bit IV, equals to zero
  -t, --tag size                    Authentication tag size. Tag size is in bytes and can be 12B or 16B -
default: 12
  -a, --aad size                    Additional authenticated data size - default: 0
  -ns, --num-src-buf                Number of doca_buf for source buffer
  -nd, --num-dst-buf                Number of doca_buf for destination buffer
```

4. For additional information per sample, use the **-h** option:

```
/tmp/build/doca_<sample_name>-h
```

> The command line options **--num-src-buf** and **--num-dst-buf** show linked-list usage of **doca-buf**. The maximum supported number of elements is 64.

5.4.16.10.2 Samples

> These samples are also available on GitHub.

5.4.16.10.2.1 AES-GCM Encrypt

This sample illustrates how to encrypt data with AES-GCM.

The sample logic includes:

1. Locating DOCA device.
2. Initializing required DOCA Core structures.
3. Setting the AES-GCM encrypt tasks configuration.
4. Populating DOCA memory map with two relevant buffers.
5. Allocating element in DOCA buffer inventory for each buffer.
6. Creating DOCA AES-GCM key.
7. Allocating and initializing AES-GCM encrypt task.
8. Submitting AES-GCM encrypt task.
9. Retrieving AES-GCM encrypt task once it is done.
10. Checking task result.
11. Destroying all AES-GCM and DOCA Core structures.

Reference:

- **/opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_encrypt/aes_gcm_encrypt_sample.c**
- **/opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_encrypt/aes_gcm_encrypt_main.c**
- **/opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_encrypt/meson.build**

5.4.16.10.2.2 AES-GCM Decrypt

This sample illustrates how to decrypt data with AES-GCM.

The sample logic includes:

1. Locating DOCA device.
2. Initializing needed DOCA Core structures.
3. Setting the AES-GCM decrypt tasks configuration.
4. Populating DOCA memory map with two relevant buffers.
5. Allocating element in DOCA buffer inventory for each buffer.
6. Creating DOCA AES-GCM key.
7. Allocating and initializing AES-GCM decrypt task.
8. Submitting AES-GCM decrypt task.
9. Retrieving AES-GCM decrypt task once it is done.
10. Checking task result.
11. Destroying all AES-GCM and DOCA Core structures.

Reference:

- **/opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_decrypt/aes_gcm_decrypt_sample.c**
- **/opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_decrypt/aes_gcm_decrypt_main.c**
- **/opt/mellanox/doca/samples/doca_aes_gcm/aes_gcm_decrypt/meson.build**

5.4.17 DOCA Rivermax

This guide provides instructions on building and developing applications that require media/data streaming.

5.4.17.1 Introduction

> The quality status of DOCA libraries is listed here.

DOCA Rivermax (RMAX) is the DOCA API for NVIDIA® Rivermax®, an optimized networking SDK designed specifically for media and data streaming applications. Rivermax leverages the hardware streaming acceleration technology within NVIDIA® BlueField® DPUs. This technology enables direct data transfers to and from the GPU, delivering best-in-class throughput and latency while minimizing CPU utilization for streaming workloads.

Key library features include the use of hardware-accelerated packet placement and header/payload splitting.

This document is intended for software developers who wish to accelerate their networking operations.

5.4.17.2 Prerequisites

This library follows the architecture of DOCA Core Context. It is recommended to read the following content before proceeding:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem
- Rivermax library User Manual

5.4.17.3 Environment

5.4.17.3.1 System Requirements

- DOCA Rivermax applications run only on the target DPU.
- A valid Rivermax library license is required. Refer to the NVIDIA Rivermax SDK page to obtain a license.
- Ensure an IP address is configured on the network interface being used.

5.4.17.3.2 Performance Configuration

To achieve maximum performance, it is recommended to enable at least 800 huge pages (2 MB each).

```
dpu> echo 1000000000 > /proc/sys/kernel/shmmax
dpu> echo 800 > /proc/sys/vm/nr_hugepages
```

5.4.17.3.3 User Privileges

By default, DOCA Rivermax applications must run with root privileges.

To execute the application as a regular user, you must perform the following configuration steps:

1. Grant the required Linux capabilities (**cap_net_raw** and **cap_sys_nice**) to the application binary.

```
# Set capabilities
sudo setcap cap_net_raw,cap_sys_nice=ep /path/to/application

# Verify
getcap /path/to/application
```

2. The application requires read access to the network interface's Virtual Product Data (VPD) file.

```
# Set permissions
sudo chmod 644 /sys/class/net/${INTERFACE}/device/vpd

# Verify
ls -l /sys/class/net/${INTERFACE}/device/vpd
```

Permissions on the **/sys** filesystem reset on every reboot. You must re-apply this **chmod** command after a system restart.

5.4.17.4 Linux Kernel Capabilities Requirement

To use the DOCA Rivermax library without root privileges, the following capabilities are required

Capability	Description
CAP_NET_RAW	Required for raw socket operations
CAP_SYS_NICE	Required for real-time scheduling and priority adjustments

5.4.17.5 Architecture

- DOCA Rivermax Input Stream is a DOCA Context as defined by DOCA Core
- DOCA Rivermax leverages DOCA Core architecture to expose asynchronous events that are offloaded to hardware
- DOCA Rivermax can be used to define input streams that allow packet acquisition on an IP port. Furthermore, the input stream can be split to TCP/UDP 5-tuples to allow separate handling of flows.

5.4.17.5.1 Objects

- **doca_rmax_flow** – is a flow object that represents an IP/port tuple
- **doca_rmax_in_stream** – is a **doca_ctx** that represents the input stream and can be thought of as a receive queue which scatters the received data into memory. Each stream can receive one or more flows.

5.4.17.6 Library Features

5.4.17.6.1 Header/Payload Split

The library allows incoming packets to be split into separate header and payload segments (memory blocks). It can then direct each segment into its corresponding destination buffer.

5.4.17.6.2 Hardware-Accelerated Packet Placement

This feature uses hardware acceleration to place incoming packets directly into the receive buffer according to their sequence numbers. This saves significant processing time that would otherwise be spent on software-based data reordering.

The library supports two types of sequence numbers for placement:

- RTP: The 16-bit sequence number from a standard RTP packet header (RFC 3550).
- ST 2110-20: The 32-bit sequence number used in SMPTE ST 2110-20 video streams.

> **Data Placement vs. Receive Events**
>
> This feature only handles the placement of packet *data* in the receive buffer. The application should not expect the packet *receive events* (completions) to arrive in any specific order.

5.4.17.7 Configuration Phase

To start using the library users must first go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context to allow execution of tasks and retrieval of events.

5.4.17.7.1 Configurations

The context can be configured to match the application use case.

To find if a configuration is supported or its min/max value, refer to section "Device Support".

5.4.17.7.1.1 Mandatory Configurations

These configurations must be set by the application before attempting to start the context:

- An event type must be configured. See configuration of Events.
- The library must be initialized before use and released at the end.
- The memory buffer(s) that hold packet data
- The number of stream elements
- Minimal packet segment size(s)
- Maximal packet segment size(s)

5.4.17.7.1.2 Optional Configurations

If the following configurations are not set, then a default value is used:

- CPU affinity for Rivermax internal thread. Should be done before library initialization.
- The input stream packet's data scatter type – defaults to raw.
- The input stream timestamp format – defaults to raw counter.
- The number of segments (memory blocks) to split the packet into.
- Hardware-accelerated packet placement order - disabled by default.

5.4.17.7.2 Device Support

DOCA Rivermax Input Stream requires a device to operate. For picking a device see DOCA Core Device Discovery.

The device must be from within the DPU: Either a PF or SF.

It is recommended to choose your device using the following method:

- **doca_devinfo_get_ipv4_addr()**

Some devices can allow different capabilities as follows:

- PTP clock support.

5.4.17.7.3 Buffer Support

Memory block support buffers with the following features:

Buffer Type	Memory Block
Local mmap buffer	Yes
Mmap from PCIe export buffer	Yes
Mmap from RDMA export buffer	No
Linked list buffer	Yes (header/payload split mode)

5.4.17.8 Execution Phase

This section describes execution on CPU using DOCA Core Progress Engine.

5.4.17.8.1 Events

DOCA Rivermax exposes asynchronous events to notify about changes that happen unexpectedly according to the DOCA Core architecture.

Common events are described in DOCA Core Event.

5.4.17.8.1.1 Rx Data

The Rx Data event is used by the stream to notify application that data has been received from the network.

Event Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	`doca_rmax_in_stream_event_rx_data_register`	–

Event Trigger Condition

The event is triggered anytime packet(s) arrive.

Event Output

Common output as described in DOCA Core Event.

In case of success, the following is provided:

- Number of packets received
- Time of arrival of the first packet
- Time of arrival of the last packet
- Sequence number of the first packet
- Array of memory blocks as configured by input stream

In case of error, the following is provided:

- An error code
- A human readable message

The parameters are valid only inside the event callback.

Event Handling

Once an event is triggered, the application may decide to process the received data.

5.4.17.8.2 Runtime Configurations

These configurations can be made after the context has been started:

- The minimal number of packets that the input stream may return in Rx event.
- The maximal number of packets that the input stream may return in Rx event.
- The receive timeout. The number of µsecs that library would do busy wait (polling) for reception of at least **min_packets** number of packets.

5.4.17.9 State Machine

The DOCA RMAX library follows the Context state machine as described in DOCA Core Context State Machine

The following section describes how to move to the state and what is allowed in each state.

5.4.17.9.1 Idle

In this state, it is expected that application either:

- Destroys the context

- Starts the context

Allowed operations:

- Configuring the context according to Configurations
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop

5.4.17.9.2 Starting

This state is not expected to be reached.

5.4.17.9.3 Running

In this state, it is expected that application:

- Calls progress to receive events

Allowed operations:

- Calling stop
- Changing runtime configurations as described in Runtime Configurations

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration

5.4.17.9.4 Stopping

This state is not expected to be reached.

5.4.17.10 DOCA Rivermax Samples

The samples illustrate how to use the DOCA Rivermax API to:

- List available devices, including their IP and supported capabilities
- Set CPU affinity for the internal Rivermax thread to achieve better performance
- Set the PTP clock device to be used internally in DOCA Rivermax
- Create a stream, create a flow and attach it to the created stream, and finally to start receiving data buffers (based on the attached flow)
- Create a stream in header-data split mode when packet headers and payload are split to different RX buffers
- Create a stream using hardware-accelerated packet placement mode

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

5.4.17.10.1 Running the Samples

1. Refer to the following documents:
   - DOCA Installation Guide for Linux for details on how to install BlueField-related software
   - DOCA Troubleshooting for any issue you may encounter with the installation, compilation, or execution of DOCA samples
2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

```
cd /opt/mellanox/doca/samples/doca_rmax/<sample_name>
meson /tmp/build
ninja -C /tmp/build
```

   The binary **doca_<sample_name>** is created under **/tmp/build/**.

3. Sample (e.g., **doca_rmax_create_stream**) usage:

```
Usage: doca_rmax_create_stream [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                            Print a help synopsis
  -v, --version                         Print program version information
  -l, --log-level                       Set the (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                     Parse command line flags from an input json file

Program Flags:
  -p, --pci_addr <PCI-ADDRESS>          PCI device address
```

   When running DOCA Rivermax samples, the IPv4 address 192.168.105.2 must be configured to an available uplink prior to running it for the samples to run as expected:

```
$ifconfig p0 192.168.105.2
```

4. For additional information per sample, use the **-h** option:

```
/tmp/build/<sample_name> -h
```

5.4.17.10.2 Samples

These samples are also available on GitHub.

5.4.17.10.2.1 List Devices

This sample illustrates how to list all available devices, dump their IPv4 addresses, and tell whether the PTP clock is supported.

The sample logic includes:

1. Initializing DOCA Rivermax library.
2. Iterating over the available devices.
3. Dumping their IPv4 addresses
4. Dumping whether a PTP clock is supported for each device.
5. Releasing DOCA Rivermax library.

References:

- **/opt/mellanox/doca/samples/doca_rmax/rmax_list_devices/rmax_list_devices_sample.c**
- **/opt/mellanox/doca/samples/doca_rmax/rmax_list_devices/rmax_list_devices_main.c**
- **/opt/mellanox/doca/samples/doca_rmax/rmax_list_devices/meson.build**
- **/opt/mellanox/doca/samples/doca_rmax/rmax_common.h** ; **/opt/mellanox/doca/samples/doca_rmax/rmax_common.c**

5.4.17.10.2.2 Set CPU Affinity

This sample illustrates how to set the CPU affinity mask for Rivermax internal thread to achieve better performance. This parameter must be set before library initialization otherwise it will not be applied.

The sample logic includes:

1. Setting CPU affinity using the DOCA Rivermax API.
2. Initializing DOCA Rivermax library.
3. Releasing DOCA Rivermax library.

References:

- **/opt/mellanox/doca/samples/doca_rmax/rmax_set_affinity/rmax_set_affinity_sample.c**
- **/opt/mellanox/doca/samples/doca_rmax/rmax_set_affinity/rmax_set_affinity_main.c**
- **/opt/mellanox/doca/samples/doca_rmax/rmax_set_affinity/meson.build**
- **/opt/mellanox/doca/samples/doca_rmax/rmax_common.h** ; **/opt/mellanox/doca/samples/doca_rmax/rmax_common.c**

5.4.17.10.2.3 Set Clock

This sample illustrates how to set the PTP clock device to be used internally in DOCA Rivermax.

The sample logic includes:

1. Opening a DOCA device with a given PCIe address.
2. Initializing the DOCA Rivermax library.
3. Setting the device to use for obtaining PTP time.
4. Releasing the DOCA Rivermax library.

References:

- **/opt/mellanox/doca/samples/doca_rmax/rmax_set_clock/rmax_set_clock_sample.c**
- **/opt/mellanox/doca/samples/doca_rmax/rmax_set_clock/rmax_set_clock_main.c**
- **/opt/mellanox/doca/samples/doca_rmax/rmax_set_clock/meson.build**
- **/opt/mellanox/doca/samples/doca_rmax/rmax_common.h** ; **/opt/mellanox/doca/samples/doca_rmax/rmax_common.c**

5.4.17.10.2.4 Create Stream

This sample illustrates how to create a stream, create a flow and attach it to the created stream, and finally to start receiving data buffers (based on the attached flow).

The sample logic includes:

1. Opening a DOCA device with a given PCIe address.
2. Initializing the DOCA Rivermax library.
3. Creating an input stream.
4. Creating the context from the created stream.
5. Initializing DOCA Core related objects.
6. Setting the attributes of the created stream.
7. Creating a flow and attaching it to the created stream.
8. Starting to receive data buffers.
9. Clean up—detaches flow and destroys it, destroys created stream and DOCA Core related objects.

References:

- **/opt/mellanox/doca/samples/doca_rmax/rmax_create_stream/rmax_create_stream_sample.c**
- **/opt/mellanox/doca/samples/doca_rmax/rmax_create_stream/rmax_create_stream_main.c**
- **/opt/mellanox/doca/samples/doca_rmax/rmax_create_stream/meson.build**
- **/opt/mellanox/doca/samples/doca_rmax/rmax_common.h** ; **/opt/mellanox/doca/samples/doca_rmax/rmax_common.c**

5.4.17.10.2.5 Create Stream – Header-data Split Mode

This sample illustrates how to create a stream in header-data split mode when packet headers and payload are split to different RX buffers.

The sample logic includes:

1. Opening a DOCA device with a given PCIe address.
2. Initialize the DOCA Rivermax library.
3. Creating an input stream.
4. Creating a context from the created stream.
5. Initializing DOCA Core related objects.
6. Setting attributes of the created stream. Chaining buffers and setting header size to non-zero is essential to create a stream with header-data split mode.
7. Creating a flow and attaching it to the created stream.
8. Starting to receive data to split buffers.

9. Clean up—detaches flow and destroys it, destroys created stream and DOCA Core related objects.

References:

- **/opt/mellanox/doca/samples/doca_rmax/rmax_create_stream_hds/rmax_create_stream_hds_sample.c**
- **/opt/mellanox/doca/samples/doca_rmax/rmax_create_stream_hds/rmax_create_stream_hds_main.c**
- **/opt/mellanox/doca/samples/doca_rmax/rmax_create_stream_hds/meson.build**
- **/opt/mellanox/doca/samples/doca_rmax/rmax_common.h** ; **/opt/mellanox/doca/samples/doca_rmax/rmax_common.c**

5.4.17.10.2.6 Create Stream – Hardware Packet Placement Order Mode

This sample demonstrates how to enable hardware-accelerated packet placement for a DOCA Rivermax stream. In this mode, the NIC uses the sequence number from the RTP header to determine the correct order for placing packets in the receive buffer.

The sample logic includes:

1. Opening a DOCA device with a given PCIe address.
2. Initialize the DOCA Rivermax library.
3. Creating an input stream.
4. Creating a context from the created stream.
5. Initializing DOCA Core related objects.
6. Setting attributes of the created stream (packet placement order, etc.), allocating header and payload buffers.
7. Creating a flow and attaching it to the created stream.
8. Starting to receive data to split buffers.
9. Clean up—detaches flow and destroys it, destroys created stream and DOCA Core related objects.

References:

- **/opt/mellanox/doca/samples/doca_rmax/rmax_create_stream_hw_order_hds/rmax_create_stream_hw_order_hds_sample.c**
- **/opt/mellanox/doca/samples/doca_rmax/rmax_create_stream_hw_order_hds/rmax_create_stream_hw_order_hds_main.c**
- **/opt/mellanox/doca/samples/doca_rmax/rmax_create_stream_hw_order_hds/meson.build**
- **/opt/mellanox/doca/samples/doca_rmax/rmax_common.h** ; **/opt/mellanox/doca/samples/doca_rmax/rmax_common.c**

5.4.18 DOCA Telemetry Exporter

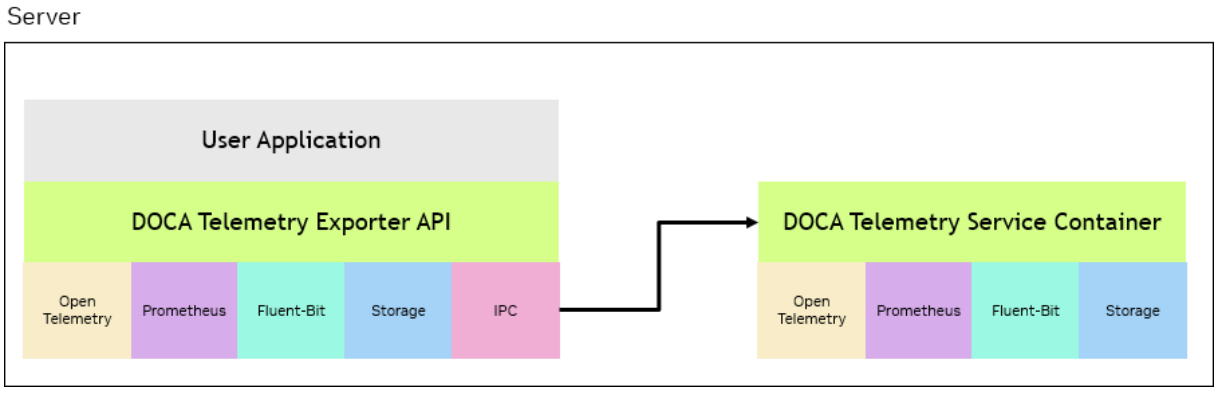
This guide provides an overview and configuration instructions for DOCA Telemetry Exporter API.

5.4.18.1 Introduction

The quality status of DOCA libraries is listed here.

DOCA Telemetry Exporter API offers a fast and convenient way to transfer user-defined data to DOCA Telemetry Service (DTS). In addition, the API provides several built-in outputs for user convenience, including saving data directly to storage, NetFlow, Fluent Bit forwarding, and Prometheus endpoint.

The following figure shows an overview of the telemetry exporter API. The telemetry exporter client side, based on the telemetry exporter API, collects user-defined telemetry and sends it to DTS which runs as a container on BlueField or host. DTS does further data routing, including export with filtering. DTS can process several user-defined telemetry exporter clients and can collect pre-defined counters by itself. Additionally, telemetry exporter API has built-in data outputs that can be used from telemetry exporter client applications.



The following scenarios are available:

- Send data via IPC transport to DTS. For IPC, refer to Inter-process Communication.
- Write data as binary files to storage (for debugging data format).
- Export data directly from DOCA Telemetry Exporter API application using the following options:
  - Fluent Bit exports data through forwarding
  - NetFlow exports data from NetFlow API. Available from both API and DTS. See details in Data Outputs.
  - Prometheus
    - Server mode – Creates Prometheus endpoint and keeps the most recent data to be scraped by Prometheus.
    - Remote write mode – Pushes Prometheus time-series metrics to a remote Prometheus DB supporting remote-write import.
  - Open Telemetry export

Users can either enable or disable any of the data outputs mentioned above. See Data Outputs to see how to enable each output.

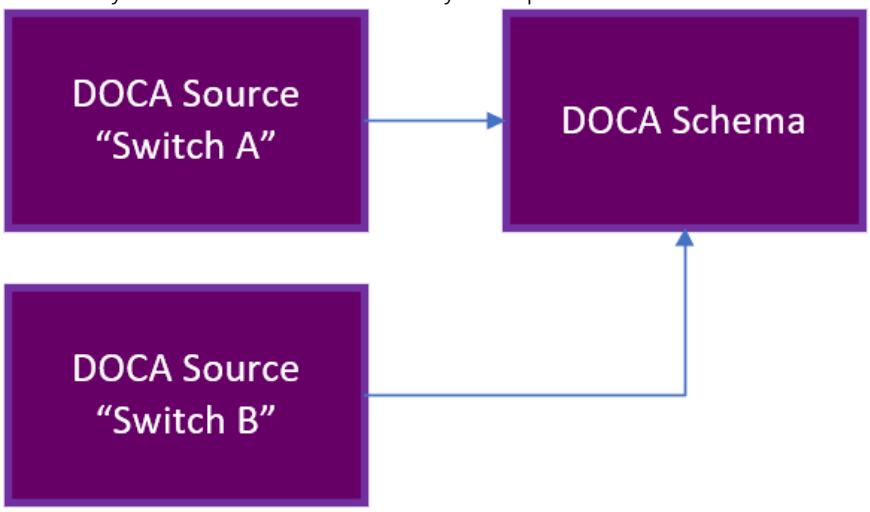
The library stores data in an internal buffer and flushes it to DTS/exporters in the following scenarios:

- Once the buffer is full. Buffer size is configurable with different attributes.
- When **doca_telemetry_exporter_source_flush(void *doca_source)** function is invoked.
- When the telemetry exporter client terminates. If the buffer has data, it is processed before the library's context cleanup.

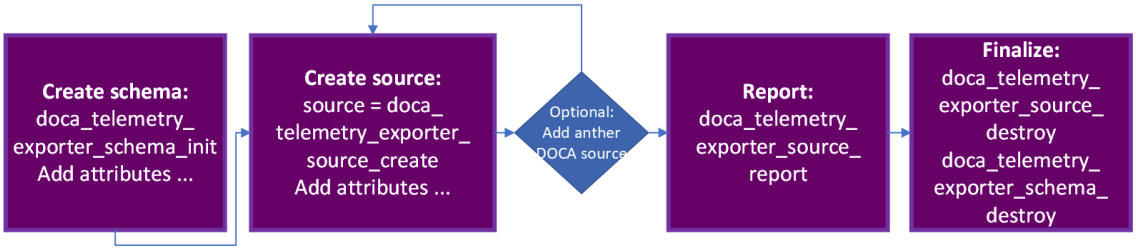
5.4.18.2 Architecture

DOCA Telemetry Exporter API is fundamentally built around four major parts:

- DOCA schema – defines a reusable structure (see doca_telemetry_exporter_type) of telemetry data which can be used by multiple sources



- Source – the unique identifier of the telemetry exporter source that periodically reports telemetry data.
- Report – exports the information to the DTS
- Finalize – releases all the resources



5.4.18.2.1 DOCA Telemetry Exporter API Walkthrough

The NVIDIA DOCA Telemetry Exporter API's definitions can be found in the **doca_telemetry_exporter.h** file.

The following is a basic walkthrough of the needed steps for using the DOCA Telemetry Exporter API.

1. Create **doca_schema** .
    a. Initialize an empty schema with default attributes:

```
struct doca_telemetry_exporter_schema *doca_schema;
doca_telemetry_exporter_schema_init("example_doca_schema_name", &doca_schema);
```

b. Set the following attributes if needed:

- **doca_telemetry_exporter_schema_set_buffer_attr_*(...)**
- **doca_telemetry_exporter_schema_set_file_write_*(...)**
- **doca_telemetry_exporter_schema_set_ipc_*(...)**

c. Add user event types:

Event type (**struct doca_telemetry_exporter_type**) is the user-defined data structure that describes event fields. The user is allowed to add multiple fields to the event type. Each field has its own attributes that can be set (see example). Each event type is allocated an index (**doca_telemetry_exporter_type_index_t**) which can be used to refer to the event type in future API calls.

```
struct doca_telemetry_exporter_type *doca_type;
struct doca_telemetry_exporter_field *field1;

doca_telemetry_exporter_type_create(&doca_type);
doca_telemetry_exporter_field_create(&field1);

doca_telemetry_exporter_field_set_name(field1, "sport");
doca_telemetry_exporter_field_set_description(field1, "Source port")
doca_telemetry_exporter_field_set_type_name(field1, DOCA_TELEMETRY_EXPORTER_FIELD_TYPE_UINT16);
doca_telemetry_exporter_field_set_array_length(field1, 1);

/* The user loses ownership on field1 after a successful invocation of the function */
doca_telemetry_exporter_type_add_field(type, field1);

/* Add more fields if needed */

/* The user loses ownership on doca_type after a successful invocation of the function */
doca_telemetry_exporter_schema_add_type(doca_schema, "example_event", doca_type, &type_index);
```

d. Apply attributes and types to start using the schema:

```
doca_telemetry_exporter_schema_start(doca_schema)
```

2. Create **doca_source** :

a. Initialize:

```
struct doca_telemetry_exporter_source *doca_source;
doca_telemetry_exporter_source_create(doca_schema, &doca_source);
```

b. Set source ID and tag:

```
doca_telemetry_exporter_source_set_id(doca_source, "example id");
doca_telemetry_exporter_source_set_tag(doca_source, "example tag");
```

c. Apply attributes to start using the source:

```
doca_telemetry_exporter_source_start(doca_source)
```

You may optionally add more **doca_sources** if needed.

3. Collect the data per source and use:

```
doca_telemetry_exporter_source_report(source, type_index, &my_app_test_ev1, num_events)
```

4. Finalize:

a. For every source:

```
doca_telemetry_exporter_source_destroy(source)
```

b. Destroy:

```
doca_telemetry_exporter_schema_destroy(doca_schema)
```

Example implementation may be found in the **telemetry_export** DOCA sample (**telemetry_export_sample.c**).

5.4.18.2.2 DOCA Telemetry Exporter NetFlow API Walkthrough

The DOCA telemetry exporter API also supports NetFlow using DOCA Telemetry Exporter NetFlow API. This API is designed to allow customers to easily support the NetFlow protocol at the endpoint side. Once an endpoint produces NetFlow data using the API, the corresponding exporter can be used to send the data to a NetFlow collector.

The NVIDIA DOCA Telemetry Exporter Netflow API's definitions can be found in the **doca_telemetry_exporter_netflow.h** file.

The following are the steps to use the NetFlow API:

1. Initiate the API with an appropriate source ID:

```
doca_telemetry_exporter_netflow_init(source_id)
```

2. Set the relevant attributes:
   - **doca_telemetry_exporter_netflow_set_buffer_*(...)**
   - **doca_telemetry_exporter_netflow_set_file_write_*(...)**
   - **doca_telemetry_exporter_netflow_set_ipc_*(...)**
   - **doca_telemetry_exporter_netflow_source_set_*()**
3. Start the API to use the configured attribute:

```
doca_telemetry_exporter_netflow_start();
```

4. Form a desired NetFlow template and the corresponding NetFlow records.
5. Collect the NetFlow data.

```
doca_telemetry_exporter_netflow_send(…)
```

6. (Optional) Flush the NetFlow data to send data immediately instead of waiting for the buffer to fill:

```
doca_telemetry_exporter_netflow_flush()
```

7. Clean up the API:

```
doca_telemetry_exporter_netflow_destroy()
```

Example implementation may be found in the **telemetry_export_netflow** DOCA sample (**telemetry_export_netflow_sample.c**).

5.4.18.2.3 DOCA Telemetry Exporter Metrics API Walkthrough

The DOCA Telemetry Exporter provides a Metrics API for the programmatic collection and export of performance metrics from applications. This API enables applications to report counters, gauges, and histograms with multi-dimensional labels, supporting export to multiple backends including file storage, IPC, and Prometheus.

5.4.18.2.3.1 Supported Metric Types

- Counter: Monotonically increasing values (e.g., total requests, bytes sent, operations completed).
- Gauge: Current values that can increase or decrease (e.g., memory usage, queue depth, bandwidth).
- Histogram: Distribution of values across configurable buckets (e.g., latency percentiles, message sizes).

5.4.18.2.3.2 Initialization

The Metrics API requires a DOCA Telemetry Exporter source to be initialized before metrics can be reported.

Prerequisites:

1. Initialize the DOCA schema (**doca_telemetry_exporter_schema_init**).
2. Configure and start the schema (**doca_telemetry_exporter_schema_start**).
3. Create the source (**doca_telemetry_exporter_source_create**).
4. Start the source (**doca_telemetry_exporter_source_start**).

At least one export backend (file, IPC, or Prometheus) must be enabled on the schema before creating the metrics context.

Example initialization:

```
struct doca_telemetry_exporter_schema *schema = NULL;
struct doca_telemetry_exporter_source *source = NULL;
doca_error_t result;

// 1. Initialize schema and source
result = doca_telemetry_exporter_schema_init("my_app", &schema);
if (result != DOCA_SUCCESS) { /* Handle error */ }

doca_telemetry_exporter_schema_start(schema);
doca_telemetry_exporter_source_create(schema, &source);
doca_telemetry_exporter_source_set_id(source, "my_source");
doca_telemetry_exporter_source_set_tag(source, "");
doca_telemetry_exporter_source_start(source);

// 2. Create metrics context
result = doca_telemetry_exporter_metrics_create_context(source);
if (result != DOCA_SUCCESS) {
    // Handle error
}

// 3. Configure metrics (Optional)
// Set periodic flush to every 1 second (1000 ms)
doca_telemetry_exporter_metrics_set_flush_interval_ms(source, 1000);
// Add a constant label applied to all metrics from this source
doca_telemetry_exporter_metrics_add_constant_label(source, "application", "my_app");
```

5.4.18.2.3.3 Working with Labels

Label sets define the dimensional structure of metrics. A label set specifies the *names* of the labels (e.g., "interface", "direction"), while the metrics themselves provide the specific *values* (e.g., "eth0", "tx").

Creating Label Set

```
// Define label names
const char *label_names[] = {"interface", "direction"};
doca_telemetry_exporter_label_set_id_t label_set_id;

// Register the label set
result = doca_telemetry_exporter_metrics_add_label_names(
    source,
    label_names,
    2,  // Number of labels
    &label_set_id
);
```

Using Label Values

```
// Define values matching the label names
const char *eth0_tx_labels[] = {"eth0", "tx"};
const char *eth1_rx_labels[] = {"eth1", "rx"};

uint64_t timestamp;
doca_telemetry_exporter_get_timestamp(&timestamp);

// Report metrics using different label combinations
doca_telemetry_exporter_metrics_add_counter(source, timestamp,
    "bytes_total", 1024000, label_set_id, eth0_tx_labels);

doca_telemetry_exporter_metrics_add_counter(source, timestamp,
    "bytes_total", 2048000, label_set_id, eth1_rx_labels);
```

Constant Labels

Constant labels are applied globally to all metrics reported from a specific source.

```
doca_telemetry_exporter_metrics_add_constant_label(source, "hostname", "server01");
doca_telemetry_exporter_metrics_add_constant_label(source, "version", "1.0");
```

5.4.18.2.3.4 Counters

Counters represent cumulative totals. They are strictly monotonically increasing.

API functions:

- **doca_telemetry_exporter_metrics_add_counter()** : Set counter to an absolute value.
- **doca_telemetry_exporter_metrics_add_counter_increment()** : Increment counter by a specified amount.

Example:

```
uint64_t timestamp;
doca_telemetry_exporter_get_timestamp(&timestamp);
const char *label_values[] = {"requests"};

// Set absolute counter value
doca_telemetry_exporter_metrics_add_counter(source, timestamp,
    "operations_total", 42, label_set_id, label_values);

// Increment counter
doca_telemetry_exporter_metrics_add_counter_increment(source, timestamp,
    "operations_total", 1, label_set_id, label_values);
```

5.4.18.2.3.5 Gauges

Gauges represent instantaneous values that can go up or down, such as memory usage or bandwidth.

API functions:

- **doca_telemetry_exporter_metrics_add_gauge()** : Report a gauge with a **double** precision value.
- **doca_telemetry_exporter_metrics_add_gauge_uint64()** : Report a gauge with a **uint64_t** value.

Example:

```
uint64_t timestamp;
doca_telemetry_exporter_get_timestamp(&timestamp);
const char *label_values[] = {"bandwidth"};

// Double precision gauge
double bandwidth_gbps = 10.5;
doca_telemetry_exporter_metrics_add_gauge(source, timestamp,
    "bandwidth_gbps", bandwidth_gbps, label_set_id, label_values);

// Integer gauge
uint64_t memory_bytes = 1073741824;
doca_telemetry_exporter_metrics_add_gauge_uint64(source, timestamp,
    "memory_usage_bytes", memory_bytes, label_set_id, label_values);
```

5.4.18.2.3.6 Histograms

Histograms track the distribution of values (observations) across configurable buckets.

Creating and Observing a Histogram

```
double latency_buckets[] = {1.0, 5.0, 10.0, 25.0, 50.0, 100.0, 500.0, 1000.0};
const char *label_values[] = {"tcp"};
doca_telemetry_exporter_histogram_id_t histogram_id;

// Create the histogram definition
result = doca_telemetry_exporter_metrics_add_histogram(
    source, timestamp, "latency_ms",
    latency_buckets, 8,  // 8 bucket boundaries
    label_set_id, label_values,
    &histogram_id
);

// Record observations
doca_telemetry_exporter_metrics_histogram_observe(source, histogram_id, 15.3);
doca_telemetry_exporter_metrics_histogram_observe(source, histogram_id, 8.7);
doca_telemetry_exporter_metrics_histogram_observe(source, histogram_id, 42.1);
```

Base Histogram (Templates)

For metrics that require multiple label combinations, use a base histogram as a template.

```
doca_telemetry_exporter_base_histogram_id_t base_histogram_id;

// Create base histogram template
result = doca_telemetry_exporter_metrics_add_base_histogram(
    source, "latency_ms",
    latency_buckets, 8,
    label_set_id,
    1000,  // flush_interval_ms
    &base_histogram_id
);

// Observe with specific label values (creates instance automatically)
const char *tcp_labels[] = {"tcp"};
doca_telemetry_exporter_histogram_id_t tcp_histogram_id;

doca_telemetry_exporter_metrics_base_histogram_observe(
    source, base_histogram_id, tcp_labels, 15.3, &tcp_histogram_id
);
```

5.4.18.2.3.7 Best Practices

Flushing Metrics

Metrics are buffered internally. Configure periodic automatic flushing for frequent exports, and always flush before program exit.

```
// Configure automatic flush (e.g., every 1 second)
doca_telemetry_exporter_metrics_set_flush_interval_ms(source, 1000);

// Flush manually before destruction
doca_telemetry_exporter_metrics_flush(source);
```

Label Cardinality

Keep the total number of unique label combinations reasonable to avoid high memory usage and backend strain.

- Good: Low cardinality labels (e.g., interface names, status codes).
- Bad: High cardinality labels (e.g., random flow IDs, unique request IDs).
- Recommendation: Keep total unique label combinations under 10,000.

Metric Naming Conventions

For Prometheus compatibility, adhere to the following naming standards:

- Format: **snake_case** (e.g., **network_bandwidth_gbps**).
- Units: Suffix with the unit (e.g., **_bytes**, **_seconds**).
- Counters: End with **_total** (e.g., **requests_total**).
- Gauges: Reflect current state (e.g., **memory_usage_bytes**).

Error Handling

Always check return values (**doca_error_t**) for all metrics functions.

```
if (result != DOCA_SUCCESS) {
    DOCA_LOG_ERR("Failed to add counter: %s", doca_error_get_descr(result));
}
```

Cleanup

Properly destroy resources in reverse order of creation.

```
doca_telemetry_exporter_metrics_flush(source);
doca_telemetry_exporter_metrics_destroy_context(source);
doca_telemetry_exporter_source_destroy(source);
doca_telemetry_exporter_schema_destroy(schema);
```

5.4.18.2.4 DOCA Telemetry Exporter OTLP Logs API Walkthrough

The DOCA Telemetry Exporter provides an OTLP Logs API for the programmatic collection and export of structured log events from applications.

This API enables applications to report log records with resources, scopes, severity levels, and typed attributes, supporting export to OTLP-compatible backends such as OpenTelemetry Collector, Jaeger, and cloud observability platforms.

5.4.18.2.4.1 Supported Concepts

Concept	Description
Resource	Represents the entity generating logs (e.g., service, device, node). Resources carry attributes such as **service.name**, **host.name**, and **device.name**.
Scope	Represents an instrumentation library or functional component. A scope defines the event schema, including attribute keys, attribute types, event name, and default severity.
Event	A log record emitted using a resource and scope. Events provide attribute values that must match the scope schema and may override the default severity.
Attributes	Key-value pairs attached to resources or events. Supported types are **STRING**, **INT**, **DOUBLE**, and **BOOL**.

5.4.18.2.4.2 Initialization

The OTLP Logs API requires a DOCA Telemetry Exporter schema and source to be initialized before log events can be reported.

5.4.18.2.4.3 Prerequisites

1. Initialize the DOCA schema (**doca_telemetry_exporter_schema_init**).
2. Enable opaque events (**doca_telemetry_exporter_schema_set_opaque_events_enabled**)

   > ⚠ Required for OTLP logs.

3. Configure and start the schema (**doca_telemetry_exporter_schema_start**).
4. Create the source (**doca_telemetry_exporter_source_create**).
5. Start the source (**doca_telemetry_exporter_source_start**).
6. Create the OTLP logs context (**doca_telemetry_exporter_otlp_logs_create_context**).

5.4.18.2.4.4 Example Initialization

```
struct doca_telemetry_exporter_schema *schema = NULL;
struct doca_telemetry_exporter_source *source = NULL;
doca_error_t result;

// Initialize schema
result = doca_telemetry_exporter_schema_init("my_app", &schema);
if (result != DOCA_SUCCESS) { /* Handle error */ }

// Configure schema
doca_telemetry_exporter_schema_set_buf_data_root(schema, "/tmp/doca_telemetry_buffer");
doca_telemetry_exporter_schema_set_opaque_events_enabled(schema); // Required

// Start schema and source
doca_telemetry_exporter_schema_start(schema);
doca_telemetry_exporter_source_create(schema, &source);
doca_telemetry_exporter_source_set_id(source, "my_source");
doca_telemetry_exporter_source_set_tag(source, "");
doca_telemetry_exporter_source_start(source);

// Create OTLP logs context
result = doca_telemetry_exporter_otlp_logs_create_context(source);
if (result != DOCA_SUCCESS) { /* Handle error */ }
```

5.4.18.2.4.5 Working with Resources

Resources describe the entity producing log events.

Creating a Resource

```
doca_telemetry_exporter_otlp_logs_resource_id_t resource_id;
result = doca_telemetry_exporter_otlp_logs_add_resource(source, &resource_id);
```

Adding Resource Attributes

```
doca_telemetry_exporter_otlp_logs_add_resource_attribute_string(
    source, resource_id, "service.name", "doca_network_service");

doca_telemetry_exporter_otlp_logs_add_resource_attribute_string(
    source, resource_id, "host.name", "server01");

doca_telemetry_exporter_otlp_logs_add_resource_attribute_string(
    source, resource_id, "device.name", "mlx5_0");

doca_telemetry_exporter_otlp_logs_add_resource_attribute_int(
    source, resource_id, "device.port", 1);

doca_telemetry_exporter_otlp_logs_add_resource_attribute_bool(
    source, resource_id, "is_primary", true);
```

5.4.18.2.4.6 Working with Scopes

Scopes define the event schema and default severity.

Creating a Scope

```
const char *attr_keys[] = {"message", "error_code", "severity"};
doca_telemetry_exporter_otlp_attribute_type_t attr_types[] = {
    DOCA_TELEMETRY_EXPORTER_OTLP_ATTRIBUTE_TYPE_STRING,
    DOCA_TELEMETRY_EXPORTER_OTLP_ATTRIBUTE_TYPE_INT,
    DOCA_TELEMETRY_EXPORTER_OTLP_ATTRIBUTE_TYPE_STRING
};

doca_telemetry_exporter_otlp_logs_scope_id_t scope_id;

result = doca_telemetry_exporter_otlp_logs_add_scope(
    source,
    "doca.telemetry.example",  // scope_name
    "1.0.0",                   // scope_version
    "network_event",           // event_name
    "INFO",                    // default_severity
    attr_keys,
    attr_types,
    3,
    &scope_id
);
```

5.4.18.2.4.7 Writing Log Events

Events must provide attribute values matching the scope schema and order.

Writing Event with Default Severity

```
doca_telemetry_exporter_otlp_attribute_value_t values[3];
values[0].string_value = "Connection established";
values[1].int_value = 0;
values[2].string_value = "INFO";

doca_telemetry_exporter_otlp_logs_write_event(
    source, resource_id, scope_id, values);
```

Writing Event with Custom Severity

```
doca_telemetry_exporter_otlp_attribute_value_t warn_values[3];
warn_values[0].string_value = "Packet loss detected";
warn_values[1].int_value = 2001;
warn_values[2].string_value = "WARN";

doca_telemetry_exporter_otlp_logs_write_event_with_severity(
    source, resource_id, scope_id, warn_values, "WARN");
```

5.4.18.2.4.8 Flushing and Cleanup

```
doca_telemetry_exporter_otlp_logs_flush(source);
doca_telemetry_exporter_otlp_logs_destroy_context(source);
doca_telemetry_exporter_source_destroy(source);
doca_telemetry_exporter_schema_destroy(schema);
```

5.4.18.2.4.9 Best Practices

Flushing Logs

Logs are buffered internally. Flush periodically or before shutdown to ensure delivery.

Severity Usage

Use consistent severity levels (`INFO` , `WARN` , `ERROR`) to simplify downstream filtering and alerting.

5.4.18.2.4.10 Error Handling

Always check `doca_error_t` return values.

```
if (result != DOCA_SUCCESS) { DOCA_LOG_ERR("OTLP logs error: %s", doca_error_get_descr(result)); }
```

5.4.18.3 API

Refer to DOCA Library APIs, for more detailed information on DOCA Telemetry Exporter API.

> ⚠ The pkg-config (`*.pc` file) for the DOCA Telemetry Exporter library is `doca-telemetry-exporter` .

The following sections provide additional details about the library API.

Some attributes are optional as they are initialized with default values. Refer to the documentation of the setter functions of respective attributes for more information.

5.4.18.3.1 DOCA Telemetry Exporter Buffer Attributes

Buffer attributes are used to set the internal buffer size and data root used by all DOCA sources in the schema.

Configuring the attributes is optional as they are initialized with default values.

```
doca_telemetry_exporter_schema_set_buffer_size(doca_schema, 16 * 1024); /* 16KB - arbitrary value */
doca_telemetry_exporter_schema_set_buffer_data_root(doca_schema, "/opt/mellanox/doca/services/telemetry/data/");
```

- **buffer_size [in]** – the size of the internal buffer which accumulates the data before sending it to the outputs. Data is sent automatically once the internal buffer is full. Larger buffers mean fewer data transmissions and vice versa.
- **data_root [in]** – the path to where data is stored (if **file_write_enabled** is set to true). See section "DOCA Telemetry Exporter File Write Attributes".

5.4.18.3.2 DOCA Telemetry Exporter File Write Attributes

File write attributes are used to enable and configure data storage to the file system in binary format.

Configuring the attributes is optional as they are initialized with default values.

```
doca_telemetry_exporter_schema_set_file_write_enabled(doca_schema);
doca_telemetry_exporter_schema_set_file_write_max_size(doca_schema, 1 * 1024 * 1024); /* 1 MB */
doca_telemetry_exporter_schema_set_file_write_max_age(doca_schema, 60 * 60 * 1000000L); /* 1 Hour */
```

- **file_write_enable [in]** – use this function to enable storage. Storage/FileWrite is disabled by default.
- **file_write_max_size [in]** – maximum file size (in bytes) before a new file is created.
- **file_write_max_age [in]** – maximum file age (in microseconds) before a new file is created.

5.4.18.3.3 DOCA Telemetry Exporter IPC Attributes

IPC attributes are used to enable and configure IPC transport. IPC is disabled by default.

Configuring the attributes is optional as they are initialized with default values.

> ⚠ It is important to make sure that the IPC location matches the IPC location used by DTS, otherwise IPC communication will fail. IPC requires write access to the DTS socket. Since it is created by root, the client may need to run as root.

```
doca_telemetry_exporter_schema_set_ipc_enabled(doca_schema);
doca_telemetry_exporter_schema_set_ipc_sockets_dir(doca_schema, "/path/to/sockets/");
doca_telemetry_exporter_schema_set_ipc_reconnect_time(doca_schema, 100); /* 100 milliseconds */
doca_telemetry_exporter_schema_set_ipc_reconnect_tries(doca_schema, 3);
doca_telemetry_exporter_schema_set_ipc_socket_timeout(doca_schema, 3 * 1000) /* 3 seconds */
```

- **ipc_enabled [in]** – use this function to enable communication. IPC is disabled by default.
- **ipc_sockets_dir [in]** – a directory that contains UDS for IPC messages. Both the telemetry exporter program and DTS must use the same folder. DTS that runs on BlueField as a container has the default folder **/opt/mellanox/doca/services/telemetry/ipc_sockets**.
- **ipc_reconnect_time [in]** – maximum reconnection time in milliseconds after which the client is considered disconnected.
- **ipc_reconnect_tries [in]** – maximum reconnection attempts.
- **ipc_socket_timeout [in]** – timeout for the IPC socket.

5.4.18.3.4 DOCA Telemetry Exporter Source Attributes

Source attributes are used to create proper folder structure. All the data collected from the same host is written to the **source_id** folder under data root.

> ⚠ Sources attributes are mandatory and must be configured before invoking **doca_telemetry_exporter_source_start()**.

```
doca_telemetry_exporter_source_set_id(doca_source, "example_source");
doca_telemetry_exporter_source_set_tag(doca_source, "example_tag");
```

- **source_id [in]** – describes the data's origin. It is recommended to set it to the hostname. In later dataflow steps, data is aggregated from multiple hosts/DPUs and **source_id** helps navigate in it.
- **source_tag [in]** – a unique data identifier. It is recommended to set it to describe the data collected in the application. Several telemetry exporter apps can be deployed on a single node (host/DPU). In that case, each telemetry data would have a unique tag and all of them would share a single **source_id**.

5.4.18.3.5 DOCA Telemetry Exporter Netflow Collector Attributes

DOCA Telemetry Exporter NetFlow API attributes are optional and should only be used for debugging purposes. They represent the NetFlow collector's address while working locally, effectively enabling the local NetFlow exporter.

```
doca_telemetry_exporter_netflow_set_collector_addr("127.0.0.1");
doca_telemetry_exporter_netflow_set_collector_port(6343);
```

- **collector_addr [in]** – NetFlow collector's address (IP or name). Default value is **NULL**.
- **collector_port [in]** – NetFlow collector's port. Default value is **DOCA_NETFLOW_DEFAULT_PORT (2055)**.

5.4.18.3.6 DOCA Telemetry Exporter Source Report

The source report function is the heart of communication with the DTS. The report operation causes event data to be allocated to the internal buffer. Once the buffer is full, data is forwarded onward according to the set configuration.

```
doca_error_t doca_telemetry_exporter_source_report(struct doca_telemetry_exporter_source *doca_source,
                                                   doca_telemetry_exporter_type_index_t index,
                                                   void *data,
                                                   int count);
```

- **doca_source [in]** – a pointer to the **doca_telemetry_exporter_source** which reports the event
- **index [in]** – the event type index received when the schema was created
- **data [in]** – a pointer to the data buffer that needs to be sent
- **count [in]** – numbers of events to be written to the internal buffer

The function returns **DOCA_SUCCESS** if successful, or a **doca_error_t** if an error occurs. If a memory-related error occurs, try a larger buffer size that matches the event's size.

5.4.18.3.7 DOCA Telemetry Exporter Schema Add Type

This function allows adding a reusable telemetry data struct, also known as a schema. The schema allows sending a predefined data structure to the telemetry service. Note that it is mandatory to define a schema for proper functionality of the library. After adding the schemas, one needs to invoke the schema start function.

```
doca_error_t doca_telemetry_exporter_schema_add_type(struct doca_telemetry_exporter_schema *doca_schema,
                                                     const char *new_type_name,
                                                     struct doca_telemetry_exporter_type *type,
                                                     doca_telemetry_exporter_type_index_t *type_index);
```

- **doca_schema [in]** – a pointer to the schema to which the type is added
- **new_type_name [in]** – name of the new type
- **fields [in]** – user-defined fields to be used for the schema. Multiple fields can (and should) be added.
- **type_index [out]** – type index for the created type is written to this output variable

The function returns **DOCA_SUCCESS** if successful, or **doca_error_t** if an error occurs.

5.4.18.3.8 DOCA Telemetry Exporter Metrics

The Metrics API provides functions for programmatic metrics collection and export. These functions are used in conjunction with the core DOCA Telemetry Exporter API to report performance metrics.

Return Values:

All functions return **doca_error_t** with **DOCA_SUCCESS** on success or error codes on failure. Common error codes include:

- **DOCA_ERROR_INVALID_VALUE** – NULL parameters or invalid arguments.
- **DOCA_ERROR_NO_MEMORY** – Memory allocation failure.

See **doca_telemetry_exporter.h** for detailed function signatures and parameters.

The Metrics API functions are organized into the following five categories:

5.4.18.3.8.1 DOCA Telemetry Exporter Metrics Context Management

These functions manage the lifecycle and state of the metrics context.

Function Name	Description
doca_telemetry_exporter_metrics_create_context	Creates and initializes the metrics context.
doca_telemetry_exporter_metrics_destroy_context	Destroys the metrics context and frees resources.
doca_telemetry_exporter_metrics_flush	Flushes remaining buffered metrics to exporters.

5.4.18.3.8.2 DOCA Telemetry Exporter Metrics Configuration

These functions configure the behavior of the metrics collection, including flushing and labeling.

Function Name	Description
`doca_telemetry_exporter_metrics_set_flush_interval_ms`	Sets periodic automatic flush interval in milliseconds.
`doca_telemetry_exporter_metrics_set_min_sample_interval_us`	Sets minimum time between samples per time series.
`doca_telemetry_exporter_metrics_add_constant_label`	Adds a constant label to all metrics reported by the source.
`doca_telemetry_exporter_metrics_add_label_names`	Defines a label set (list of label names) and returns its ID.

5.4.18.3.8.3 DOCA Telemetry Exporter Metrics Counters

Functions for reporting monotonically increasing values.

Function Name	Description
`doca_telemetry_exporter_metrics_add_counter`	Reports a counter with an absolute value.
`doca_telemetry_exporter_metrics_add_counter_increment`	Increments a counter by a specified value.

5.4.18.3.8.4 DOCA Telemetry Exporter Metrics Gauges

Functions for reporting values that can arbitrarily increase or decrease.

Function Name	Description
`doca_telemetry_exporter_metrics_add_gauge`	Reports a gauge with a `double` precision value.
`doca_telemetry_exporter_metrics_add_gauge_uint64`	Reports a gauge with a `uint64_t` value.

5.4.18.3.8.5 DOCA Telemetry Exporter Metrics Histograms

Functions for creating, observing, and managing histograms.

Function Name	Description
`doca_telemetry_exporter_metrics_add_histogram`	Creates a histogram instance with specific label values.
`doca_telemetry_exporter_metrics_add_base_histogram`	Creates a base histogram template for use with multiple label sets.
`doca_telemetry_exporter_metrics_histogram_observe`	Records a value in a specific histogram instance.

Function Name	Description
doca_telemetry_exporter_metrics_base_histogram_observe	Records a value in a base histogram using specific label values.
doca_telemetry_exporter_metrics_histogram_flush	Flushes data for a specific histogram immediately.
doca_telemetry_exporter_metrics_histogram_remove	Removes a histogram instance from the context.
doca_telemetry_exporter_metrics_set_histogram_flush_interval_target	Sets the flush interval target for a specific histogram.

5.4.18.3.9 DOCA Telemetry Exporter Logs

The Logs API provides functions for programmatic log record collection and export using the OpenTelemetry Protocol (OTLP).

These functions are used together with the core DOCA Telemetry Exporter API to report structured log events.

5.4.18.3.9.1 Return Values

All functions return `doca_error_t` with `DOCA_SUCCESS` on success or an error code on failure.

Common error codes:

- `DOCA_ERROR_INVALID_VALUE` – Invalid arguments or NULL parameters
- `DOCA_ERROR_NO_MEMORY` – Memory allocation failure
- `DOCA_ERROR_BAD_STATE` – Invalid context or state

See `doca_telemetry_exporter.h` for full API definitions.

5.4.18.3.9.2 DOCA Telemetry Exporter Logs Context Management

Functions for managing the lifecycle of the OTLP logs context.

Function Name	Description
doca_telemetry_exporter_otlp_logs_create_context	Creates and initializes the OTLP logs context.
doca_telemetry_exporter_otlp_logs_destroy_context	Destroys the logs context and frees resources.
doca_telemetry_exporter_otlp_logs_flush	Flushes buffered log records to the exporter.

5.4.18.3.9.3 DOCA Telemetry Exporter Logs Resource Management

Functions for defining log resources and their attributes.

Function Name	Description
doca_telemetry_exporter_otlp_logs_add_resource	Adds a new resource and returns its ID.

Function Name	Description
doca_telemetry_exporter_otlp_logs_add_resource_attribute_string	Adds a string attribute to a resource.
doca_telemetry_exporter_otlp_logs_add_resource_attribute_int	Adds an integer attribute to a resource.
doca_telemetry_exporter_otlp_logs_add_resource_attribute_double	Adds a double attribute to a resource.
doca_telemetry_exporter_otlp_logs_add_resource_attribute_bool	Adds a Boolean attribute to a resource.

5.4.18.3.9.4 DOCA Telemetry Exporter Logs Scope Management

Functions for defining log scopes and schemas.

Function Name	Description
doca_telemetry_exporter_otlp_logs_add_scope	Adds a scope with predefined attribute schema and returns its ID.

5.4.18.3.9.5 DOCA Telemetry Exporter Logs Event Writing

Functions for emitting log events.

Function Name	Description
doca_telemetry_exporter_otlp_logs_write_event	Writes a log event using the scope default severity.
doca_telemetry_exporter_otlp_logs_write_event_with_severity	Writes a log event with explicit severity.

5.4.18.3.9.6 DOCA Telemetry Exporter Logs Data Types

Type Name	Description
doca_telemetry_exporter_otlp_logs_resource_id_t	Resource identifier type.
doca_telemetry_exporter_otlp_logs_scope_id_t	Scope identifier type.
doca_telemetry_exporter_otlp_attribute_type_t	Attribute type enum.
doca_telemetry_exporter_otlp_attribute_value_t	Union holding attribute values.
DOCA_TELEMETRY_EXPORTER_OTLP_LOGS_INVALID_ID	Invalid ID constant.

5.4.18.4 Telemetry Event Data Format

The internal data format consists of 2 parts: A schema containing metadata, and the actual binary data. When data is written to storage, the data schema is written in JSON format, and the data is written as binary files. In the case of IPC transport, both schema and binary data are sent to DTS. In the case of export, data is converted to the formats required by exporter.

Adding custom event types to the schema can be done using **doca_telemetry_exporter_schema_add_type** API call.

See available **DOCA_TELEMETRY_EXPORTER_FIELD_TYPE** s in **doca_telemetry_exporter.h**.
See example of usage in **/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export/telemetry_export_sample.c**.

It is highly recommended to have the timestamp field as the first field since it is required by most databases. To get the current timestamp in the correct format use:

```
doca_error_t doca_telemetry_exporter_get_timestamp(doca_telemetry_exporter_timestamp_t *timestamp);
```

5.4.18.5 Data Outputs

This section describes available exporters:

- IPC
- NetFlow
- Fluent Bit
- Prometheus Endpoint
- Prometheus Remote-Write
- Open Telemetry

Fluent Bit and Prometheus exporters are presented in both API and DTS. Even though DTS export is preferable, the API has the same possibilities for development flexibility.

5.4.18.5.1 Inter-process Communication

IPC transport automatically transfers the data from the telemetry-exporter-based program to DTS service.

It is implemented as a UNIX domain socket (UDS) sockets for short messages and shared memory for data. DTS and the telemetry-exporter-based program must share the same **ipc_sockets** directory.

When IPC transport is enabled, the data is sent from the DOCA-telemetry-exporter-based application to the DTS process via shared memory.

To enable IPC, use the **doca_telemetry_exporter_schema_set_ipc_enabled** API function.

To check the IPC status for the current context, use:

```
doca_error_t doca_telemetry_exporter_check_ipc_status(struct doca_telemetry_exporter_source *doca_source,
                                                      doca_telemetry_exporter_ipc_status_t *status)
```

If IPC is enabled and for some reason connection is lost, it would try to automatically reconnect on every report's function call.

5.4.18.5.1.1 Using IPC with Non-containerized Application

When developing or testing a DOCA Telemetry Exporter-based program that runs natively (outside a container), specific configurations are required to allow IPC with the containerized DTS.

To enable this interaction, the DTS container must be configured to use the host's IPC namespace rather than an isolated shared memory volume.

Docker Deployment (Command Line)

When deploying DTS via the **docker** command, ensure the container uses the host's IPC namespace.

- Recommended method: Use the **--ipc=host** flag.
- Alternative method: Explicitly mount the host's shared memory directory:

```
-v /dev/shm:/dev/shm
```

Do not mount non-standard shared memory folders if you intend to use the default system paths.

Kubernetes/Pod Deployment (YAML)

When deploying DTS via a YAML configuration (e.g., on the DPU), you must enable **hostIPC** and remove any specific **volumeMounts** that isolate shared memory (such as **telemetry-ipc-shm**).

Configuration Comparison

- Before change (default Isolated Mode):

```
spec:
  hostNetwork: true
  volumes:
  - name: telemetry-service-config
    hostPath:
      path: /opt/mellanox/doca/services/telemetry/config
      type: DirectoryOrCreate
  # [REMOVE] The following volume isolates shared memory
  - name: telemetry-ipc-shm
    hostPath:
      path: /dev/shm/telemetry
      type: DirectoryOrCreate
  containers:
      volumeMounts:
      - name: telemetry-service-config
        mountPath: /config
      # [REMOVE] The following mount point
      - name: telemetry-ipc-shm
        mountPath: /dev/shm
```

- After change (Host IPC Enabled):

```
spec:
  hostNetwork: true
  hostIPC: true  # [ADD] Enable Host IPC
  volumes:
  - name: telemetry-service-config
    hostPath:
      path: /opt/mellanox/doca/services/telemetry/config
      type: DirectoryOrCreate
  containers:
      volumeMounts:
      - name: telemetry-service-config
        mountPath: /config
```

When developing and testing a non-container DOCA Telemetry-Exporter-based program and its IPC interaction with DTS, some modifications are necessary in DTS's deployment for the program to interact with DTS over IPC:

- Shared memory mapping should be removed: **telemetry-ipc-shm**
- Host IPC should be enabled: **hostIPC**

Host case (docker command)

In case of docker command make sure that DTS uses:

```
--ipc=host
```

and does not mount non-standard shared memory folder.
In addition user can directly specify to mount /dev/shm in docker command for DTS:

```
-v /dev/shm:/dev/shm
```

DPU case (yaml)

File before the change:

```
spec:
  hostNetwork: true
  volumes:
  - name: telemetry-service-config
    hostPath:
      path: /opt/mellanox/doca/services/telemetry/config
      type: DirectoryOrCreate
  ...
  - name: telemetry-ipc-shm
    hostPath:
      path: /dev/shm/telemetry
      type: DirectoryOrCreate
  containers:
  ...
      volumeMounts:
      - name: telemetry-service-config
        mountPath: /config
  ...
      - name: telemetry-ipc-shm
        mountPath: /dev/shm
```

File after the change:

```
spec:
  hostNetwork: true
  hostIPC: true
  volumes:
  - name: telemetry-service-config
    hostPath:
      path: /opt/mellanox/doca/services/telemetry/config
      type: DirectoryOrCreate
  ...
  containers:
  ...
      volumeMounts:
      - name: telemetry-service-config
        mountPath: /config
```

These changes ensure that a DOCA-based program running outside of a container is able to communicate with DTS over IPC.

5.4.18.5.2 NetFlow

When the NetFlow exporter is enabled (NetFlow Collector Attributes are set), it sends the NetFlow data to the NetFlow collector specified by the attributes: Address and port. This exporter must be used when using DOCA Telemetry Exporter NetFlow API.

5.4.18.5.3 Fluent Bit

Fluent Bit export is based on **fluent_bit_configs** with **.exp** files for each destination. Every export file corresponds to one of Fluent Bit's destinations. All found and enabled **.exp** files are used as separate export destinations. Examples can be found after running DTS container under its configuration folder (**/opt/mellanox/doca/services/telemetry/config/fluent_bit_configs/**).

All **.exp** files are documented in-place.

```
DPU# ls -l /opt/mellanox/doca/services/telemetry/config/fluent_bit_configs/
/opt/mellanox/doca/services/telemetry/config/fluent_bit_configs/:
total 56
-rw-r--r-- 1 root root  528 Oct 11 07:52 es.exp
-rw-r--r-- 1 root root  708 Oct 11 07:52 file.exp
-rw-r--r-- 1 root root 1135 Oct 11 07:52 forward.exp
-rw-r--r-- 1 root root  719 Oct 11 07:52 influx.exp
-rw-r--r-- 1 root root  571 Oct 11 07:52 stdout.exp
-rw-r--r-- 1 root root  578 Oct 11 07:52 stdout_raw.exp
-rw-r--r-- 1 root root 2137 Oct 11 07:52 ufm_enterprise.fset
```

Fluent Bit **.exp** files have 2-level data routing:

- **source_tags** in **.exp** files (documented in-place)
- Token-based filtering governed by **.fset** files (documented in **ufm_enterprise.fset**)

To run with Fluent Bit exporter, set **enable=1** in required **.exp** files and set the environment variables before running the application:

```
export FLUENT_BIT_EXPORT_ENABLE=1
export FLUENT_BIT_CONFIG_DIR=/path/to/fluent_bit_configs
export LD_LIBRARY_PATH=/opt/mellanox/collectx/lib
```

5.4.18.5.4 Prometheus Endpoint

Prometheus exporter sets up endpoint (HTTP server) which keeps the most recent events data as text records.

The Prometheus server can scrape the data from the endpoint while the DOCA-Telemetry-Exporter-API-based application stays active.

Check the generic example of Prometheus records:

```
event_name_1{label_1="label_1_val", label_2="label_2_val", label_3="label_3_val", label_4="label_4_val"}
counter_value_1 timestamp_1
event_name_2{label_1="label_1_val", label_2="label_2_val", label_3="label_3_val", label_4="label_4_val"}
counter_value_2 timestamp_2
...
```

Labels are customizable metadata which can be set from data file. Events names could be filtered by token-based name-match according to **.fset** files.

Set the following environment variables before running.

```
# Set the endpoint host and port to enable export.
export PROMETHEUS_ENDPOINT=http://0.0.0.0:9101

# Set the folder which contains fset-files. If set, Prometheus will scrape
# only filtered data according to fieldsets.
export PROMETHEUS_CSET_DIR=/path/to/prometheus_cset
```

To scrape the data without the Prometheus server, use:

```
curl -s http://0.0.0.0:9101/metrics
```

Or:

```
curl -s http://0.0.0.0:9101/{fset_name}
```

5.4.18.5.5 Prometheus Remote Write

The Prometheus remote write exporter sends time-series metrics to a remote Prometheus receiver using the Prometheus remote-write protocol.

To use the Prometheus Remote-Write exporter, set the `CLX_REMOTE_WRITE_RECEIVER` environment variable to the receiver's address before running:

```
export CLX_REMOTE_WRITE_RECEIVER=http://$IP:$PORT/api/v1/write   # set the IP and PORT with the receiver address
(default port is 9502)
```

Replace `$IP` and `$PORT` with the receiver's IP address and port (default port is 9090).

5.4.18.5.6 Open Telemetry

Open Telemetry exporter streams telemetry to remote OTLP receiver. The supported transport is http.

To run with Open Telemetry exporter, set the following environment variables before running.

```
export CLX_OPEN_TELEMETRY_RECEIVER=http://$IP:$PORT/v1/metrics   # set the IP and PORT with the receiver address
(default port is 9502)
```

5.4.18.6 DOCA Telemetry Exporter Samples

This section provides DOCA Telemetry Exporter sample implementations on top of the BlueField DPU.

The telemetry exporter samples in this document demonstrate an initial recommended configuration that covers two use cases:

- Standard DOCA Telemetry Exporter data
- DOCA Telemetry Exporter for NetFlow data

The telemetry exporter samples run on BlueField. If write-to-file is enabled, telemetry data is stored to BlueField's storage. If inter-process communication (IPC) is enabled, data is sent to the DOCA Telemetry Service (DTS) running on the same BlueField.

For information on initializing and configuring DTS, refer to DOCA Telemetry Service Guide.

> All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

5.4.18.6.1 Running the Sample

1. Refer to the following documents:
   - DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - DOCA Troubleshooting for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

   ```
   cd /opt/mellanox/doca/samples/doca_telemetry_exporter/<sample_name>
   meson /tmp/build
   ninja -C /tmp/build
   ```

   > The binary **doca_<sample_name>** will be created under **/tmp/build/** .

3. Sample (e.g., **telemetry_export**) usage:

   ```
   Usage: doca_telemetry_export [DOCA Flags]

   DOCA Flags:
     -h, --help                        Print a help synopsis
     -v, --version                     Print program version information
     -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
   30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
     --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
   20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
     -j, --json <path>                 Parse command line flags from an input json file
   ```

4. For additional information per sample, use the **-h** option:

   ```
   /tmp/build/doca_<sample_name> -h
   ```

5.4.18.6.2 Samples

> These samples are also available on GitHub.

5.4.18.6.2.1 Telemetry Export

This sample illustrates how to use the telemetry exporter API. The sample uses a custom schema for telemetry exporter.

The sample logic includes:

1. Configuring schema attributes.
2. Initializing schema.
3. Creating telemetry exporter source.
4. Creating example events.
5. Reporting example events via DOCA Telemetry Exporter.
6. Destroying source and schema.

Reference:

- **/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export/telemetry_export_sample.c**
- **/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export/telemetry_export_main.c**
- **/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export/meson.build**

5.4.18.6.2.2 Telemetry Export NetFlow

This sample illustrates how to use the NetFlow functionality of the telemetry exporter API.

The sample logic includes:

1. Configuring NetFlow attributes.
2. Initializing NetFlow.
3. Creating telemetry exporter source.
4. Starting NetFlow.
5. Creating example events.
6. Reporting example events via DOCA Telemetry Exporter.
7. Destroying NetFlow.

Reference:

- **/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export_netflow/telemetry_export_netflow_sample.c**
- **/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export_netflow/telemetry_export_netflow_main.c**
- **/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export_netflowt/meson.build**

5.4.18.6.2.3 Telemetry Export Metrics

This sample illustrates how to use the Metrics API of the telemetry exporter to collect and export performance metrics (counters, gauges) with multi-dimensional labels.

The sample logic includes:

1. Initializing the telemetry exporter schema and source.
2. Creating and configuring the metrics context.
3. Setting flush intervals and adding constant labels.
4. Defining dynamic label sets (e.g., for network interfaces).
5. Reporting counter (absolute and incremental) and gauge metrics.
6. Flushing and destroying the metrics context, source, and schema.

Reference:

- `/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export_metrics/telemetry_export_metrics_sample.c`
- `/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export_metrics/telemetry_export_metrics_main.c`
- `/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export_metrics/meson.build`

5.4.18.6.2.4 Telemetry Export OTLP Logs

This sample illustrates how to use the OTLP Logs API of the telemetry exporter to collect and export structured log events as OpenTelemetry Protocol (OTLP) log records with resources, scopes, and typed attributes.

The sample logic includes:

1. Initializing the telemetry exporter schema and source.
2. Enabling opaque events (required for OTLP logs).
3. Creating and configuring the OTLP logs context.
4. Creating a resource and adding resource attributes.
5. Defining an event schema with attribute keys and types.
6. Creating a scope (instrumentation library) with default severity.
7. Writing log events with default and explicit severity levels.
8. Flushing and destroying the logs context, source, and schema.

Reference:

- `/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export_otlp_logs/telemetry_export_otlp_logs_sample.c`
- `/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export_otlp_logs/telemetry_export_otlp_logs_main.c`
- `/opt/mellanox/doca/samples/doca_telemetry_exporter/telemetry_export_otlp_logs/meson.build`

5.4.19 DOCA Telemetry

This guide provides instructions for building and developing applications that require telemetry data collection, supported by the NVIDIA® BlueField and NVIDIA® ConnectX® families of networking platforms.

5.4.19.1 Introduction

> The quality status of DOCA libraries is listed here.

DOCA Telemetry includes a suite of specialized libraries, each designed to access a specific hardware feature or function of a device. Each telemetry library provides a dedicated interface and context, allowing users to gather metrics related to the corresponding operation.

5.4.19.2 Supported Libraries

DOCA Telemetry supports the following libraries:

- DOCA Telemetry Adp Retx – Adaptive Retransmission telemetry
- DOCA Telemetry Diag – High-frequency diagnostic telemetry
- DOCA Telemetry DPA – Data Path Accelerator telemetry
- DOCA Telemetry PCC – Programmable Congestion Control telemetry
- DOCA Telemetry PCI – Peripheral Component Interconnect telemetry
- DOCA Telemetry PHY – Physical layer telemetry

5.4.19.3 DOCA Telemetry Adp Retx

This guide provides instructions for building and developing applications that require telemetry data collection from NVIDIA® BlueField® and NVIDIA® ConnectX® families of networking platforms.

5.4.19.3.1 Introduction

The **doca_telemetry_adp_retx** library provides statistics on Adaptive Retransmission Algorithm timeouts that have been configured on a given DOCA device, corresponding to an NVIDIA® BlueField® or NVIDIA® ConnectX® network card.

The library includes mechanisms for configuring and reading Adaptive Retransmissions in a histogram format. Each histogram read provides a series of bins, where each bin corresponds to a specific time range. The value of the bin is a count of the retransmissions that occurred due to a timeout falling within that time range.

The histogram can return information about events on all QPs of functions associated with the DOCA device, or it can be configured to track the QPs of a single VHCA ID.

5.4.19.3.2 Prerequisites

To utilize DOCA Telemetry Adp Retx, your system must meet the following baseline requirements:

- Firmware: Version **>=28/32/40.43.1000** is required for ConnectX-7, BlueField-3, and ConnectX-8 devices.
- Driver: The **fwctl** driver must be fully installed and actively loaded on the system.

5.4.19.3.2.1 Verifying the fwctl Driver

To verify that the **fwctl** driver is successfully loaded, check the device directories:

```
$ ls /sys/class/fwctl/
$ ls /dev/fwctl
```

The expected output for a standard 2-port device is **fwctl0 fwctl1**.

5.4.19.3.2.2 Manually Loading the Driver

If the directories **/sys/class/fwctl** or **/dev/fwctl** do not exist or are empty, the module may be installed but inactive.

Check for the module's presence:

```
$ grep fwctl -R /lib/modules/$(uname -r)/
```

If the output confirms the presence of **fwctl.ko** and **mlx5_fwctl.ko**, manually load the module and verify its status:

```
$ sudo modprobe mlx5_fwctl
$ lsmod | grep fwctl
```

5.4.19.3.2.3 Reinstalling the DOCA Host Package

If you cannot locate the installed **fwctl** module while manually loading the driver, or if the **modprobe** command fails to load it successfully, you must reinstall the DOCA Host package.

1. Download the package (DOCA 3.3.0 example):

   ```
   $ wget https://www.mellanox.com/downloads/DOCA/DOCA_v3.3.0/host/doca-host_3.3.0-088000-26.01-ubuntu2204_amd64.deb
   ```

2. Purge existing DOCA and OFED modules:

   ```
   $ sudo for f in $( dpkg --list | grep doca | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
   $ sudo for f in $( dpkg --list | grep mlnx | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
   $ sudo for f in $( dpkg --list | grep dpdk | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
   $ sudo for f in $( dpkg --list | grep ofed | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
   $ sudo /usr/sbin/ofed_uninstall.sh --force
   $ sudo apt-get autoremove
   ```

3. Install the new package and restart services:

   ```
   $ sudo dpkg -i doca-host_3.3.0-088000-26.01-ubuntu2204_amd64.deb
   $ sudo apt-get update
   $ sudo apt-get -y install doca-all
   $ sudo /etc/init.d/openibd restart
   ```

   Once the reinstallation is complete, confirm the module is successfully loaded according to section "DOCA Telemetry Adp Retx#Verifying the fwctl Driver".

5.4.19.3.3 Environment

DOCA Telemetry-based applications can run on either the host machine (ConnectX-7 or BlueField-3 and newer) or on the DPU (BlueField-3 and newer).

5.4.19.3.4 Architecture

The **doca_telemetry_adp_retx** library provides statistics on Adaptive Retransmission configured devices, including the number of retransmissions and their timeout ranges in a histogram format.

To interact with a device (typically corresponding to a specific NIC port), you must create a `doca_telemetry_adp_retx` context using `doca_telemetry_adp_retx_create()`.

5.4.19.3.5 Configuration Phase

5.4.19.3.5.1 Device Support

A DOCA device is required for the library to operate. For guidance on selecting a device, refer to the "DOCA Core Device Discovery" documentation.

Device support for `doca_telemetry_adp_retx` and its features can be checked with the following capability calls:

- `doca_telemetry_adp_retx_cap_is_supported()`
- `doca_telemetry_adp_retx_cap_histogram_is_supported()`

The maximum number of bins and the supported time units can be queried using:

- `doca_telemetry_adp_retx_cap_get_hist_max_bins()`
- `doca_telemetry_adp_retx_cap_get_hist_time_units()`

5.4.19.3.5.2 Histogram Configuration

The histogram divides retransmission events into bins, each representing a time range. If a retransmission timeout falls within a bin's range, that bin's counter is incremented. The number of bins and their time ranges are configurable.

The bin widths and timespans are determined by five main configuration options:

API Configuration	Description
`doca_telemetry_adp_retx_set_hist_num_bins()`	Number of bins to use in the histogram
`doca_telemetry_adp_retx_set_hist_bin0_width()`	Width (in time units) of the first bin
`doca_telemetry_adp_retx_set_hist_bin1_width()`	Width (in time units) of the second bin; also used as the base for calculating subsequent bins
`doca_telemetry_adp_retx_set_hist_time_unit()`	The time unit for bin0 and bin1 widths (e.g., `nsec`, `usec`, `msec`)
`doca_telemetry_adp_retx_set_hist_bin_width_node()`	The calculation mode for bins after bin1: either `fixed` (same width as bin1) or `double` (exponentially doubling)

Example:

- Fixed Mode: 4 bins, `bin0_width` =50, `bin1_width` =100, `time_unit` =msec, `width_mode` =fixed.
  - Bin 0: 0-50 msec
  - Bin 1: 50-150 msec (base + 100)
  - Bin 2: 150-250 msec (base + 100)

- Bin 3: 250-350 msec (base + 100)

Bin 0	Bin 1	Bin 2	Bin 3
Timeout Range:	***Timeout Range:***	***Timeout Range:***	***Timeout Range:***
0 msec - 49 msec	50 msec - 149 msec	150 msec - 249 msec	> 250 msec
(bin0_width)	*(bin1_width)*	*(bin1_width)*	

- Double Mode: 5 bins, `bin0_width` =50, `bin1_width` =100, `time_unit` =msec, `width_mode` =double.
  - Bin 0: 0-50 msec
  - Bin 1: 50-150 msec (base + 100)
  - Bin 2: 150-350 msec (base + 200)
  - Bin 3: 350-750 msec (base + 400)
  - Bin 4: 750-1550 msec (base + 800)

Bin 0	Bin 1	Bin 2	Bin 3	Bin 4
Timeout Range:	***Timeout Range:***	***Timeout Range:***	***Timeout Range:***	***Timeout Range:***
0 msec - 49 msec	50 msec - 149 msec	150 msec - 349 msec	350 msec - 749 msec	> 750 msec
(bin0_width)	*(bin1_width)*	*(bin1 x 2)*	*(bin2 x 2)*	

Further options control how the histogram is populated:

API Configuration	Description
`doca_telemetry_adp_retx_set_hist_vhca_id()`	Populates the histogram with retransmissions from a single VHCA ID only
`doca_telemetry_adp_retx_set_hist_clear_on_read()`	Clears (resets to 0) the histogram bin counters after each read
`doca_telemetry_adp_retx_set_hist_count_enable()`	Enables the counters. This must be set for the histogram to start gathering statistics.

5.4.19.3.6 Execution Phase

After configuration, the histogram is loaded and begins running on the device when `doca_telemetry_adp_retx_start()` is called. The bin counters can then be read from the device.

Shared Resource

`doca_telemetry_adp_retx` contexts do not have sole ownership or a locking mechanism on the device histogram. It is possible for another process to update the histogram's configuration while your context is in the execution phase, which can lead to misinterpretation of the bin counters.

The user is responsible for ensuring sole ownership of the histogram and verifying data integrity. An API function is provided to help detect these external changes.

The following functions are used during the execution phase:

API Datapath Functions	Description
`doca_telemetry_adp_retx_read_hist_bins()`	Reads the configured N histogram bin counters as an array of N 64-bit values
`doca_telemetry_adp_retx_detect_hist_conf_change()`	Indicates if the device's active histogram configuration matches the one defined in the context

5.4.19.3.7 State Machine

This section outlines the states of the `doca_telemetry_adp_retx` context.

5.4.19.3.7.1 Idle

The context has been created and is Idle.

In this state, it is expected for the application to:

- Destroy the context.
- Start the context for processing.

Allowed operations:

- Configuring the context according to section "Configuration".

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop

5.4.19.3.7.2 Running

In this state it is expected for the application to:

- Stop the context.

Allowed operations:

- Reading data from the device according to section "Execution".

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Successfully start the context

5.4.19.3.8 Alternative Datapath Options

DOCA Telemetry Adp Retx supports only CPU-based datapaths.

5.4.19.3.9 DOCA Telemetry Adp Retx Sample

The **doca_telemetry_adp_retx** sample demonstrates how to configure the histogram from command-line arguments, run for a set period, and then print the values of the configured bin counters. This sample is also available on GitHub.

5.4.19.3.9.1 Running the Sample

- Before you begin, refer to the following documents:
  - *DOCA Installation Guide for Linux*: For details on installing BlueField-related software.
  - *NVIDIA BlueField Platform Software Troubleshooting Guide*: For any issues with installation, compilation, or execution.
- To build a given sample:

```
# Update path if you downloaded from GitHub
cd /opt/mellanox/doca/samples/doca_telemetry/telemetry_adp_retx
meson /tmp/build
ninja -C /tmp/build
```

  The binary **doca_telemetry_adp_retx** is created under **/tmp/build/**.
- Sample usage:

```
Usage: doca_telemetry_adp_retx [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse all command flags from an input json file

Program Flags:
  -p, --pci-addr                    DOCA device PCI device address
  -u, --time-unit                   Time unit to use - 'nsec', 'usec', 'usec_100', or 'msec'
  -w, --width-mode                  Bin width mode to use - 'fixed', or 'double'
  -n, --number-bins                 The number of bins to configure the histogram for
  -vid, --vhca-id                   VHCA ID to get histogram events from
  -b0, --bin-0-width                Width of bin 0 to configure histogram
  -b1, --bin-1-width                Width of bin 1 to configure histogram
  -t, --wait-time                   Time in seconds to wait before reading histogram bins
```

The sample includes:

1. Locates and opens a DOCA device.
2. Creates a **doca_telemetry_adp_retx** instance.
3. Queries the device for histogram support, max bins, and time unit capabilities.
4. Configures the histogram with the values provided via command line (number of bins, bin widths, time unit, width mode, VHCA ID, clear on read, and counter enable).
5. Waits for the specified time, then reads and displays the value of each bin.
6. Destroys the **doca_telemetry_adp_retx** context.

5.4.19.4 DOCA Telemetry Diag

This guide provides instructions on building and developing applications which require collecting telemetry information provided by NVIDIA® BlueField® and NVIDIA® ConnectX® families of networking platforms.

5.4.19.4.1 Introduction

doca_telemetry_diag provides programable access to an on-device mechanism which allows sampling of diagnostic data (e.g., statistics and counters). **doca_telemetry_diag** allows configuring such parameters as required data IDs or sampling period, and retrieving the generated information in several formats.

5.4.19.4.2 Prerequisites

To utilize DOCA Telemetry Diag, your system must meet the following baseline requirements:

- Firmware: Version **>=28/32/40.43.1000** is required for ConnectX-7, BlueField-3, and ConnectX-8 devices.
- Driver: The **fwctl** driver must be fully installed and actively loaded on the system.

5.4.19.4.2.1 Verifying the fwctl Driver

To verify that the **fwctl** driver is successfully loaded, check the device directories:

```
$ ls /sys/class/fwctl/
$ ls /dev/fwctl
```

The expected output for a standard 2-port device is **fwctl0 fwctl1**.

5.4.19.4.2.2 Manually Loading the Driver

If the directories **/sys/class/fwctl** or **/dev/fwctl** do not exist or are empty, the module may be installed but inactive.

Check for the module's presence:

```
$ grep fwctl -R /lib/modules/$(uname -r)/
```

If the output confirms the presence of **fwctl.ko** and **mlx5_fwctl.ko**, manually load the module and verify its status:

```
$ sudo modprobe mlx5_fwctl
$ lsmod | grep fwctl
```

5.4.19.4.2.3 Reinstalling the DOCA Host Package

If you cannot locate the installed **fwctl** module while manually loading the driver, or if the **modprobe** command fails to load it successfully, you must reinstall the DOCA Host package.

1. Download the package (DOCA 3.3.0 example):

```
$ wget https://www.mellanox.com/downloads/DOCA/DOCA_v3.3.0/host/doca-host_3.3.0-088000-26.01-ubuntu2204_amd64.deb
```

2. Purge existing DOCA and OFED modules:

```
$ sudo for f in $( dpkg --list | grep doca | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
$ sudo for f in $( dpkg --list | grep mlnx | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
$ sudo for f in $( dpkg --list | grep dpdk | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
$ sudo for f in $( dpkg --list | grep ofed | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
$ sudo /usr/sbin/ofed_uninstall.sh --force
$ sudo apt-get autoremove
```

3. Install the new package and restart services:

```
$ sudo dpkg -i doca-host_3.3.0-088000-26.01-ubuntu2204_amd64.deb
$ sudo apt-get update
$ sudo apt-get -y install doca-all
$ sudo /etc/init.d/openibd restart
```

Once the reinstallation is complete, confirm the module is successfully loaded according to section "Verifying the fwctl Driver".

5.4.19.4.3 Environment

DOCA Telemetry-based applications can run either on the host machine (ConnectX-7 or BlueField-3 and above) or on the DPU target (BlueField-3 and above).

DOCA Telemetry can only be run inside the DPU (on Arm) with the BlueField configured in DPU mode, as described in *BlueField Modes of Operation*.

5.4.19.4.4 Architecture

Diagnostic data is stored in firmware as a cyclic buffer of samples. Each sample represents the values of all the requested diagnostic data IDs and their corresponding sampling timestamps.

5.4.19.4.4.1 Device and Ownership

The DOCA telemetry library requires a ConnectX/BlueField DOCA device to sample from. The device can be accessed using any of its physical functions (PFs).

> ⚠ If multiple devices exist in a setup, a **doca_telemetry_diag** context should be created for each.

The device supports two ownership models:

1. Single domain

2. Multiple domains

These impact how the library can be used, but do not impact the underlying hardware resources. The Amount of hardware resources available stays the same in both modes.

> ⚠ Configuring the ownership model can be performed using: DOCA Management

Single Domain Ownership Model

In this ownership model only one domain exists per device. Upon creation, the `doca_telemetry_diag` context assumes control of the associated hardware resources to prevent conflicts and ensure accurate data sampling. In rare instances, ownership may be overridden (e.g., if a process crashes before releasing ownership).

> ℹ The `force_ownership` parameter may be used when creating the context from a second process.

> ⚠ Once ownership is enforced for one PF, it cannot be claimed by a different PF. It is recommended to always use PF0 to prevent potential conflicts.

Multi Domain Ownership Model

In this ownership model a device can support multiple domains. Each `doca_telemetry_diag` context will use one of the available domains. This allows for the case where if the first domain does not use all the hardware resources then a second domain could be created to utilize the remaining resources. When in this mode upon creation, the `doca_telemetry_diag` context will check if the device can support an additional active domain and if not it will report a failure to acquire the resource. There are some technical limitations when using multi domain mode, the most notable of these are:

1. Synchronized start is not supported.
2. Some Data ID values require a hardware resource that cannot be shared between domains. If another domain attempts to use a Data ID which requires the occupied hardware resource, it will be rejected
3. The force ownership flag is ignored.
4. The achieved sampling period for a context can be impacted by any already started contexts on the same device.

> ℹ The multi domain ownership model prevents domains from being isolated when a process crashes or fails to cleanup the domain before exiting. Because of this and because multiple contexts are supported; forcefully taking ownership is no longer required and thus ignored.

> ⚠ It is recommended to always use PF0

5.4.19.4.5 Configuration Phase

5.4.19.4.5.1 Configuration

The context can be configured to match the application use case.

To discover if a certain configuration is supported, or what the min/max value for it is, refer to section "Device Support".

Sampling Mode

The DOCA Telemetry Diagnostics library supports the following operational sampling methods:

- Single sampling – Samples are collected and stored until the firmware buffer is filled, at which point sampling is automatically terminated.

  > You can manually restart the context to collect new samples, which will overwrite the previous data in the buffer.

- Repetitive sampling – Samples are continuously collected. When the firmware buffer fills, new samples automatically overwrite the oldest samples. This process continues indefinitely until the context is explicitly stopped.
- On demand – The device does not continuously collect samples in the background. Instead, the device fetches a single sample of the data strictly upon each manual query.

Considerations for Repetitive Sampling Mode

When configuring the DOCA Telemetry Diagnostics library to repetitive sampling, it is important to ensure that the firmware buffer is adequately sized to handle the data flow between hardware sampling and software retrieval. The firmware buffer size is dictated by the **log_max_num_samples** property.

1. Determine sampling rates:
   - Hardware sampling rate – the frequency at which the hardware collects data (e.g., every 100 µsec)
   - Software retrieval rate – the average time interval between successive data retrievals by the software (e.g., every 500 msec)

Calculate **AverageSamplesPerRetrieval** using the following equation:

We don't have a way to export this macro.

For example:

We don't have a way to export this macro.

1. To ensure smooth data processing and prevent data loss, the firmware buffer should be large enough to accommodate at least twice the average number of samples collected during the retrieval period:

   We don't have a way to export this macro.

   For example:

   We don't have a way to export this macro.
   samples

Moreover, the number of retrieved samples should be enlarged if the retrieving process may spike occasionally. For example, if the process time between retrieval calls is up to 6 times of the average, then the number of samples should be multiplied by 6+1=7.

Sampling Period

The sampling period can be configured using **doca_telemetry_diag_set_sample_period**.

In some cases, depending on the number and type of data IDs configured, the actual sampling period may be higher. The actual sampling period can be queried using **doca_telemetry_diag_get_sample_period** after configuring the data IDs.

After starting the **doca_telemetry_diag** context with **doca_telemetry_diag_start**, the actual sampling period may differ from the desired sampling rate. To address this, you can dynamically adjust the sampling period using **doca_telemetry_diag_reconfig_sample_period**.

Synchronized Start

Diagnostics data is sampled by the device every given sampling period. When sampling this way, each data entry in a sample may be recorded at a slightly different time.

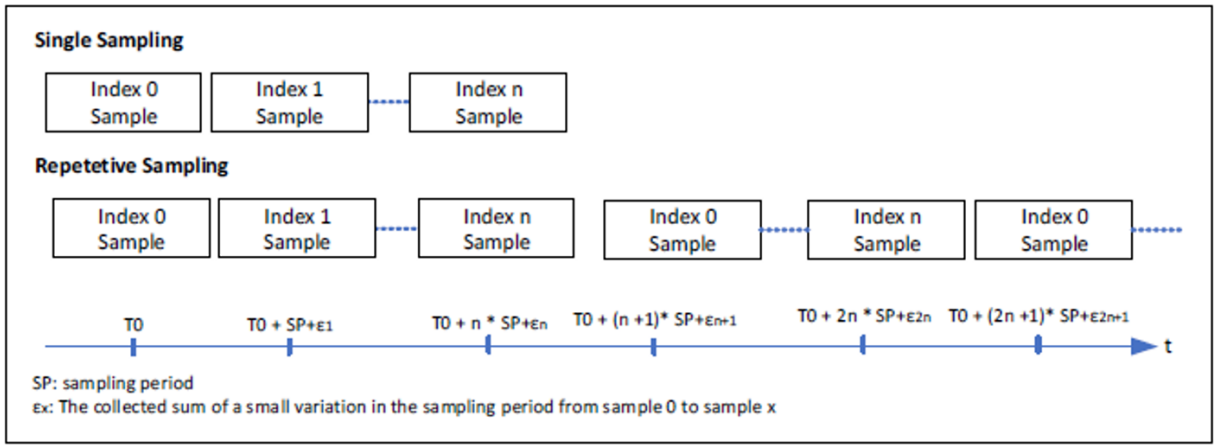
Synchronized start mode enables diagnostics counters to begin all data measurements at the same time (i.e., during the same clock cycle). This way, the sample period is guaranteed to be identical for all samples. Synchronized start diagnostic counters can be configured to be cleared at the beginning of each sampling period.

> ⚠ Not all data IDs can be sampled in synchronized start mode. See section "Data IDs" for additional details

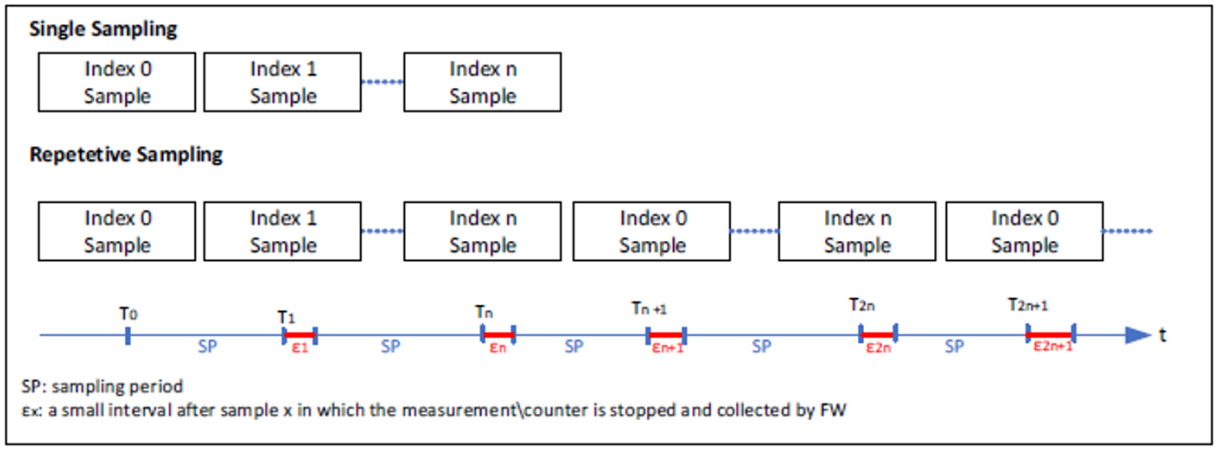
> ⚠ This is only supported when a device is configured to use the single domain ownership model

The following diagrams illustrate how synchronized start affects the sampling timeline:

Without Synched Start:



With Synched Start:



In synchronized start mode, counters are stopped during the collection time of each sample (illustrated in red in the diagram). If the application is required to normalize the counter to time, the actual sample period should be considered.

For example, if the **global_icmc_hit** (GIH) counter is sampled and the sample period is 100 µsec, then the **global_icmc_hit** per second, should be calculated as follows:

We don't have a way to export this macro.

Data IDs

The on-device mechanism provides the following diagnostic data classes:

- Counter – monotonically increasing and counting different events in the device.
  - If **doca_telemetry_diag_set_data_clear** is set, the counters are cleared at the beginning of each sampling period (valid only if synchronized start mode is used and operational mode is set to single or repetitive sampling).
- Statistic – other collected diagnostic data about the performance of the device. Statistic diagnostic data is cleared on each sample.

Each diagnostic data is represented by a unique identifier, the data ID. Appendix "List of Supported Data IDs" lists the currently supported data IDs.

After applying the configuration, the list of data IDs to be sampled should be applied by calling `doca_telemetry_diag_apply_counters_list_by_id`.

> **Data ID Configuration Constraints**
>
> - Configuration failures – Not all combinations of data IDs can be configured simultaneously. If the `doca_telemetry_diag_apply_counters_list_by_id` operation fails, it will return the index of the faulty data ID and the reason for the failure. You can retry the operation after omitting the faulty ID.
> - Synchronized start support – Not all data IDs support synchronized start mode. If you configure synchronized start and the apply operation fails with `DOCA_ERROR_BAD_CONFIG`, the specified data ID does not support this mode.

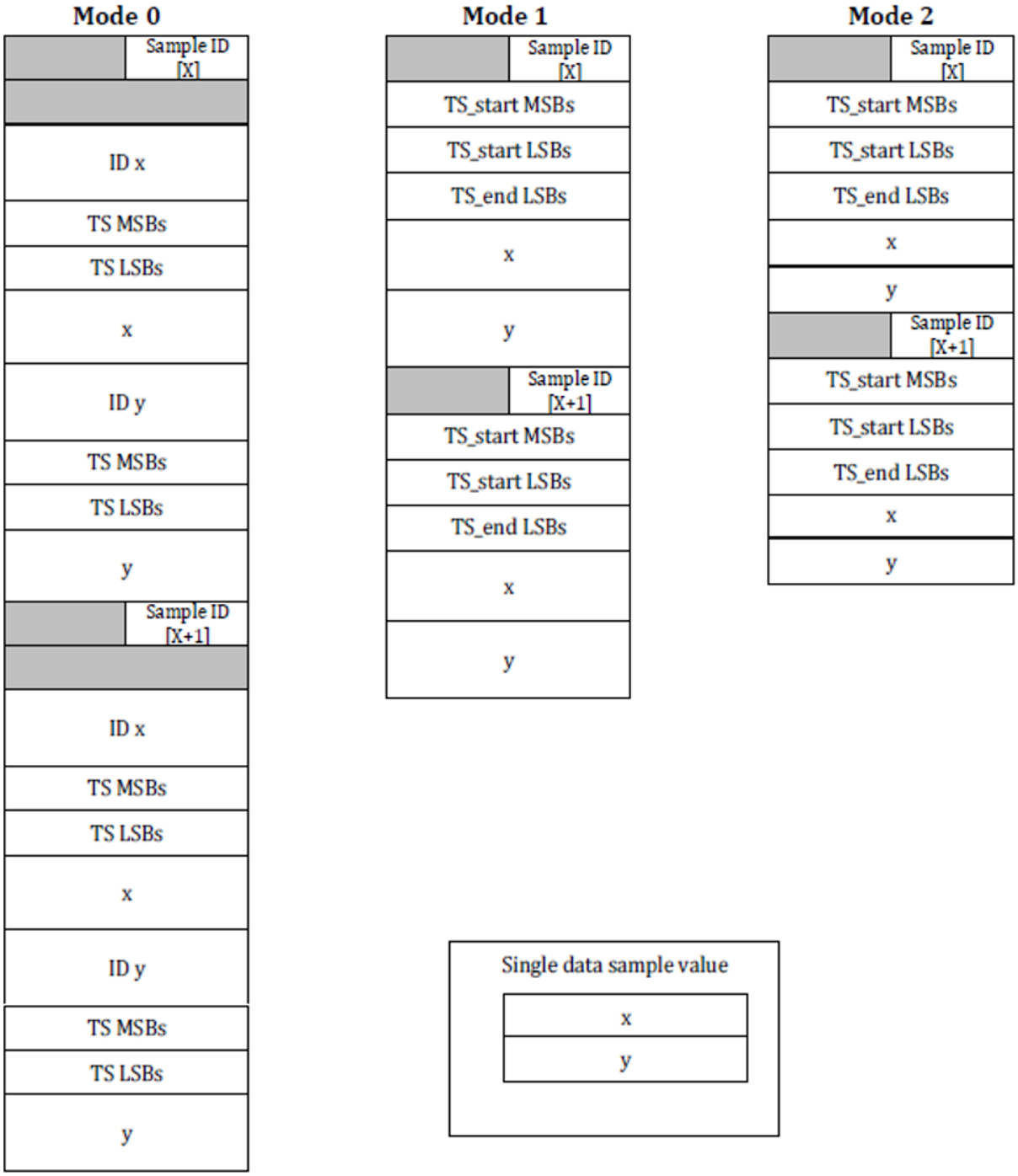
Output Formats

The `doca_telemetry_diag` context supports the following layout modes for sampled data:

Layout Mode	Data ID Presence	Data Size	Timestamp Granularity
Mode 0	Included in output	64 bits	Per data entry
Mode 1	Omitted from output	64 bits	Per sample (start and end)
Mode 2	Omitted from output	32 bits	Per sample (start and end)

> The order of the data IDs in the output strictly matches the order in which they were applied using `doca_telemetry_diag_apply_counters_list_by_id`.

The sample layout of these modes is illustrated in the following diagrams:



5.4.19.4.5.2 Device Support

DOCA Telemetry Diagnostics requires a device to operate. For picking a device, refer to "DOCA Core Device Discovery".

As device capabilities may change (see DOCA Core Device Support), it is recommended to select your device using the following method:

- **doca_telemetry_diag_cap_is_supported**

Some devices can allow different capabilities as follows:

- The maximum number of data IDs
- The maximum number of samples that can be held by the firmware
- Support for data clear
- Support for synchronized start
- Support for different sample modes
- Support for different timestamp sources

5.4.19.4.6 Execution Phase

Once the **doca_telemetry_diag** context is started, samples can be retrieved by calling the **doca_telemetry_diag_query_counters** function.

The library only retrieves new samples without duplications and returns fewer samples than requested if there are no more new samples.

There is a slight difference in behavior between the different sampling modes:

- In case of "On demand" sampling mode, each time the **doca_telemetry_diag_query_counters** is called, a new sample is captured and retrieved.
- For "Single" and "Repetitive" sampling modes, multiple samples can be retrieved in a single call. The application defines the maximum number of samples it wishes to retrieve and supplies a buffer large enough to contain these samples.

  > The sample size can be obtained using **doca_telemetry_diag_get_sample_size**.

  > The user should limit the number of requested samples according to the **log_max_num_samples** property.

- In case of "Single" sampling mode, once sampling is terminated, the user may call **doca_telemetry_diag_restart** to restart the sampling process, overwriting the previous results.
- During the execution phase, you can dynamically adjust the sampling period using **doca_telemetry_diag_reconfig_sample_period** to change the sampling rate. Note that this function has no effect in "On demand" sampling mode. For other sampling modes, it's recommended to verify the actual sampling period using **doca_telemetry_diag_get_sample_period** after reconfiguration to ensure it matches the expected value.

5.4.19.4.7 State Machine

The following section describes the different states the **doca_telemetry_diag** context goes through, how to move between states and what is allowed in each state.

5.4.19.4.7.1 Idle

The context is Idle and has ownership of a diagnostics data domain.

In this state it is expected for the application to:

- Destroy the context (releases the ownership).
- Apply a configuration, moving the context to "Configured" state.

Allowed operations:

- Configuring the context according to Configurations.

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Configured	Call stop
Ready	Call stop
Running	Call stop

5.4.19.4.7.2 Configured

In this state it is expected for application to:

- Apply the list of data IDs configuration using `doca_telemetry_diag_apply_counters_list_by_id`, moving the context to "Ready" state.

Allowed operations:

- Checking if a data ID is supported using `doca_telemetry_diag_check_data_id`
- Calling stop, moving it to "Idle" state

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Successfully apply the configuration, calling `doca_telemetry_diag_apply_config`.

5.4.19.4.7.3 Ready

All the necessary configuration was applied and the context is ready to start sampling.

In this state it is expected that application:

- Start the context, moving it to "Running" state.

Allowed operations:

- Calling stop, moving it to "Idle" state.

It is possible to reach this state as follows:

Previous State	Transition Action
Configured	Successfully apply the list of counters, calling `doca_telemetry_diag_apply_counters_list_by_id`.

5.4.19.4.7.4 Running

In this state samples are generated and can be retrieved.

In this state it is expected that application:

- Query the counters.

Allowed operations:

- For "Single" sample mode, restarting the context if needed.
- Calling stop, moving it to "Idle" state

It is possible to reach this state as follows:

Previous State	Transition Action
Ready	Successfully start the context

5.4.19.4.8 Alternative Datapath Options

DOCA Telemetry Diag supports only CPU-based datapaths.

5.4.19.4.9 DOCA Telemetry Diagnostics Sample

This section describes a telemetry diagnostics sample based on the `doca_telemetry` library.

The sample illustrates the utilization of DOCA telemetry diagnostics APIs to initialize and configure the `doca_telemetry_diag` context, as well as querying and parsing diagnostic counters.

5.4.19.4.9.1 Running the Sample

1. Refer to the following documents:
   a. DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   b. NVIDIA BlueField Platform Software Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

   ```
   cd /opt/mellanox/doca/samples/doca_telemetry/telemetry_diag
   meson /tmp/build
   ninja -C /tmp/build
   ```

   The binary `doca_telemetry_diag` is created under `/tmp/build/`.

Sample usage:

```
Usage: doca_telemetry_diag [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -p, --pci-addr                    DOCA device PCI device address
  -di, --data-ids                   Path to data ids JSON file
  -o, --output                      Output CSV file - default: "/tmp/out.csv"
  -rt, --sample-run-time            Total sample run time, in seconds
  -sp, --sample-period              Sample period, in nanoseconds
  -ns, --log-num-samples            Log max number of samples
  -sr, --max-samples-per-read       Max num samples per read
  -sm, --sample-mode                sample mode (0 - single, 1 - repetitive, 2 - on demand)
  -of, --output-format              output format (0 - output_format_0, 1 - output_format_1, 2 - output_format_2)
  -sym, --sync-mode                 sync mode (0 - sync_mode_no_sync, 1 - sync_mode_sync_start)
  -tss, --timestamp-source          timestamp source (0 - timestamp_source_frc, 1 - timestamp_source_rtc)
  -f, --force-ownership             Force ownership when creating context
  -e, --example-json-path           Generate an example json file with the default data_ids to the given path and
exit immediately. This file can be used as input later on. All other flags are ignored
  -rs, --reconfig-sample-period     Reconfig sample period, in nanoseconds. It means after the first sampling run
with the sample_period, if the reconfig_sample_period is non-0, before the current diag instance is stopped and
destroyed, the second sampling run will be done based on the reconfig_sample_period.
```

The sample logic includes:

1. Locating a DOCA device.
2. Initializing and configuring the **doca_telemetry_diag** instance.
3. Applying a list of data IDs to sample (either from a source JSON file or the default data IDs).
4. Starting the **doca_telemetry_diag** instance.
5. Allocating a buffer according to the sample size and amount of desired samples.
6. Querying the actual sampling period, after start.
7. Retrieving samples and writing the retrieved data to a ***.csv** file (either once or periodically).
8. If **reconfig_sample_period** is non-zero, retrieving samples and writing the sample file again based on the reconfigured sampling period.
9. Stopping the data IDs sampling.
10. Releasing all resources and destroying the context.

⚠ If the error message **cannot acquire ownership** is shown:

- In the single domain ownership model; you can use the command option **--force-ownership** to take the ownership (this may; reclaim an isolated context if a process failed to release it previously, or forcefully take control from an existing process. Use this option with caution)
- In the multiple domain ownership model; the maximum number of domains for this device are already allocated, One or more of the other domains must be released so they can be reused.

The sample can use data IDs given by the user using a JSON file. An example of the JSON file format can be created by using the **-e** flag on the sample, to export an example JSON file containing the default data IDs to a given path.

5.4.19.4.10 Visualization of Data IDs

The Ngauge tool is useful to visualize the data IDs prior to use or deployment in your application. Please see the Ngauge documentation for more information.

5.4.19.4.11 Appendix – List of Supported Data IDs

The following table lists the data IDs currently supported by DOCA:

Name	Description	Data Class	Data ID
port_rx_bytes	The number of received bytes on the physical port.[1]	Counter	0x10200001000000XX • XX – Local port
port_priority_rx_bytes	The number of received bytes on the physical port and priority.[1]	Counter	0x10200002000000YXX • XX – Local port • Y – Local port priority
port_rx_packets	The number of received packets on the physical port.[1]	Counter	0x10200003000000XX • XX – Local port
port_priority_rx_packets	The number of received packets on the physical port and priority.[1]	Counter	0x10200004000000YXX • XX – Local port • Y – local port priority
port_rx_discard_buf_packets	The number of received packets dropped due to lack of buffers on a physical port.	Counter	0x10200005000000XX • XX – Local port
port_priority_rx_pauses_packets	The number of link-layer pause packets received on a physical port and priority.	Counter	0x10200006000000YXX • XX – Local port • Y – Local port priority
port_rx_received_phy_clock_tie_ns	The TIE (Interval Time Error) in nanoseconds of the local clock compared to the peer node's physical clock as reference. Measured by comparing the received symbols on the physical link to the local physical clock. Format is 2's complement. > ***Note: the initial time for the TIE measurement could be before the data ID was configured.***	Counter	0x10200007000000XX • XX – Local port
host_rx_buffer_discards	The number of packets discarded due to no available data or descriptor buffers in the RX buffer, per host.	Counter	0x10400001000000XX • XX – Host ID

Name	Description	Data Class	Data ID
`host_rx_transport_pass_packets`	The number of packets that pass from the RX Transport to the Scatter engine, per host.	Counter	0x10800001000000XX • XX – Host ID
`host_rx_transport_out_of_buffer_packets`	The number of dropped packets due to a lack of WQE for the associated QPs/RQs (excluding hairpin QPs/RQs).	Counter	0x10800002000000XX • XX – Host ID
`host_rx_transport_out_of_buffer_hairpin_packets`	The number of dropped packets due to a lack of WQE for the associated hairpin QPs/RQs.	Counter	0x10800003000000XX • XX – Host ID
`port_rx_transport_ecn_packets`	The number of RoCEv2 packets received by the notification point which were marked for experiencing the congestion (i.e., ECN bits `11` on the ingress RoCE traffic), per port.	Counter	0x10800004000000XX • XX – Local port
`port_rx_transport_cnp_handled_packets`	The number of CNP received packets handled by the Reaction Point, per port.	Counter	0x10800005000000XX • XX – Local port
`port_tx_transport_cnp_sent_packets`	The number of CNP packets sent by the Notification Point, per port.	Counter	0x11000001000000XX • XX – Local port
`tx_transport_done_due_to_cc_deschedule_events`	The number of QP de-scheduled due to congestion control rate limitation.	Counter	0x11000002000000
`port_tx_bytes`	The number of transmitted bytes on the physical port (excluding loopback traffic).	Counter	0x11400001000000XX • XX – Local port
`port_priority_tx_bytes`	The number of transmitted bytes on the physical port and priority (excluding loopback traffic).	Counter	0x11400002000000YXX • XX – Local port • Y – Local port priority
`port_tx_packets`	The number of transmitted packets on the physical port (excluding loopback traffic).	Counter	0x11400003000000XX • XX – Local port
`port_priority_tx_packets`	The number of transmitted packets on the physical port and priority (excluding loopback traffic).	Counter	0x11400004000000YXX • XX – Local port • Y – Local port priority

Name	Description	Data Class	Data ID
port_priority_tx_pauses_packets	The number of link–layer pause packets transmitted on a physical port and priority.	Counter	0x1140000500000YXX • XX – Local port • Y – Local port priority
pcie_link_inbound_bytes	The number of bytes received from the PCIe toward the device, per PCIe link.	Counter	0x1160000100ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
pcie_link_outbound_bytes	The number of bytes transmitted from the device toward the PCIe, per PCIe link.	Counter	0x1160000200ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
pcie_link_inbound_data_bytes	The number of data bytes received from the PCIe (excluding headers) toward the device, per PCIe link.	Counter	0x1160000300ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
pcie_link_outbound_data_bytes	The number of data bytes transmitted from the device toward the PCI (excluding headers), per PCIe link.	Counter	0x1160000400ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
pcie_link_write_stalled_time_no_posted_data_credits_ns	The time period (in nanoseconds) in which the device had outbound posted write requests but stalled due to insufficient data credits per PCIe link.	Counter	0x1160000500ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
pcie_link_write_stalled_time_no_posted_header_credits_ns	The time period (in nanoseconds) in which the device had outbound posted write requests but stalled due to insufficient header credits per PCIe link.	Counter	0x1160000600ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)

Name	Description	Data Class	Data ID
pcie_link_read_stalled_time_no_non_posted_data_credits_ns	The time period (in nanoseconds) in which the device had outbound non-posted read requests but stalled due to insufficient data credits per PCIe link.	Counter	0x1160000700 ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
pcie_link_read_stalled_time_no_non_posted_header_credits_ns	The time period (in nanoseconds) in which the device had outbound non-posted read requests but stalled due to insufficient header credits per PCIe link.	Counter	0x1160000800 ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
pcie_link_read_stalled_time_no_completion_buffers_ns	The time period (in nanoseconds) in which the device had outbound non-posted read requests but stalled due to no NIC completion buffers per PCIe link.	Counter	0x1160000900 ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
pcie_link_tclass_read_stalled_time_ordering_ns	The time period (in nanoseconds) in which the device had outbound non-posted read requests but stalled due to PCIe ordering semantics per PCIe link and PCIe tclass.	Counter	0x1160000aZZ ZZYYXX • XX – Node • YY – PCIe index • ZZZZ – (tclass (0 – 7) << 6) | (Depth (0 – 63))
pcie_link_latency_total_read_ns	The total latency (in nanoseconds) for all PCIe read from the device per PCIe link.[2]	Statistic	0x1160000b00 ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
pcie_link_latency_total_read_packets	The total number of packets used for the pcie_link_latency_total_read_ns calculation.	Statistic	0x1160000c00 ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
pcie_link_latency_max_read_ns	The maximum latency (in nanoseconds) for a single PCIe read from the device per PCIe link.	Statistic	0x1160000d00 ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)

Name	Description	Data Class	Data ID
pcie_link_latency_min_read_ns	The minimum latency (in nanoseconds) for a single PCIe read from the device per PCIe link.	Statistic	0x1160000e00ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
mplhr_pcie_link_latency_read_hist_packets	The number of PCIe reads with latency in the range of the PCIe Latency Histogram bin per PCIe link. The sampled value will be the same as the MPLH corresponding bin entry. "Bin" is currently 0→9 (10 bins in total).		0x1160000fZZZZYYXX • XX – Node • YY – PCIe index • ZZZZ – (Bin (0 – 31) << 6) | (Depth (0 – 63))
global_completion_engine_rx_cqes	Number of responder (RX) CQEs.	Counter	0x10c0000100000000
function_completion_engine_rx_cqes	Number of RX CQEs per function.	Counter	0x10c0000200000XXXX • XXXX – **vhca_id**
global_completion_engine_tx_cqes	Number of requestor (TX) CQEs.	Counter	0x10c0000400000000
function_completion_engine_tx_cqes	Number of TX CQEs per function.	Counter	0x10c0000500000XXXX • XXXX – **vhca_id**
global_icmc_request	Number of accesses to ICMC.	Counter	0x1180000100000000
global_icmc_hit	Number of ICMC hits.	Counter	0x1180000200000000
global_icmc_miss	Number of ICMC misses.	Counter	0x1180000300000000
global_icmc_writes	Number of writes to ICMC	Counter	0x1180000400000000
global_icmc_reads	Number of reads to ICMC	Counter	0x1180000500000000

[1] This counter includes loopback traffic and does not include packets discarded due to FCS, frame size, and similar errors.

[2] Dividing this counter by **pcie_link_latency_total_read_packets** *yields the average PCIe read latency of those reads.*

5.4.19.4.12 DOCA Telemetry Diag for Windows

> ⚠ DOCA Telemetry Diag for Windows is irrelevant to DOCA Linux developers. It is installed as part of the DOCA package to allow for exporting telemetry diagnostics data to management software that is run in a Windows environment.

This guide provides instructions on building and developing applications which require collecting telemetry information provided by NVIDIA® BlueField® and NVIDIA® ConnectX® families of networking platforms.

5.4.19.4.12.1 1. Introduction

`doca_telemetry_diag` provides programable access to an on-device mechanism which allows sampling of diagnostic data (e.g., statistics and counters). `doca_telemetry_diag` allows configuring such parameters as required data IDs or sampling period, and retrieving the generated information in several formats.

1.1. Architecture

Diagnostic data is stored in firmware as a cyclic buffer of samples. Each sample represents the values of all the requested diagnostic data IDs and their corresponding sampling timestamps.

1.2. Device and Ownership

The DOCA telemetry library requires a ConnectX/BlueField DOCA device to sample from. The device can be accessed using any of its physical functions (PFs).

The device supports two ownership models:

1. Single domain
2. Multiple domains

1.2.1. Single Domain Ownership Model

In this ownership model only one domain exists per device. Upon creation, the `doca_telemetry_diag` context assumes control of the associated hardware resources to prevent conflicts and ensure accurate data sampling. In rare instances, ownership may be overridden (e.g., if a process crashes before releasing ownership).

> The `force_ownership` parameter may be used when creating the context from a second process.

> ⚠ Once ownership is enforced for one PF, it cannot be claimed by a different PF. It is recommended to always use PF0 to prevent potential conflicts.

1.2.2. Multi Domain Ownership Model

In this ownership model a device can support multiple domains. Each `doca_telemetry_diag` context will use one of the available domains. This allows for the case where

if the first domain does not use all the hardware resources then a second domain could be created to utilize the remaining resources. There are some technical limitations when using multi domain mode, the most notable of these are:

1. Synchronized start is not supported.
2. Some Data ID values require a hardware resource that cannot be shared between domains. If another domain attempts to use a Data ID which requires the occupied hardware resource, it will be rejected
3. The force ownership flag is ignored.
4. The achieved sampling period for a context can be impacted by any already started contexts on the same device.

If multiple devices exist in a setup, a **doca_telemetry_diag** context should be created for each individual domain.

WinOF-2 provides specific commands to switch between single-domain and multi-domain mode.

For WinOF-2 users, there is not any difference for a single **doca_telemetry_diag** instance to run under single-domain or multi-domain mode. User applications developed for single-domain mode does not need any change to run under multi-domain mode.

1.3. Configuration Phase

1.3.1. Configuration

The context can be configured to match the application use case.

To discover if a certain configuration is supported, or what the min/max value for it is, refer to section "Device Support".

1.3.1.1. Sampling Mode

The DOCA Telemetry Diagnostics library supports the following operational sampling methods:

- Single sampling – Samples are collected and stored until the firmware buffer is filled, at which point sampling is automatically terminated.
  You can manually restart the context to collect new samples, which will overwrite the previous data in the buffer.
- Repetitive sampling – Samples are continuously collected. When the firmware buffer fills, new samples automatically overwrite the oldest samples. This process continues indefinitely until the context is explicitly stopped.
- On demand – The device does not continuously collect samples in the background. Instead, the device fetches a single sample of the data strictly upon each manual query.

1.3.1.1.1. Considerations for Repetitive Sampling Mode

When configuring the DOCA Telemetry Diagnostics library to repetitive sampling, it is important to ensure that the firmware buffer is adequately sized to handle the data flow between hardware sampling and software retrieval. The firmware buffer size is dictated by the **log_max_num_samples** property.

1. Determine sampling rates:
    - Hardware sampling rate – the frequency at which the hardware collects data (e.g., every 100 µsec)
    - Software retrieval rate – the average time interval between successive data retrievals by the software (e.g., every 500 msec)

Calculate **AverageSamplesPerRetrieval** using the following equation:

We don't have a way to export this inline extension.

For example:

We don't have a way to export this inline extension.

1. To ensure smooth data processing and prevent data loss, the firmware buffer should be large enough to accommodate at least twice the average number of samples collected during the retrieval period:
   For example:

We don't have a way to export this inline extension.

Moreover, the number of retrieved samples should be enlarged if the retrieving process may spike occasionally. For example, if the process time between retrieval calls is up to 6 times of the average, then the number of samples should be multiplied by 6+1=7.

1.3.1.2. Sampling Period

The sampling period can be configured using **doca_telemetry_diag_set_sample_period**.

In some cases, depending on the number and type of data IDs configured, the actual sampling period may be higher. The actual sampling period can be queried using **doca_telemetry_diag_get_sample_period** after configuring the data IDs.

After starting the **doca_telemetry_diag** context with **doca_telemetry_diag_start**, the actual sampling period may differ from the desired sampling rate. To address this, you can dynamically adjust the sampling period using **doca_telemetry_diag_reconfig_sample_period**.

1.3.1.3. Synchronized Start

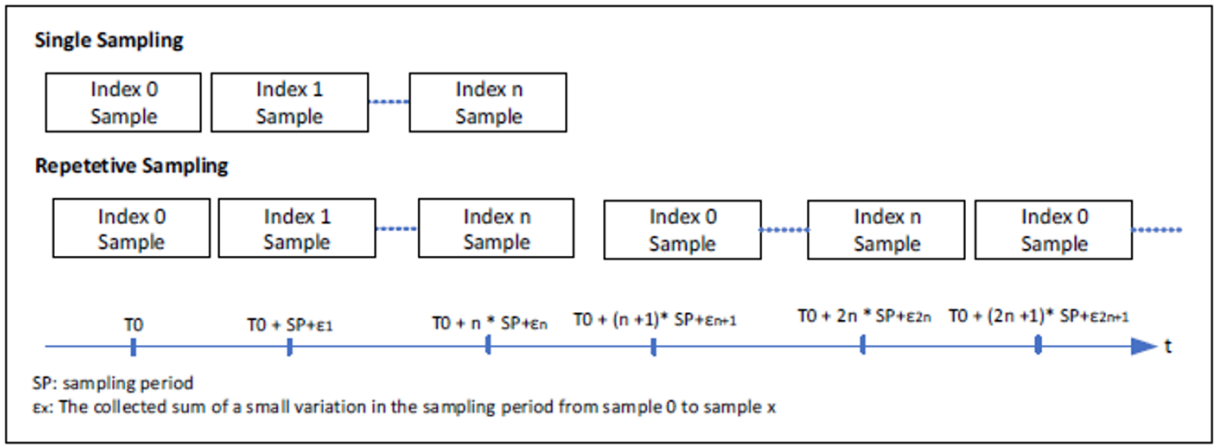
Diagnostics data is sampled by the device every given sampling period. When sampling this way, each data entry in a sample may be recorded at a slightly different time.

Synchronized start mode enables diagnostics counters to begin all data measurements at the same time (i.e., during the same clock cycle). This way, the sample period is guaranteed to be identical for all samples. Synchronized start diagnostic counters can be configured to be cleared at the beginning of each sampling period.

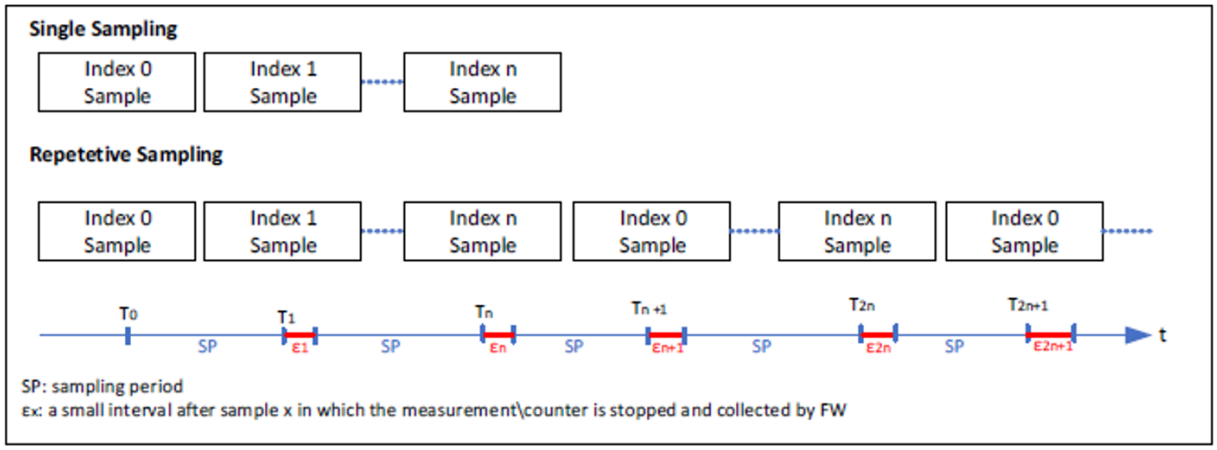
Not all data IDs can be sampled in synchronized start mode. See section "List of Supported Data IDs" for additional details.

The following diagrams illustrate how synchronized start affects the sampling timeline:

Without Synched Start:



With Synched Start:



In synchronized start mode, counters are stopped during the collection time of each sample (illustrated in red in the diagram). If the application is required to normalize the counter to time, the actual sample period should be considered.

For example, if the **global_icmc_hit** (GIH) counter is sampled and the sample period is 100 µsec, then the **global_icmc_hit** per second, should be calculated as follows:

We don't have a way to export this inline extension.

1.3.1.4. Data IDs

The on-device mechanism provides the following diagnostic data classes:

- Counter – monotonically increasing and counting different events in the device.

- If **doca_telemetry_diag_set_data_clear** is set, the counters are cleared at the beginning of each sampling period (valid only if synchronized start mode is used and operational mode is set to single or repetitive sampling).
- Statistic – other collected diagnostic data about the performance of the device. Statistic diagnostic data is cleared on each sample.

Each diagnostic data is represented by a unique identifier, the data ID. Appendix "List of Supported Data IDs" lists the currently supported data IDs.

After applying the configuration, the list of data IDs to be sampled should be applied by calling **doca_telemetry_diag_apply_counters_list_by_id**.

- Configuration failures – Not all combinations of data IDs can be configured simultaneously. If the **doca_telemetry_diag_apply_counters_list_by_id** operation fails, it will return the index of the faulty data ID and the reason for the failure. You can retry the operation after omitting the faulty ID.
- Synchronized start support – Not all data IDs support synchronized start mode. If you configure synchronized start and the apply operation fails with **DOCA_ERROR_BAD_CONFIG**, the specified data ID does not support this mode.

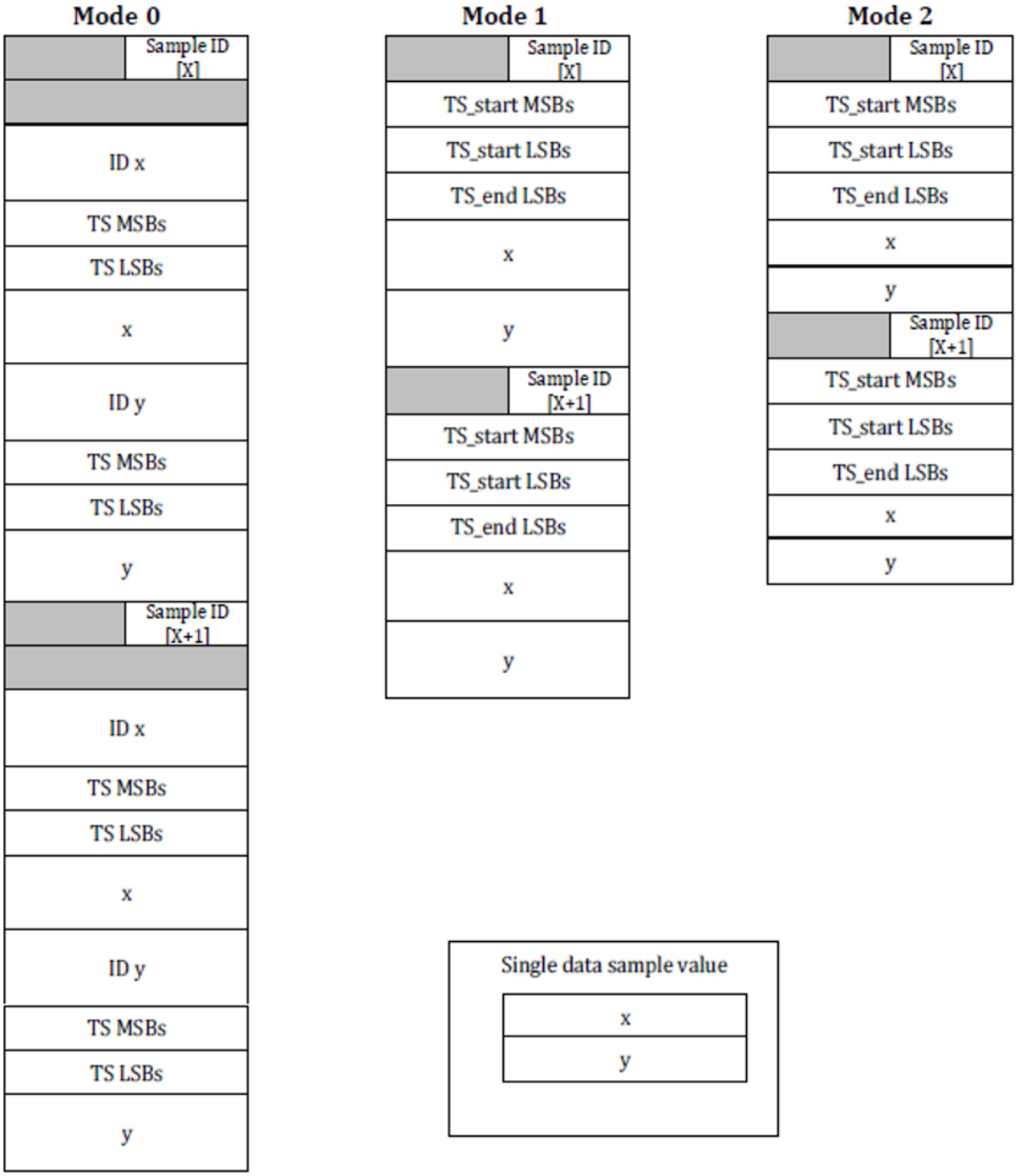
1.3.1.5. Output Formats

The **doca_telemetry_diag** context supports the following layout modes for sampled data:

Layout Mode	Data ID Presence	Data Size	Timestamp Granularity
Mode 0	Included in output	64 bits	Per data entry
Mode 1	Omitted from output	64 bits	Per sample (start and end)
Mode 2	Omitted from output	32 bits	Per sample (start and end)

The order of the data IDs in the output strictly matches the order in which they were applied using **doca_telemetry_diag_apply_counters_list_by_id**.

The sample layout of these modes is illustrated in the following diagrams:



1.3.2. Device Support

DOCA Telemetry Diagnostics requires a device to operate. For picking a device, refer to "DOCA Core Device Discovery".

As device capabilities may change (see DOCA Core Device Support), it is recommended to select your device using the following method:

- **doca_telemetry_diag_cap_is_supported**

Some devices can allow different capabilities as follows:

- The maximum number of data IDs
- The maximum number of samples that can be held by the firmware
- Support for data clear
- Support for synchronized start
- Support for different sample modes
- Support for different timestamp sources

1.4. Execution Phase

Once the `doca_telemetry_diag` context is started, samples can be retrieved by calling the `doca_telemetry_diag_query_counters` function.

The library only retrieves new samples without duplications and returns fewer samples than requested if there are no more new samples.

There is a slight difference in behavior between the different sampling modes:

- In case of "On demand" sampling mode, each time the `doca_telemetry_diag_query_counters` is called, a new sample is captured and retrieved.
- For "Single" and "Repetitive" sampling modes, multiple samples can be retrieved in a single call. The application defines the maximum number of samples it wishes to retrieve and supplies a buffer large enough to contain these samples.
- In case of "Single" sampling mode, once sampling is terminated, the user may call `doca_telemetry_diag_restart` to restart the sampling process, overwriting the previous results.
- During the execution phase, you can dynamically adjust the sampling period using `doca_telemetry_diag_reconfig_sample_period` to change the sampling rate. Note that this function has no effect in "On demand" sampling mode. For other sampling modes, it's recommended to verify the actual sampling period using `doca_telemetry_diag_get_sample_period` after reconfiguration to ensure it matches the expected value.

For "Single" and "Repetitive" sampling modes:

- The sample size can be obtained using `doca_telemetry_diag_get_sample_size`
- The user should limit the number of requested samples according to the `log_max_num_samples` property

1.5. State Machine

The following section describes the different states the `doca_telemetry_diag` context goes through, how to move between states and what is allowed in each state.

1.5.1. Idle

The context is Idle and has ownership of a diagnostics data domain.

In this state it is expected for the application to:

- Destroy the context (releases the ownership).
- Apply a configuration, moving the context to "Configured" state.

Allowed operations:

- Configuring the context according to section Configurations.

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Configured	Call stop
Ready	Call stop
Running	Call stop

1.5.2. Configured

In this state it is expected for application to:

- Apply the list of data IDs configuration using `doca_telemetry_diag_apply_counters_list_by_id`, moving the context to "Ready" state.

Allowed operations:

- Checking if a data ID is supported using `doca_telemetry_diag_check_data_id`
- Calling stop, moving it to "Idle" state

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Successfully apply the configuration, calling `doca_telemetry_diag_apply_config`.

1.5.3. Ready

All the necessary configuration was applied and the context is ready to start sampling.

In this state it is expected that application:

- Start the context, moving it to "Running" state.

Allowed operations:

- Calling stop, moving it to "Idle" state.

It is possible to reach this state as follows:

Previous State	Transition Action
Configured	Successfully apply the list of counters, calling `doca_telemetry_diag_apply_counters_list_by_id`.

1.5.4. Running

In this state samples are generated and can be retrieved.

In this state it is expected that application:

- Query the counters.

Allowed operations:

- For "Single" sample mode, restarting the context if needed.
- Calling stop, moving it to "Idle" state

It is possible to reach this state as follows:

Previous State	Transition Action
Ready	Successfully start the context

5.4.19.4.12.2 2. Prerequisites

The Windows system must meet the following baseline requirements:

2.1. WinOF-2 Package

The WinOF-2 package (containing the **doca_telemetry_diag** distribution) is installed or extracted

2.2. Environment

DOCA Telemetry Diag applications can run on the host machine (ConnectX-7 or BlueField-3 and above) for both x86_64 and ARM64 platform.

2.3. Acceptable Windows PCI Address Format

- Bus:Device.Function (XX:XX.X)
  - This format is identical to the Linux format
- Segment:Bus:Device.Function (XXXX:XX:XX.X)
  - The “segment“ is equivalent to the “domain“ in Linux
  - Thus this format is identical to the Linux format
- SegmentBus:Device.Function (XXXX:XX.X)
  - The value-of-SegmentBus = value-of-Segment * 256 + value-of-Bus
  - This is only for Windows
- For example, on a ARM64 Windows server, we can get the following available Mellanox devices PCI address:

```
> mlx5cmd -stat
the pci segment is part of the bus dword
NIC 1: physical_location=Bus 3, Device 0, Function 0
NIC 2: physical_location=Bus 515, Device 0, Function 0
NIC 3: physical_location=Bus 4099, Device 0, Function 0
NIC 4: physical_location=Bus 4611, Device 0, Function 0
```

- Then, users need to manually change decimal value to hex value:

Decimal Value	Segment:Bus Format	SegmentBus Format
515 = 2 * 256 + 3	0002:03	0203
4099 = 16 * 256 + 3	0010:03	1003
4611 = 18 * 256 + 3	0012:03	1203

- Thus, the acceptable address for the above 4 NICs are

NIC	XX:XX.X	XXXX:XX:XX.X	XXXX:XX.X
1	03:00.0	0000:03:00.0	0003:00.0
2	none	0002:03:00.0	0203:00.0
3	none	0010:03:00.0	1003:00.0
4	none	0012:03:00.0	1203:00.0

5.4.19.4.12.3 3. DOCA Telemetry Diagnostics Sample

This section describes a telemetry diagnostics sample based on the **doca_telemetry_diag** library.

The sample illustrates the usage of DOCA telemetry diagnostics APIs to initialize and configure the **doca_telemetry_diag** context, as well as querying and parsing diagnostic counters.

3.1. Building the Sample

After the WinOF-2 package is installed or extracted, the customer can find the **doca_telemetry_diag** sample code directory, and all the necessary DOCA header files.

Windows customers Only need to install WinOF-2 package, but do NOT need to install WinOF-2 DevX SDK.

3.1.1. Sample steps to build customer own json-c library

```
Open a Windows command terminal:
> git clone https://github.com/Microsoft/vcpkg.git
> cd vcpkg
> bootstrap-vcpkg.sh
> bootstrap-vcpkg.bat
> vcpkg.exe integrate install
> vcpkg.exe install json-c
```

There are many different ways to install json-c library for Windows, the above sample is just a demo based on the “Windows Server 2022 Datacenter” on x86_64 platform, it may be not suitable for other Windows platform.

3.1.2. Build Sample Command

```
> cd $CUSTOEMR_DOCA_TELMETRY_DIAG_SAMPLE_DIRECTORY
> meson setup build -Djson_c_root=$THE_ABOVE_JSON_C_DIRECTORY
> ninja -C build
```

The proper json_c_root must be provided.

3.2. Running the Sample

3.2.1. Sample Usage

```
Usage: doca_telemetry_diag [DOCA Flags] [Program Flags]
```

```
DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -p, --pci-addr                    DOCA device PCI device address
  -di, --data-ids                   Path to data ids JSON file
  -o, --output                      Output CSV file - default: "/tmp/out.csv"
  -rt, --sample-run-time            Total sample run time, in seconds
  -sp, --sample-period              Sample period, in nanoseconds
  -ns, --log-num-samples            Log max number of samples
  -sr, --max-samples-per-read       Max num samples per read
  -sm, --sample-mode                sample mode (0 - single, 1 - repetitive, 2 - on demand)
  -of, --output-format              output format (0 - output_format_0, 1 - output_format_1, 2 - output_format_2)
  -sym, --sync-mode                 sync mode (0 - sync_mode_no_sync, 1 - sync_mode_sync_start)
  -tss, --timestamp-source          timestamp source (0 - timestamp_source_frc, 1 - timestamp_source_rtc)
  -f, --force-ownership             Force ownership when creating context
  -e, --example-json-path           Generate an example json file with the default data_ids to the given path and
exit immediately. This file can be used as input later on. All other flags are ignored
  -rs, --reconfig-sample-period     Reconfig sample period, in nanoseconds. It means after the first sampling run
with the sample_period, if the reconfig_sample_period is non-0, before the current diag instance is stopped and
destroyed, the second sampling run will be done based on the reconfig_sample_period.
```

3.2.2. Sample Logic

1. Locating a DOCA device.
2. Initializing and configuring the **doca_telemetry_diag** instance.
3. Applying a list of data IDs to sample (either from a source JSON file or the default data IDs).
4. Starting the **doca_telemetry_diag** instance.
5. Allocating a buffer according to the sample size and amount of desired samples.
6. Querying the actual sampling period, after start.
7. Retrieving samples and writing the retrieved data to a ***.csv** file (either once or periodically).
8. If **reconfig_sample_period** is non-zero, retrieving samples and writing the sample file again based on the reconfigured sampling period.
9. Stopping the data IDs sampling.
10. Releasing all resources and destroying the context.

If the error message **cannot acquire ownership** is shown: you can use the command option **--force-ownership** to take the ownership (this may; reclaim an isolated context if a process failed to release it previously, or forcefully take control from an existing process. Use this option with caution)

The sample can use data IDs given by the user using a JSON file. An example of the JSON file format can be created by using the **-e** flag on the sample, to export an example JSON file containing the default data IDs to a given path.

3.2.3. Sample Running Commands with Different PCI Address

```
> doca_telemetry_diag.exe -p 03:00.0
[2026-04-30 15:43:04:801018][7640][DOCA][INF][CORE][doca_log.cpp:1281][_common_write_version_to_backend] DOCA
version 3.5.0011
[2026-04-30 15:43:04:802139][7640][DOCA][INF][TELEMETRY_DIAG][telemetry_diag_main.c:1012][main] Starting the sample
[2026-04-30 15:43:05:821181][7640][DOCA][INF][TELEMETRY_DIAG][telemetry_diag_main.c:1069][main] Sample finished
successfully

> doca_telemetry_diag.exe -p 1203:00.0
[2026-04-30 15:43:04:801018][7640][DOCA][INF][CORE][doca_log.cpp:1281][_common_write_version_to_backend] DOCA
version 3.5.0011
[2026-04-30 15:43:04:802139][7640][DOCA][INF][TELEMETRY_DIAG][telemetry_diag_main.c:1012][main] Starting the sample
[2026-04-30 15:43:05:821181][7640][DOCA][INF][TELEMETRY_DIAG][telemetry_diag_main.c:1069][main] Sample finished
successfully
```

5.4.19.4.12.4 4. Appendix – List of Supported Data IDs

The following table lists the data IDs currently supported by DOCA:

Name	Description	DataClass	Data ID
port_rx_bytes	The number of received bytes on the physical port.[1]	Counter	0x10200001000000XX • XX – Local port
port_priority_rx_bytes	The number of received bytes on the physical port and priority.[1]	Counter	0x10200002000000YXX • XX – Local port • Y – Local port priority
port_rx_packets	The number of received packets on the physical port.[1]	Counter	0x10200003000000XX • XX – Local port
port_priority_rx_packets	The number of received packets on the physical port and priority.[1]	Counter	0x10200004000000YXX • XX – Local port • Y – local port priority
port_rx_discard_buf_packets	The number of received packets dropped due to lack of buffers on a physical port.	Counter	0x10200005000000XX • XX – Local port
port_priority_rx_pauses_packets	The number of link-layer pause packets received on a physical port and priority.	Counter	0x10200006000000YXX • XX – Local port • Y – Local port priority
host_rx_buffer_discards	The number of packets discarded due to no available data or descriptor buffers in the RX buffer, per host.	Counter	0x10400001000000XX • XX – Host ID
host_rx_transport_pass_packets	The number of packets that pass from the RX Transport to the Scatter engine, per host.	Counter	0x10800001000000XX • XX – Host ID

Name	Description	DataClass	Data ID
host_rx_transport_out_of_buffer_packets	The number of dropped packets due to a lack of WQE for the associated QPs/RQs (excluding hairpin QPs/RQs).	Counter	0x10800002000000XX • XX – Host ID
host_rx_transport_out_of_buffer_hairpin_packets	The number of dropped packets due to a lack of WQE for the associated hairpin QPs/RQs.	Counter	0x10800003000000XX • XX – Host ID
port_rx_transport_ecn_packets	The number of RoCEv2 packets received by the notification point which were marked for experiencing the congestion (i.e., ECN bits `11` on the ingress RoCE traffic), per port.	Counter	0x10800004000000XX • XX – Local port
port_rx_transport_cnp_handled_packets	The number of CNP received packets handled by the Reaction Point, per port.	Counter	0x10800005000000XX • XX – Local port
port_tx_transport_cnp_sent_packets	The number of CNP packets sent by the Notification Point, per port.	Counter	0x11000001000000XX • XX – Local port
tx_transport_done_due_to_cc_deschedule_events	The number of QP de-scheduled due to congestion control rate limitation.	Counter	0x11000002000000
port_tx_bytes	The number of transmitted bytes on the physical port (excluding loopback traffic).	Counter	0x11400001000000XX • XX – Local port
port_priority_tx_bytes	The number of transmitted bytes on the physical port and priority (excluding loopback traffic).	Counter	0x11400002000000YXX • XX – Local port • Y – Local port priority
port_tx_packets	The number of transmitted packets on the physical port (excluding loopback traffic).	Counter	0x11400003000000XX • XX – Local port
port_priority_tx_packets	The number of transmitted packets on the physical port and priority (excluding loopback traffic).	Counter	0x11400004000000YXX • XX – Local port • Y – Local port priority

Name	Description	DataClass	Data ID
port_priority_tx_pauses_packets	The number of link–layer pause packets transmitted on a physical port and priority.	Counter	0x1140000500000YXX • XX – Local port • Y – Local port priority
pcie_link_inbound_bytes	The number of bytes received from the PCIe toward the device, per PCIe link.	Counter	0x1160000100ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
pcie_link_outbound_bytes	The number of bytes transmitted from the device toward the PCIe, per PCIe link.	Counter	0x1160000200ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
pcie_link_inbound_data_bytes	The number of data bytes received from the PCIe (excluding headers) toward the device, per PCIe link.	Counter	0x1160000300ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
pcie_link_outbound_data_bytes	The number of data bytes transmitted from the device toward the PCI (excluding headers), per PCIe link.	Counter	0x1160000400ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
pcie_link_write_stalled_time_no_posted_data_credits_ns	The time period (in nanoseconds) in which the device had outbound posted write requests but stalled due to insufficient data credits per PCIe link.	Counter	0x1160000500ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)

Name	Description	DataClass	Data ID
pcie_link_write_stalled_time_no_posted_header_credits_ns	The time period (in nanoseconds) in which the device had outbound posted write requests but stalled due to insufficient header credits per PCIe link.	Counter	0x1160000600 ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
pcie_link_read_stalled_time_no_non_posted_data_credits_ns	The time period (in nanoseconds) in which the device had outbound non-posted read requests but stalled due to insufficient data credits per PCIe link.	Counter	0x1160000700 ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
pcie_link_read_stalled_time_no_non_posted_header_credits_ns	The time period (in nanoseconds) in which the device had outbound non-posted read requests but stalled due to insufficient header credits per PCIe link.	Counter	0x1160000800 ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
pcie_link_read_stalled_time_no_completion_buffers_ns	The time period (in nanoseconds) in which the device had outbound non-posted read requests but stalled due to no NIC completion buffers per PCIe link.	Counter	0x1160000900 ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
pcie_link_tclass_read_stalled_time_ordering_ns	The time period (in nanoseconds) in which the device had outbound non-posted read requests but stalled due to PCIe ordering semantics per PCIe link and PCIe tclass.	Counter	0x1160000aZZ ZZYYXX • XX – Node • YY – PCIe index • ZZZZ – (tclass (0 – 7) << 6) | (Depth (0 – 63))
pcie_link_latency_total_read_ns	The total latency (in nanoseconds) for all PCIe read from the device per PCIe link.[2]	Statistic	0x1160000b00 ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)

Name	Description	DataClass	Data ID
pcie_link_latency_total_read_packets	The total number of packets used for the pcie_link_latency_total_read_ns calculation.	Statistic	0x1160000c00ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
pcie_link_latency_max_read_ns	The maximum latency (in nanoseconds) for a single PCIe read from the device per PCIe link.	Statistic	0x1160000d00ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
pcie_link_latency_min_read_ns	The minimum latency (in nanoseconds) for a single PCIe read from the device per PCIe link.	Statistic	0x1160000e00ZZYYXX • XX – Node • YY – PCIe index • ZZ – Depth (0 – 63)
mplhr_pcie_link_latency_read_hist_packets	The number of PCIe reads with latency in the range of the PCIe Latency Histogram bin per PCIe link. The sampled value will be the same as the MPLH corresponding bin entry. "Bin" is currently 0→9 (10 bins in total).		0x1160000fZZZZYYXX • XX – Node • YY – PCIe index • ZZZZ – (Bin (0 – 31) << 6) | (Depth (0 – 63))
global_completion_engine_rx_cqes	Number of responder (RX) CQEs.	Counter	0x10c0000100000000
function_completion_engine_rx_cqes	Number of RX CQEs per function.	Counter	0x10c0000200000XXXX • XXXX – vhca_id
global_completion_engine_tx_cqes	Number of requestor (TX) CQEs.	Counter	0x10c0000400000000
function_completion_engine_tx_cqes	Number of TX CQEs per function.	Counter	0x10c0000500000XXXX • XXXX – vhca_id

Name	Description	DataClass	Data ID
global_icmc_request	Number of accesses to ICMC.	Counter	0x1180000100000000
global_icmc_hit	Number of ICMC hits.	Counter	0x1180000200000000
global_icmc_miss	Number of ICMC misses.	Counter	0x1180000300000000

[1] This counter includes loopback traffic and does not include packets discarded due to FCS, frame size, and similar errors.

[2] Dividing this counter by `pcie_link_latency_total_read_packets` *yields the average PCIe read latency of those reads.*

5.4.19.5 DOCA Telemetry DPA

This guide provides instructions for building and developing applications that require telemetry data collection from NVIDIA® BlueField® and NVIDIA® ConnectX® networking platforms using the DOCA Telemetry DPA API.

5.4.19.5.1 Introduction

The DOCA Telemetry DPA library provides access to detailed telemetry data and performance statistics for the Data Path Accelerator (DPA) on supported NVIDIA networking platforms. With its API, developers can monitor and analyze DPA processes, threads, and profiling data for efficient application performance optimization.

5.4.19.5.2 Prerequisites

To utilize DOCA Telemetry DPA, your system must meet the following baseline requirements:

- Firmware: Version `>=28/32/40.43.1000` is required for ConnectX-7, BlueField-3, and ConnectX-8 devices.
- Driver: The `fwctl` driver must be fully installed and actively loaded on the system.

5.4.19.5.2.1 Verifying the fwctl Driver

To verify that the `fwctl` driver is successfully loaded, check the device directories:

```
$ ls /sys/class/fwctl/
```

```
$ ls /dev/fwctl
```

⚠ The expected output for a standard 2-port device is **fwctl0 fwctl1**.

5.4.19.5.2.2 Manually Loading the Driver

If the directories **/sys/class/fwctl** or **/dev/fwctl** do not exist or are empty, the module may be installed but inactive.

Check for the module's presence:

```
$ grep fwctl -R /lib/modules/$(uname -r)/
```

If the output confirms the presence of **fwctl.ko** and **mlx5_fwctl.ko**, manually load the module and verify its status:

```
$ sudo modprobe mlx5_fwctl
$ lsmod | grep fwctl
```

5.4.19.5.2.3 Reinstalling the DOCA Host Package

If you cannot locate the installed **fwctl** module while manually loading the driver, or if the **modprobe** command fails to load it successfully, you must reinstall the DOCA Host package.

1. Download the package (DOCA 3.3.0 example):

   ```
   $ wget https://www.mellanox.com/downloads/DOCA/DOCA_v3.3.0/host/doca-host_3.3.0-088000-26.01-ubuntu2204_amd64.deb
   ```

2. Purge existing DOCA and OFED modules:

   ```
   $ sudo for f in $( dpkg --list | grep doca | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
   $ sudo for f in $( dpkg --list | grep mlnx | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
   $ sudo for f in $( dpkg --list | grep dpdk | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
   $ sudo for f in $( dpkg --list | grep ofed | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
   $ sudo /usr/sbin/ofed_uninstall.sh --force
   $ sudo apt-get autoremove
   ```

3. Install the new package and restart services:

   ```
   $ sudo dpkg -i doca-host_3.3.0-088000-26.01-ubuntu2204_amd64.deb
   $ sudo apt-get update
   $ sudo apt-get -y install doca-all
   $ sudo /etc/init.d/openibd restart
   ```

   Once the reinstallation is complete, confirm the module is successfully loaded according to section "DOCA Telemetry DPA#Verifying the fwctl Driver".

5.4.19.5.3 Environment

DOCA Telemetry DPA-based applications can run on:

- Host machines – ConnectX-7 or BlueField-3 and newer

- DPU targets – BlueField-3 and newer

5.4.19.5.4 Architecture

DOCA Telemetry DPA provides comprehensive profiling data, including:

- Process and thread information – Monitor all active DPA processes and threads
- Cumulative performance counters – Track cumulative performance metrics to evaluate application behavior
- Event tracer data – Capture detailed event-based traces for in-depth analysis

To interact with a device, users must create a DOCA Telemetry DPA context using **doca_telemetry_dpa_create()**. Each context is independent of DOCA DPA contexts, meaning changes in DPA configurations are not automatically reflected in the telemetry context. A device typically corresponds to a specific port on a NIC.

Performance counters are not owned by the **doca_telemetry_dpa** context. Other active telemetry contexts could manage the same counters. Ensure only a single context is used to profile the DPA to avoid conflicts.

5.4.19.5.5 Configuration Phase

5.4.19.5.5.1 Device Support

DOCA Telemetry DPA requires a device to operate. For picking a device, refer to "DOCA Core Device Discovery".

As device capabilities may change (see DOCA Core Device Support), it is recommended to check your device using the **doca_telemetry_dpa_cap_is_supported()** method:

5.4.19.5.5.2 Output Structure Format

The user application is responsible for allocating the output structures. To that end, DOCA Telemetry DPA provides helper methods that return the structure size in bytes (see section DOCA Telemetry DPA#Execution Phase for more details).

The **doca_telemetry_dpa** context supports the following layout structures for the profile data:

doca telemetry_dpa_process_info	
dpa_process_id	Global DPA process ID
num_of_threads	Number of threads in the process
process_name	The name of the process

doca_telemetry_dpa_thread_info	
dpa_process_id	Global DPA process ID
dpa_thread_id	Global DPA thread ID

doca_telemetry_dpa_thread_info	
thread_name	The name of the thread

doca_telemetry_dpa_cumul_info	
dpa_process_id	Global DPA process ID
dpa_thread_id	Global DPA thread ID
time	Total time in ticks the thread has been active
cycles	Total execution unit cycles the thread used
instructions	Total number of instructions the thread executed
num_executions	Total number of thread executions

doca_telemetry_dpa_event_sample	
timestamp	Timestamp in µsec
cycles	Stamp of total execution unit (EU) cycles
instructions	Stamp of total number of instructions of this DPA EU
dpa_thread_id	Global DPA thread ID
eu_id	Execution unit ID
sample_id_in_eu	Running sample ID per EU. A single sample_id is assigned to both schedule in and out samples.
type	Type of event sample: • DOCA_TELEMETRY_DPA_EVENT_SAMPLE_TYPE_EMPTY_SAMPLE • DOCA_TELEMETRY_DPA_EVENT_SAMPLE_TYPE_SCHEDULE_IN • DOCA_TELEMETRY_DPA_EVENT_SAMPLE_TYPE_SCHEDULE_OUT • DOCA_TELEMETRY_DPA_EVENT_SAMPLE_TYPE_BUFFER_FULL

The user can retrieve the DPA timer ticks frequency, given in kHZ, using doca_telemetry_dpa_get_dpa_timer_freq(). With this frequency, timer ticks can be converted to running clock using the formula: clock_time = ticks/dpa_timer_frequency.

5.4.19.5.5.3 Establishing the Amount of Event Tracer Samples

The user must set the maximum amount of event tracer samples to retrieve. This value can be set using doca_telemetry_dpa_set_max_perf_event_samples() and retrieved using doca_telemetry_dpa_get_max_perf_event_samples().

5.4.19.5.5.4 Addressing All Processes and Threads

- To retrieve a unique ID to address all processes running on the DPA:

```
uint32_t all_processes_id;
doca_telemetry_dpa_get_all_process_id(&all_processes_id);
```

- To retrieve a unique ID to address all threads running on the DPA:

```
uint32_t all_threads_id;
doca_telemetry_dpa_get_all_threads_id(&all_threads_id);
```

5.4.19.5.6 Execution Phase

5.4.19.5.6.1 Retrieving Running DPA Process Information

Information about specific process or all processes running on the DPA can be retrieved using these steps:

1. Get memory size for process list allocation:

```
uint32_t size;
doca_telemetry_dpa_get_process_list_size(context, process_id, &size);
```

2. Retrieve process list:

```
doca_telemetry_dpa_read_processes_list(context, processs_id, &process_num, &process_list);
```

Ensure memory is properly allocated using the retrieved size before calling **doca_telemetry_dpa_read_processes_list()**.

5.4.19.5.6.2 Retrieving Running DPA Thread Information

Information about specific thread or all threads running on the DPA can be retrieved using these steps:

1. Get memory size for thread list allocation:

```
uint32_t size;
doca_telemetry_dpa_get_thread_list_size(context, process_id, thread_id, &size);
```

2. Retrieve thread list:

```
doca_telemetry_dpa_read_thread_list(context, process_id, thread_id, &threads_num, &thread_list);
```

If the process ID is set to address all processes, the thread ID must also be set to address all threads.

⚠ Ensure memory is properly allocated using the retrieved size before calling **doca_telemetry_dpa_read_thread_list()**.

5.4.19.5.6.3 Retrieving Cumulative Performance Profile Information Samples

Cumulative performance samples can be retrieved for specific processes and threads or for all processes and threads (see DOCA Telemetry DPA#Address all processes and threads) running on the DPA using these steps:

1. Start cumulative counter/s:

```
doca_telemetry_dpa_counter_start(context, process_id, DOCA_TELEMETRY_DPA_COUNTER_TYPE_CUMULATIVE_EVENT);
```

2. Stop cumulative counter/s:

```
doca_telemetry_dpa_counter_stop(context, process_id, DOCA_TELEMETRY_DPA_COUNTER_TYPE_CUMULATIVE_EVENT);
```

3. Get memory size for cumulative samples list allocation:

```
uint32_t size;
doca_telemetry_dpa_get_cumul_samples_size(context, process_id, thread_id, &size);
```

4. Retrieve cumulative sample list:

```
doca_telemetry_dpa_read_cumul_info_list(context, process_id, thread_id, &cumul_samples_num,
&cumul_info_list);
```

ℹ The current state and type of a counter can be retrieved using **doca_telemetry_dpa_get_counter_state()** and **doca_telemetry_dpa_get_counter_type()**.

⚠ If the process ID is set to address all processes, the thread ID must also be set to address all threads.

⚠ Ensure memory is properly allocated using the retrieved size before calling **doca_telemetry_dpa_read_cumul_info_list()**.

⚠ Reset counters using **doca_telemetry_dpa_counter_restart()** to clear previous data.

5.4.19.5.6.4 Retrieving Event Tracer Profile Information Samples

Event tracer samples can be retrieved for specific processes and threads or for all processes and threads running on the DPA using these steps:

1. Start event tracer counter/s:

```
doca_telemetry_dpa_counter_start(context, all_processes_id, DOCA_TELEMETRY_DPA_COUNTER_TYPE_EVENT_TRACER);
```

2. Stop event tracer counter/s:

```
doca_telemetry_dpa_counter_stop(context, all_processes_id, DOCA_TELEMETRY_DPA_COUNTER_TYPE_EVENT_TRACER);
```

3. Get memory size for event samples list allocation:

```
uint32_t size;
doca_telemetry_dpa_get_perf_event_samples_size(context, process_id, thread_id, &size);
```

4. Retrieve event tracer sample list:

```
doca_telemetry_dpa_read_perf_event_list(context, process_id, thread_id, &perf_event_samples_num,
&event_info_list);
```

The current state and type of a counter can be retrieved using **doca_telemetry_dpa_get_counter_state()** and **doca_telemetry_dpa_get_counter_type()**.

Ensure the maximum number of samples is set before starting counters using **doca_telemetry_dpa_set_max_perf_event_samples()**.

If the process ID is set to address all processes, the thread ID must also be set to address all threads.

Ensure memory is properly allocated using the retrieved size before calling **doca_telemetry_dpa_read_perf_event_list()**.

Reset counters using **doca_telemetry_dpa_counter_restart()** before reuse.

5.4.19.5.7 State Machine

The following section describes the different states the **doca_telemetry_dpa** context goes through, how to move between states and what is allowed in each state.

5.4.19.5.7.1 Idle

The context has been created and is idle.

In this state, it is expected for the application to:

- Destroy the context
- Start the context for processing

Allowed operations:

- Configuring the context according to section "DOCA Telemetry DPA#Configuration Phase"

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop

5.4.19.5.7.2 Running

In this state it is expected for the application to:

1. Stop the context.
2. Retrieve process information list
3. Retrieve thread information list.
4. Start/stop/reset counters for profiling capabilities.
5. Retrieve profile samples for cumulative performance counters.
6. Retrieve profile samples for event tracer.

Allowed operations:

- Calling stop, moving the application to "Idle" state

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Successfully start the context

There are currently no state restrictions on the majority of API functions.

5.4.19.5.8 Alternative Datapath Options

DOCA Telemetry DPA supports only CPU-based datapaths.

5.4.19.5.8.1 Running the Sample

1. Refer to the following documents:
   a. DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   b. NVIDIA BlueField Platform Software Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

```
cd /opt/mellanox/doca/samples/doca_telemetry/telemetry_dpa
meson /tmp/build
ninja -C /tmp/build
```

The binary `doca_telemetry_dpa` is created under `/tmp/build/`.

Sample usage:

```
Usage: doca_telemetry_dpa [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -p, --pci-addr                    DOCA device PCI device address
  -rt, --sample-run-time            Total sample run time, in miliseconds
  -ct, --counter-type               Counter type, cumulus (0) or event (1)
  -es, --event-samples              Set the maximum number of perf event samples to retrieve
  -pi, --process-id                 Specific process id to address
  -ti, --thread-id                  Specific thread id to address
```

The sample includes:

1. Locating and opening a DOCA device.
2. Creating a `doca_telemetry_dpa` instance.
3. Retrieval of all or one specific process
4. Retrieval of all or one specific thread
5. Starting counters for the selected profile capability
6. Retrieving the profile samples for the selected profile capability
7. Displaying the retrieve profile information.
8. Destroying the `doca_telemetry_dpa` context.

5.4.19.6 DOCA Telemetry PCC

This guide provides instructions for building and developing applications that require telemetry data collection from NVIDIA® BlueField® and NVIDIA® ConnectX® families of networking platforms.

5.4.19.6.1 Introduction

The `doca_telemetry_pcc` library provides access to algorithm information and counter statistics for any Programmable Congestion Control (PCC) implementations currently installed on NVIDIA® BlueField or ConnectX® networking platforms.

5.4.19.6.2 Prerequisites

To utilize DOCA Telemetry PCC, your system must meet the following baseline requirements:

- Firmware: Version `>=28/32/40.43.1000` is required for ConnectX-7, BlueField-3, and ConnectX-8 devices (`>=*.49.1000` for histogram support).
- Driver: The `fwctl` driver must be fully installed and actively loaded on the system.

5.4.19.6.2.1 Verifying the fwctl Driver

To verify that the `fwctl` driver is successfully loaded, check the device directories:

```
$ ls /sys/class/fwctl/
$ ls /dev/fwctl
```

⚠ The expected output for a standard 2-port device is **fwctl0 fwctl1**.

5.4.19.6.2.2 Manually Loading the Driver

If the directories **/sys/class/fwctl** or **/dev/fwctl** do not exist or are empty, the module may be installed but inactive.

Check for the module's presence:

```
$ grep fwctl -R /lib/modules/$(uname -r)/
```

If the output confirms the presence of **fwctl.ko** and **mlx5_fwctl.ko**, manually load the module and verify its status:

```
$ sudo modprobe mlx5_fwctl
$ lsmod | grep fwctl
```

5.4.19.6.2.3 Reinstalling the DOCA Host Package

If you cannot locate the installed **fwctl** module while manually loading the driver, or if the **modprobe** command fails to load it successfully, you must reinstall the DOCA Host package.

1. Download the package (DOCA 3.3.0 example):

   ```
   $ wget https://www.mellanox.com/downloads/DOCA/DOCA_v3.3.0/host/doca-host_3.3.0-088000-26.01-ubuntu2204_amd64.deb
   ```

2. Purge existing DOCA and OFED modules:

   ```
   $ sudo for f in $( dpkg --list | grep doca | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
   $ sudo for f in $( dpkg --list | grep mlnx | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
   $ sudo for f in $( dpkg --list | grep dpdk | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
   $ sudo for f in $( dpkg --list | grep ofed | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
   $ sudo /usr/sbin/ofed_uninstall.sh --force
   $ sudo apt-get autoremove
   ```

3. Install the new package and restart services:

   ```
   $ sudo dpkg -i doca-host_3.3.0-088000-26.01-ubuntu2204_amd64.deb
   $ sudo apt-get update
   $ sudo apt-get -y install doca-all
   $ sudo /etc/init.d/openibd restart
   ```

   Once the reinstallation is complete, confirm the module is successfully loaded according to section "DOCA Telemetry PCC#Verifying the fwctl Driver".

5.4.19.6.3 Environment

DOCA Telemetry-based applications can run on either the host machine (ConnectX-7 or BlueField-3 and newer) or the DPU target (BlueField-3 and newer).

5.4.19.6.4 Architecture

DOCA Telemetry PCC allows for the collection of information from any PCC images installed and actively running on a ConnectX or BlueField device.

To interact with a device, a separate **doca_telemetry_pcc** context must be created for each device. A device typically corresponds to a specific port on a NIC.

> ⚠ You can also create a context on a representor device using **doca_telemetry_rep_pcc**. This approach is often required in configurations like switchdev bonds, where lower-level ports are exposed as DOCA device representors rather than full DOCA devices.

The **doca_telemetry_pcc** context operates independently of any **doca_pcc** contexts. This means that changes to the configuration of a running PCC image, such as disabling an algorithm, are not automatically reflected in the telemetry context. Implementing applications must handle these changes either by directly interacting with the PCC application or by scanning all algorithm slots to determine which are currently active.

It is assumed that a new PCC image will not be installed while a **doca_telemetry_pcc** context is active on the device.

> ⚠ If a complete PCC re-image is performed, a new **doca_telemetry_pcc** context must be created.

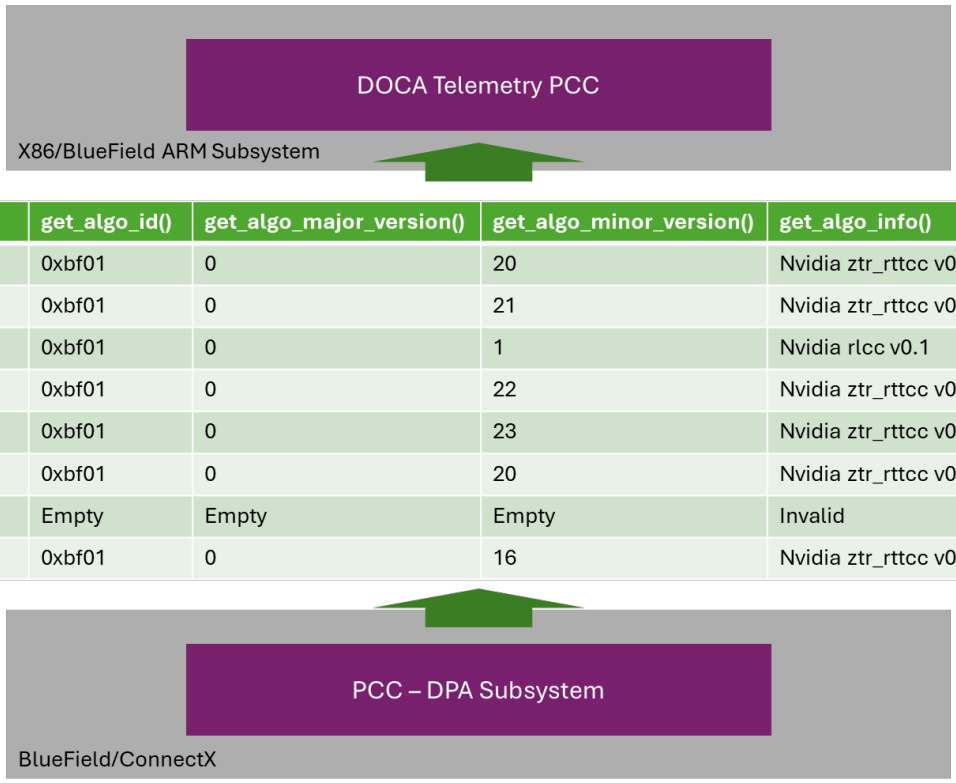
5.4.19.6.5 Schema Retrieval

5.4.19.6.5.1 Slots/Algorithms

An active PCC image running on the DPA subsystem of a device can contain multiple Congestion Control algorithms. Each algorithm is assigned to a specific "slot," with only one slot/algorithm active on the device at any given time.

The algorithms in each slot cannot be modified without re-imaging the PCC algorithm on the DPAs. While DOCA Telemetry PCC does not provide functionality to configure slots or algorithms, it enables the calling application to retrieve information about the algorithms installed in each populated slot.

For example:



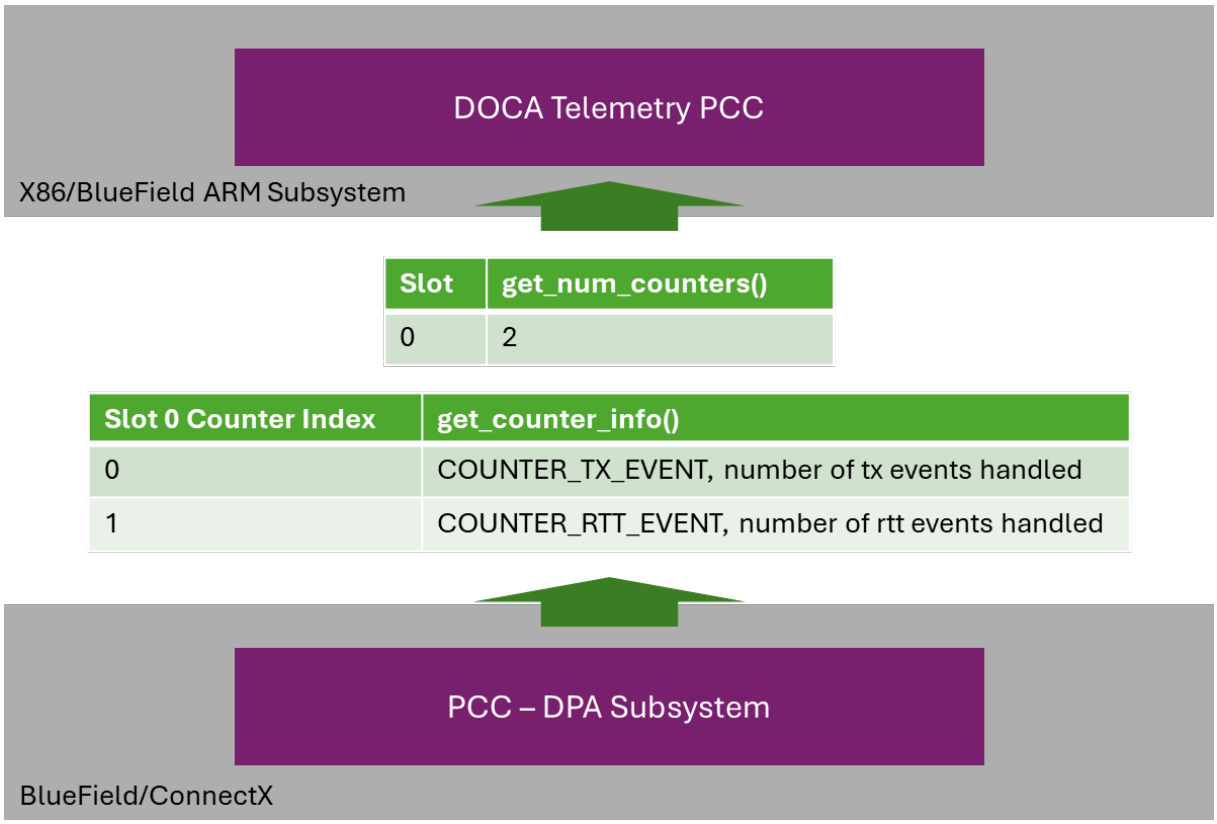
Slot Index	get_algo_id()	get_algo_major_version()	get_algo_minor_version()	get_algo_info()
1	0xbf01	0	20	Nvidia ztr_rttcc v0.20
2	0xbf01	0	21	Nvidia ztr_rttcc v0.21
3	0xbf01	0	1	Nvidia rlcc v0.1
4	0xbf01	0	22	Nvidia ztr_rttcc v0.22
5	0xbf01	0	23	Nvidia ztr_rttcc v0.23
6	0xbf01	0	20	Nvidia ztr_rttcc v0.20
7	Empty	Empty	Empty	Invalid
8	0xbf01	0	16	Nvidia ztr_rttcc v0.16

5.4.19.6.5.2 Counter Information

Similar to slots, the counter definitions for each algorithm cannot be modified without re-imaging a new PCC.

DOCA Telemetry PCC allows applications to retrieve the number of counters defined for an algorithm applied to a specific port. Each counter's definition can then be accessed using an index value, ranging from `0` to `total_counters-1`.

For example:

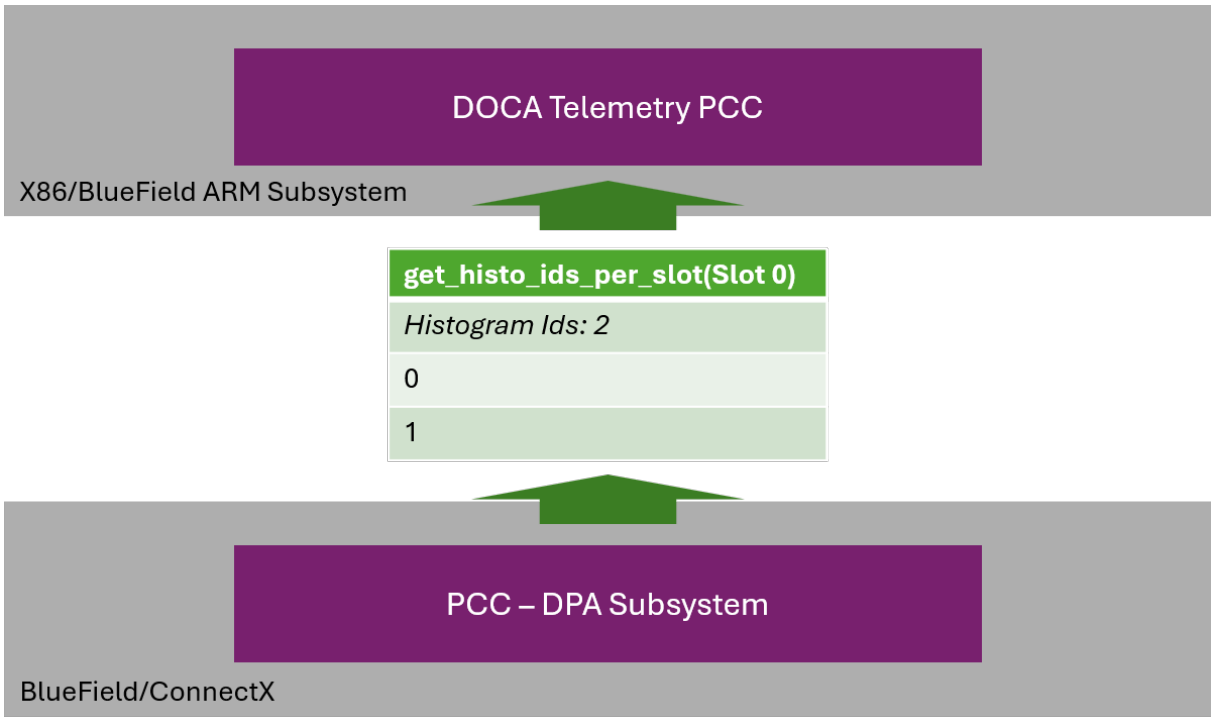


5.4.19.6.5.3 Histogram Information

Algorithms running on a given slot may contain one or more histograms.

Unlike simple counters that track totals, histograms provide finer-grained insights by grouping events into value ranges (such as time intervals) and reporting counts per bin.

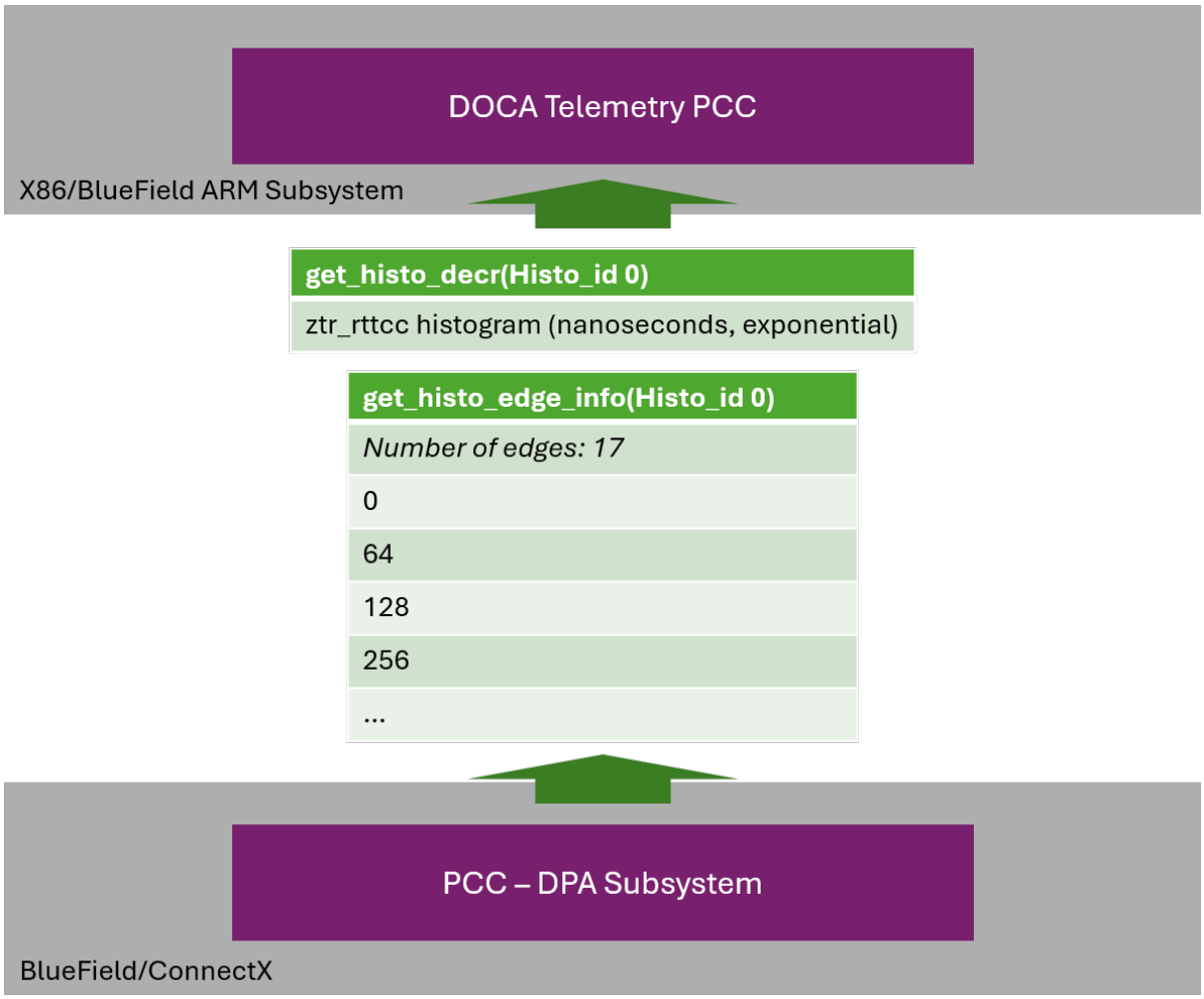
A slot can be queried to determine if it has histogram IDs associated with it.



Histogram information can then be accessed directly using a valid ID.

Histogram edge values define the boundaries of each bin, where bin 0 includes values from edge 0 up to edge 1, bin 1 spans edge 1 to edge 2, and so on.

For example:



5.4.19.6.6 Run-time Retrieval

5.4.19.6.6.1 Status

The congestion algorithm actively running on a device at any given time is determined by the lowest enabled slot index, as multiple slots can be marked as enabled simultaneously.

The enabling or disabling of a slot can be triggered dynamically at runtime by external processes, independent of DOCA Telemetry PCC. Automatically detecting these changes falls outside the scope of a **doca_telemetry_pcc** context. It is the user's responsibility to either scan all slots periodically or interface directly with the external process to monitor changes.

In addition to enabling/disabling a slot, counters for a slot can also be enabled or disabled. Only one slot can have active counters at any given time. If counters are not active on an algorithm slot, any attempt to poll their values will result in a **DOCA_ERROR_BAD_STATE**.

The status of both the slot and its counters can be inspected using the **doca_telemetry_pcc_get_algo_enable_status** API.

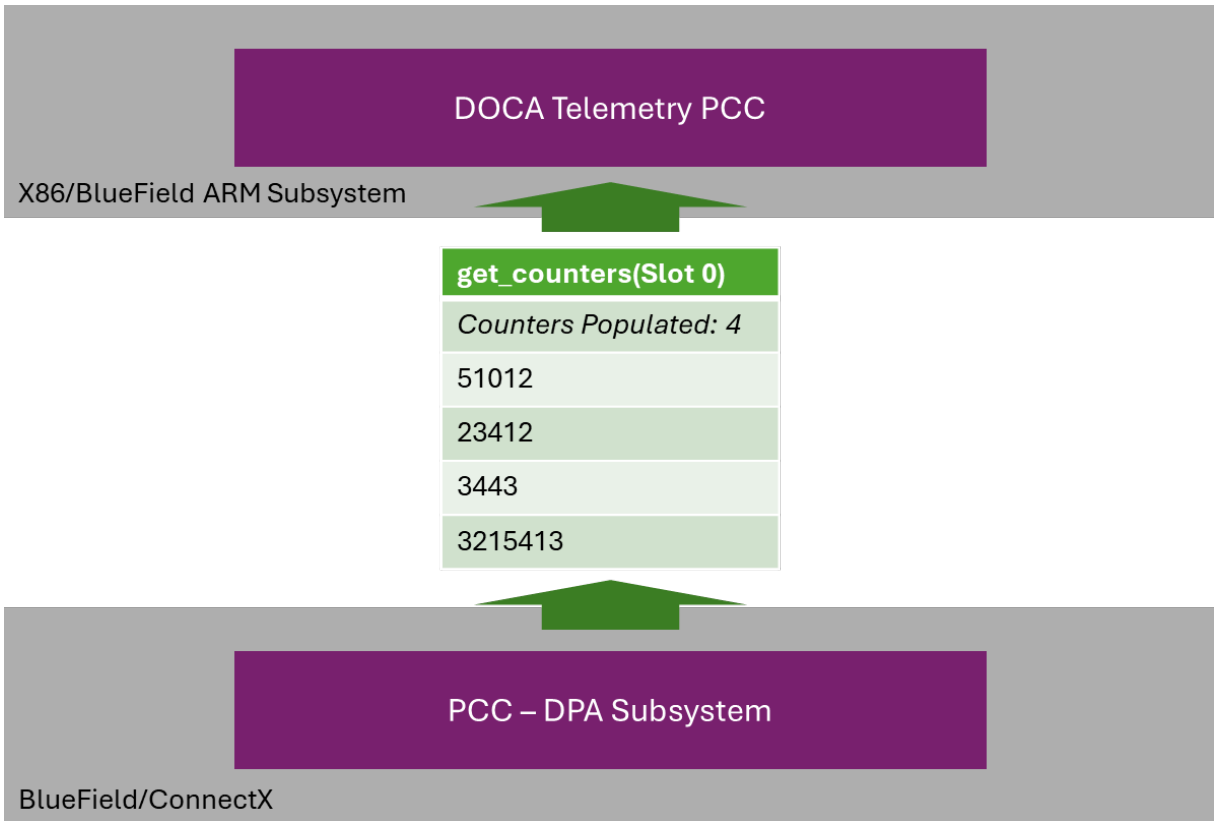
⚠ Since PCC configuration is managed outside DOCA Telemetry PCC, there is no guarantee that system conditions remain consistent between API calls. For example, `get_algo_enable_status()` may indicate that counters are enabled, but subsequent polling of those counters could fail if they were disabled between the calls.

5.4.19.6.6.2 Counters

Counter values for a slot with enabled counters can be polled using a `doca_telemetry_pcc` context associated with a specific device.

When retrieving counter values, the API returns all available counters as a block rather than individually. The counters are provided in an array of 32-bit values, along with the total number of counters populated in the array.

It is assumed that the user is familiar with what each counter index represents (refer to the "DOCA Telemetry PCC#Counter Information" section for details).



DOCA Telemetry PCC provides another API function which enables the resetting of counters if so required.

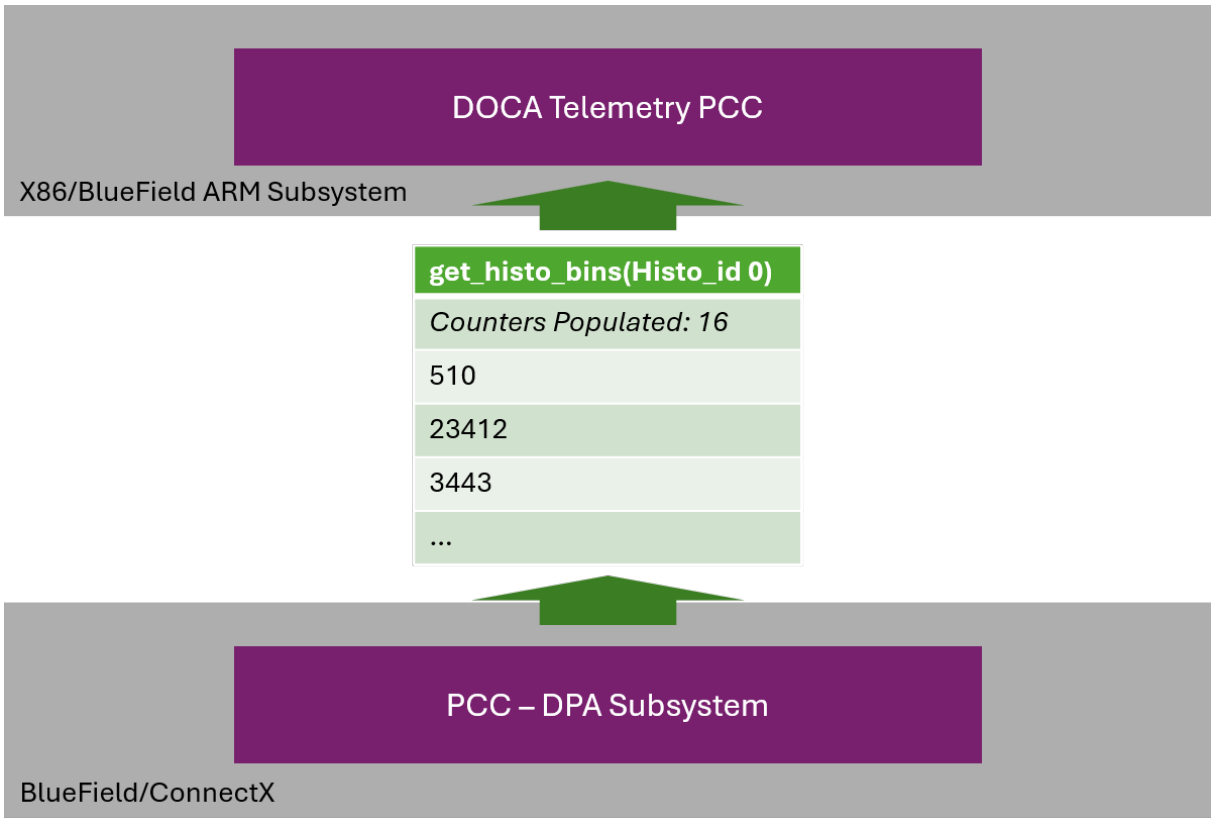
5.4.19.6.6.3 Histogram Bins

Histogram bins values are returned as an array of 32-bit values in same way as counters.

The difference is that the request is made with a histogram ID rather than a slot value.

The meaning of each bin is defined by the description and edge values extracted via API calls from sectionDOCA Telemetry PCC .

The number of bins will therefore be one fewer than the number of edge values.



Similar to counters, DOCA Telemetry PCC offers a further API to reset histogram bins after reading.

5.4.19.6.7 State Machine

This section outlines the various states the `doca_telemetry_pcc` context transitions through, the methods for transitioning between these states, and the operations permitted in each state.

5.4.19.6.7.1 Idle

The context has been created and is Idle.

In this state, it is expected for the application to:

- Destroy the context.
- Start the context for processing.

Allowed operations:

- Configuring the context.

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop

5.4.19.6.7.2 Running

In this state it is expected for the application to:

- Stop the context.

Allowed operations:

- Calling stop, moving the application to "Idle" state

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Successfully start the context

There are currently no state restrictions on the majority of API functions.

5.4.19.6.8 Alternative Datapath Options

DOCA Telemetry PCC supports only CPU-based datapaths.

5.4.19.6.9 DOCA Telemetry PCC Sample

This section describes a telemetry PCC sample based on the **doca_telemetry_pcc** library. The sample illustrates the utilization of DOCA telemetry PCC APIs to initialize and configure the **doca_telemetry_pcc** context, as well as querying and parsing PCC schema and counters.

These samples are also available on GitHub.

5.4.19.6.9.1 Running the Sample

1. Refer to the following documents:
   a. *DOCA Installation Guide for Linux* for details on how to install BlueField-related software.
   b. *NVIDIA BlueField Platform Software Troubleshooting Guide* for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

   ```
   cd /opt/mellanox/doca/samples/doca_telemetry/telemetry_pcc
   meson /tmp/build
   ninja -C /tmp/build
   ```

   The binary **doca_telemetry_pcc** is created under **/tmp/build/**.

Sample usage:

```
Usage: doca_telemetry_pcc [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -p, --pci-addr                    DOCA device PCI device address
  -r, --rep-addr                    DOCA device representor PCI device address
```

The sample includes:

1. Locating and opening a DOCA device.
2. Creating a **doca_telemetry_pcc** instance.
3. Parsing all possible slots, printing each:
    a. Algo ID
    b. Major version
    c. Minor version
    d. Description
4. Determining and displaying the current enabled status of each slot.
5. Counter values and descriptions for any slot with counters enabled.
6. Histogram description, bin information, and counters if any are associated with a slot.
7. Destroying the **doca_telemetry_pcc** context.

5.4.19.7 DOCA Telemetry PCI

This guide provides instructions for building and developing applications that require telemetry data collection from NVIDIA® BlueField and NVIDIA® ConnectX® families of networking platforms.

5.4.19.7.1 Introduction

The **doca_telemetry_pci** library provides access to PCIe status and performance information from BlueField or ConnectX networking platforms.

5.4.19.7.2 Prerequisites

To utilize DOCA Telemetry PCI, your system must meet the following baseline requirements:

- Firmware: Version **>=28/32/40.43.1000** is required for ConnectX-7, BlueField-3, and ConnectX-8 devices.
- Driver: The **fwctl** driver must be fully installed and actively loaded on the system.

5.4.19.7.2.1 Verifying the fwctl Driver

To verify that the **fwctl** driver is successfully loaded, check the device directories:

```
$ ls /sys/class/fwctl/
$ ls /dev/fwctl
```

⚠ The expected output for a standard 2-port device is `fwctl0 fwctl1`.

5.4.19.7.2.2 Manually Loading the Driver

If the directories `/sys/class/fwctl` or `/dev/fwctl` do not exist or are empty, the module may be installed but inactive.

Check for the module's presence:

```
$ grep fwctl -R /lib/modules/$(uname -r)/
```

If the output confirms the presence of `fwctl.ko` and `mlx5_fwctl.ko`, manually load the module and verify its status:

```
$ sudo modprobe mlx5_fwctl
$ lsmod | grep fwctl
```

5.4.19.7.2.3 Reinstalling the DOCA Host Package

If you cannot locate the installed `fwctl` module while manually loading the driver, or if the `modprobe` command fails to load it successfully, you must reinstall the DOCA Host package.

1. Download the package (DOCA 3.3.0 example):

   ```
   $ wget https://www.mellanox.com/downloads/DOCA/DOCA_v3.3.0/host/doca-host_3.3.0-088000-26.01-ubuntu2204_amd64.deb
   ```

2. Purge existing DOCA and OFED modules:

   ```
   $ sudo for f in $( dpkg --list | grep doca | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
   $ sudo for f in $( dpkg --list | grep mlnx | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
   $ sudo for f in $( dpkg --list | grep dpdk | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
   $ sudo for f in $( dpkg --list | grep ofed | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
   $ sudo /usr/sbin/ofed_uninstall.sh --force
   $ sudo apt-get autoremove
   ```

3. Install the new package and restart services:

   ```
   $ sudo dpkg -i doca-host_3.3.0-088000-26.01-ubuntu2204_amd64.deb
   $ sudo apt-get update
   $ sudo apt-get -y install doca-all
   $ sudo /etc/init.d/openibd restart
   ```

   Once the reinstallation is complete, confirm the module is successfully loaded according to section "DOCA Telemetry PCI#Verifying the fwctl Driver".

5.4.19.7.3 Environment

DOCA Telemetry-based applications can run on either the host machine (ConnectX-7 or BlueField-3 and newer) or the DPU target (BlueField-3 and newer).

5.4.19.7.4 Architecture

DOCA Telemetry PCI provides insights into PCIe devices, including:

- Management information: PCIe link and speed details, power usage, function count, error detection flags, and more.
- PCIe performance counters: Data transfer rates, error rates, stall counters, L0 recovery count, and other performance metrics.
- PCIe latency histogram: Helps understand the duration of PCIe operations.

To interact with a device, typically corresponding to a specific NIC port, create a DOCA Telemetry PCI context using `doca_telemetry_pci_create()`.

5.4.19.7.5 Configuration Phase

5.4.19.7.5.1 Device Support

DOCA Telemetry PCI requires a device to operate. For picking a device, refer to "DOCA Core Device Discovery".

As device capabilities may change, it is recommended to check your device using the required set of PCIe telemetry options you desire before opening it to be confident the operations you desire are available. The set of available capability checks for DOCA Telemetry PCI are out lined below:

Functionality	Method
PCI Management Information	`doca_telemetry_pci_cap_management_info_is_supported()`
PCI Performance Counters Group 1	`doca_telemetry_pci_cap_perf_counters_1_is_supported()`
PCI Latency Histogram	`doca_telemetry_pci_cap_latency_histogram_is_supported()`

Within the structures provided during the execution phase some fields are only populated which a further sub-capability is also supported:

Functionality	Field(s)	Method
PCI Management Information	`pwr_status`, `pci_power`	`doca_telemetry_pci_cap_management_info_power_reporting_is_supported()`
	`link_peer_max_speed`	`doca_telemetry_pci_cap_management_info_link_peer_max_speed_is_supported()`
PCI Performance Counters Group 1	`tx_overflow_buffer_pkt`, `tx_overflow_buffer_marked_pkt`	`doca_telemetry_pci_cap_perf_counters_1_tx_overflow_is_supported()`
	`outbound_stalled_reads`, `outbound_stalled_writes`, `outbound_stalled_reads_events`, `outbound_stalled_writes_events`	`doca_telemetry_pci_cap_perf_counters_1_outbound_stalled_is_supported()`

Functionality	Field(s)	Method
	fec_correctable_error_counter, fec_uncorrectable_error_counter	doca_telemetry_pci_cap_perf_counters_1_fec_error_is_supported()
	fber_magnitude, fber_coef	doca_telemetry_pci_cap_perf_counters_1_fber_is_supported()

5.4.19.7.6 Execution Phase

5.4.19.7.6.1 Retrieving PCIe Management Information

Using a running `doca_telemetry_pci` context which supports PCIe management information the user can call `doca_telemetry_pci_read_management_info` as many times as they like to get the most recent data available with each call.

The following is a more complete example:

```
doca_error_t result;

// Check for Ability to read management info
result = doca_telemetry_pci_cap_management_info_is_supported(devinfo);
if(result != DOCA_SUCCESS)
    // Capability is not supported or an error occoured, stop

// Check any sub capabilities if you require those fields

// Create PCI telemetry
struct doca_telemetry_pci *pci_telem;
result = doca_telemetry_pci_create(dev, &pci_telem);
if(result != DOCA_SUCCESS)
    // Handle failure to create telemetry instance

// Start PCI telemetry
result = doca_telemetry_pci_start(pci_telem);
if(result != DOCA_SUCCESS)
    // Handle failure to start telemetry instance

// Read management info
struct doca_telemetry_pci_management_info management_info = {};

// Use DPN to identify the data subject
struct doca_telemetry_pci_dpn dpn = {0, 0, 0};
result = doca_telemetry_pci_read_management_info(pci_telem, dpn, &management_info);
if(result != DOCA_SUCCESS)
    // Handle failure to read data

// Use PCI address to identify the data subject (Replace SSSS.BB.DD.F with an actual PCI address)
result = doca_telemetry_pci_read_management_info_by_pci_addr(pci_telem, "SSSS.BB.DD.F", &management_info);
if(result != DOCA_SUCCESS)
    // Handle failure to read data

// Use the data

// Cleanup
doca_telemetry_pci_stop(pci_telem);
doca_telemetry_pci_destroy(pci_telem);
```

5.4.19.7.6.2 Retrieving Performance Counters Group 1

Using a running `doca_telemetry_pci` context which supports performance counters group 1 the user can call `doca_telemetry_pci_read_perf_counters_1` as many times as they like to get the most recent data available with each call.

The following is a more complete example:

```
doca_error_t result;

// Check for Ability to read perf counters group 1
```

```
result = doca_telemetry_pci_cap_perf_counters_1_is_supported(devinfo);
if(result != DOCA_SUCCESS)
    // Capability is not supported or an error occoured, stop

// Check any sub capabilities if you require those fields

// Create PCI telemetry
struct doca_telemetry_pci *pci_telem;
result = doca_telemetry_pci_create(dev, &pci_telem);
if(result != DOCA_SUCCESS)
    // Handle failure to create telemetry instance

// Start PCI telemetry
result = doca_telemetry_pci_start(pci_telem);
if(result != DOCA_SUCCESS)
    // Handle failure to start telemetry instance

// Read perf counters group 1
struct doca_telemetry_pci_perf_counters_1 counters= {};

// Use DPN to identify the data subject
struct doca_telemetry_pci_dpn dpn = {0, 0, 0};
result = doca_telemetry_pci_read_perf_counters_1(pci_telem, dpn, &counters);
if(result != DOCA_SUCCESS)
    // Handle failure to read data

// Use PCI address to identify the data subject (Replace SSSS.BB.DD.F with an actual PCI address)
result = doca_telemetry_pci_read_perf_counters_1_by_pci_addr(pci_telem, "SSSS.BB.DD.F", &counters);
if(result != DOCA_SUCCESS)
    // Handle failure to read data

// Use the data

// Cleanup
doca_telemetry_pci_stop(pci_telem);
doca_telemetry_pci_destroy(pci_telem);
```

5.4.19.7.6.3 Retrieving Latency Histogram

Using a running **doca_telemetry_pci** context which supports latency histogram the user must first call **doca_telemetry_pci_get_latency_histogram_dimensions** to learn the correct dimmensions of the histogram. They can then allocate an array of histogram values and then finally they can call **doca_telemetry_pci_read_latency_histogram** as many times as they like to get the most recent data available with each call.

The following is a more complete example:

```
doca_error_t result;

// Check for Ability to read latency histogram
result = doca_telemetry_pci_cap_latency_histogram_is_supported(devinfo);
if(result != DOCA_SUCCESS)
    // Capability is not supported or an error occoured, stop

// Create PCI telemetry
struct doca_telemetry_pci *pci_telem;
result = doca_telemetry_pci_create(dev, &pci_telem);
if(result != DOCA_SUCCESS)
    // Handle failure to create telemetry instance

// Start PCI telemetry
result = doca_telemetry_pci_start(pci_telem);
if(result != DOCA_SUCCESS)
    // Handle failure to start telemetry instance

// Learn the histograms dimmensions
uint32_t bucket_count;
uint32_t bucket_width_ns;

// Use DPN to identify the data subject
struct doca_telemetry_pci_dpn dpn = {0, 0, 0};
result = doca_telemetry_pci_get_latency_histogram_dimensions(pci_telem, dpn, &bucket_count, &bucket_width_ns);
if(result != DOCA_SUCCESS)
    // Handle failure to get histogram dimmensions

// Use PCI address to identify the data subject (Replace SSSS.BB.DD.F with an actual PCI address)
result = doca_telemetry_pci_get_latency_histogram_dimensions_by_pci_addr(pci_telem, "SSSS.BB.DD.F", &bucket_count,
&bucket_width_ns);
if(result != DOCA_SUCCESS)
    // Handle failure to get histogram dimmensions

// Allocate memory to hold histogram data
uint64_t* buckets_arr = malloc(bucket_count * sizeof(uint64_t));
if( buckets_arr == NULL)
    // Handle failure to allocate memory

// Fetch histogram data

// Use DPN to identify the data subject
result = doca_telemetry_pci_read_latency_histogram(pci_telem, dpn, buckets_arr);
if(result != DOCA_SUCCESS)
```

```
    // Handle failure to read data

// Use PCI address to identify the data subject (Replace SSSS.BB.DD.F with an actual PCI address)
result = doca_telemetry_pci_read_latency_histogram_by_pci_addr(pci_telem, "SSSS.BB.DD.F", buckets_arr);
if(result != DOCA_SUCCESS)
    // Handle failure to read data

// Use the data

// Cleanup
free(buckets_arr);
doca_telemetry_pci_stop(pci_telem);
doca_telemetry_pci_destroy(pci_telem);
```

5.4.19.7.7 Alternative Datapath Options

DOCA Telemetry PCI supports only CPU-based datapaths.

5.4.19.8 DOCA Telemetry PHY

This guide provides instructions for building and developing applications that require telemetry data collection from NVIDIA® BlueField and NVIDIA® ConnectX® families of networking platforms.

5.4.19.8.1 Introduction

The DOCA Telemetry PHY library provides access to detailed telemetry data and statistics from NVIDIA BlueField and ConnectX networking platforms. It allows developers to monitor and analyze link, cable, and port status and statistics.

5.4.19.8.2 Prerequisites

To utilize DOCA Telemetry PHY, your system must meet the following baseline requirements:

- Firmware: Version **>=28/32/40.43.1000** is required for ConnectX-7, BlueField-3, and ConnectX-8 devices.
- Driver: The **fwctl** driver must be fully installed and actively loaded on the system.

5.4.19.8.2.1 Verifying the fwctl Driver

To verify that the **fwctl** driver is successfully loaded, check the device directories:

```
$ ls /sys/class/fwctl/
$ ls /dev/fwctl
```

The expected output for a standard 2-port device is **fwctl0 fwctl1**.

5.4.19.8.2.2 Manually Loading the Driver

If the directories **/sys/class/fwctl** or **/dev/fwctl** do not exist or are empty, the module may be installed but inactive.

Check for the module's presence:

```
$ grep fwctl -R /lib/modules/$(uname -r)/
```

If the output confirms the presence of **fwctl.ko** and **mlx5_fwctl.ko**, manually load the module and verify its status:

```
$ sudo modprobe mlx5_fwctl
$ lsmod | grep fwctl
```

5.4.19.8.2.3 Reinstalling the DOCA Host Package

If you cannot locate the installed **fwctl** module while manually loading the driver, or if the **modprobe** command fails to load it successfully, you must reinstall the DOCA Host package.

1. Download the package (DOCA 3.3.0 example):

   ```
   $ wget https://www.mellanox.com/downloads/DOCA/DOCA_v3.3.0/host/doca-host_3.3.0-088000-26.01-ubuntu2204_amd64.deb
   ```

2. Purge existing DOCA and OFED modules:

   ```
   $ sudo for f in $( dpkg --list | grep doca | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
   $ sudo for f in $( dpkg --list | grep mlnx | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
   $ sudo for f in $( dpkg --list | grep dpdk | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
   $ sudo for f in $( dpkg --list | grep ofed | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ; done
   $ sudo /usr/sbin/ofed_uninstall.sh --force
   $ sudo apt-get autoremove
   ```

3. Install the new package and restart services:

   ```
   $ sudo dpkg -i doca-host_3.3.0-088000-26.01-ubuntu2204_amd64.deb
   $ sudo apt-get update
   $ sudo apt-get -y install doca-all
   $ sudo /etc/init.d/openibd restart
   ```

   Once the reinstallation is complete, confirm the module is successfully loaded according to section "DOCA Telemetry PHY#Verifying the fwctl Driver".

5.4.19.8.3 Environment

DOCA Telemetry PHY-based applications can run on:

- Host machines with ConnectX-7 or BlueField-3 and newer
- DPU targets with BlueField-3 and newer

5.4.19.8.4 Architecture

The DOCA Telemetry PHY library provides access to the following categories of telemetry data:

- Operation information – Retrieves details related to active link technology and capabilities.
- Supported information – Retrieves details related to supported link speed capabilities.
- Troubleshooting information – Retrieves status information specifically for troubleshooting purposes.
- Module information – Retrieves details related to cable technology and capabilities.
- Counter and BER information – Retrieves details related to link statistics and Bit Error Rate (BER).

- FEC histogram information – Retrieves details related to the link's Forward Error Correction (FEC) histogram.
- Management cable information – Retrieves information related to the cable EEPROM module in raw format (single page or all related pages) or limited to cable Digital Diagnostic Monitoring (DDM).
- System temperature and power information – Retrieves information related to the system main temperature and power sensors.

To interact with a device (which typically corresponds to a specific port on a NIC), you must create a DOCA Telemetry PHY context using **doca_telemetry_phy_create()**.

Each context operates independently of standard DOCA PHY contexts. Consequently, changes made to PHY configurations are not automatically reflected within the telemetry context.

5.4.19.8.5 Configuration Phase

5.4.19.8.5.1 Device Support

DOCA Telemetry PHY requires a device to operate. For picking a device, refer to "DOCA Core Device Discovery".

As device capabilities may change, it is recommended to check your device using the **doca_telemetry_phy_is_supported()** method.

Each of the provided DOCA Telemetry PHY API functionalities can be checked independently using the method:

Functionality	Method
Operation information	**doca_telemetry_phy_cap_operation_info_is_supported()**
Module information	**doca_telemetry_phy_cap_module_info_is_supported()**
Supported information	**doca_telemetry_phy_cap_supported_info_is_supported()**
Troubleshooting information	**doca_telemetry_phy_cap_troubleshooting_info_is_supported()**
Counter and BER information	**doca_telemetry_phy_cap_counter_and_ber_info_is_supported()**
FEC histogram information	**doca_telemetry_phy_cap_fec_histogram_info_is_supported()**
Management cable information	**doca_telemetry_phy_cap_management_cable_is_supported()**
System temperature and power information	**doca_telemetry_phy_cap_system_temp_and_pwr_info_is_supported()**

5.4.19.8.5.2 Output Structure Format

The calling application is strictly responsible for allocating the necessary output structures. Refer to the `doca_telemetry_phy.h` header file for exact details on the output structures provided by the `doca_telemetry_phy` context.

> ⚠ Data Interpretation Rules
>
> - Returned values that fall outside the corresponding enumeration range or prescribed limit values must be interpreted as invalid or not applicable, unless expressly indicated otherwise.
> - Returned values that are not explicitly defined, yet fall within the corresponding enumeration range, must be interpreted as reserved or unknown, unless expressly indicated otherwise.

Output Data Fields

The following table details the specific data fields retrieved by the Telemetry PHY context based on the requested functionality category:

Category	Field / Functionality	Description
Operation	`active_protocol`	Active protocol (InfiniBand or Ethernet).
	`state`	Firmware PHY manager FSM state.
	`phy_state`	Physical state (related to InfiniBand or Ethernet).
	`link_speed_active`	Link speed active (related to InfiniBand or Ethernet).
	`link_width`	Link width.
	`fec_mode_active`	Active FEC mode.
	`loopback_mode`	Loopback mode.
	`auto_negotiation`	Auto-negotiation status.
Supported	`active_protocol`	Active protocol (InfiniBand or Ethernet).
	`enabled_link_speed`	Enabled link speed (related to InfiniBand or Ethernet).
	`supported_cable_speed`	Supported link speed (related to InfiniBand or Ethernet).
Troubleshooting	`status_opcode_raw`	Status operation code raw value.
	`status_opcode`	Status operation code.
	`group_opcode`	Group operation code.
	`status_message`	Status message.

Category	Field / Functionality	Description
Module	number_of_lanes	Number of supported module lanes.
	nominal_bit_rate	Nominal bit rate in Gb/s.
	active_protocol	Active protocol (InfiniBand or Ethernet).
	error_code	Error code response for Control or Set configuration of the data path.
	cable_vendor_info	Cable vendor ID, name, manufacture date, part, serial, and revision numbers.
	cable_general_properties_info	Static information (Memory map, cable technology, ID, type, compliance code).
	cable_power_and_temp_info	Cable power and temperature info.
	active_cable_info	Active cable information (cable emphasis, wavelength).
	error_counter_info	Error counter information (latched Tx fault, Rx loss of signal, flags per lane).
	status_info	Status information for the plugged cable.
	latency_info	Latency information.
Counter & BER	number_of_lanes	Number of supported module lanes.
	link_down_counter	Unintentional link drops counter (no remote consideration).
	link_down_recovery_counter	Successful recovery events per active link.
	time_since_last_clear	Time passed since the last counters clear event (in msec).
	active_protocol	Active protocol (InfiniBand or Ethernet).
	symbol_errors	Error counter caused by invalid bits that were not corrected by PHY correction mechanisms (Valid for InfiniBand only).
	effective_errors	Error counter caused by invalid bits that were not corrected by the FEC algorithm.
	raw_errors	Raw error counter caused by invalid bits received.
FEC Histogram	num_of_bins	Available number of bins.
	bin_range	Range of bin errors distribution.
	bin_errors	Bin errors distribution according to bin_range.
Cable Dump	num_of_pages	Number of pages returned.
	page_info	Returned page information (page ID, bytes read from offsets 0 and 128, and size of returned data).
Cable DDM	number_of_lanes	Number of supported module lanes.

Category	Field / Functionality	Description
	`module_temperature`	Temperature parameters (high/low alarm flags, current value, and thresholds in Celsius).
	`module_voltage`	Module voltage parameters (high/low alarm flags, current value, and thresholds in mV).
	`rx_power`	RX power parameters (high/low alarm flags, current value, and thresholds per lane in dBm).
	`tx_power`	TX power parameters (high/low alarm flags, current value, and thresholds per lane in dBm).
	`tx_bias`	TX bias parameters (high/low alarm flags, current value, and thresholds per lane in mA).
System Temperature and Power	`power`	Power reading from the sensor in Watts.
	`temperature`	Current temperature reading from the sensor in Celsius.
	`max_temperature`	The highest historic measured temperature from the sensor in Celsius.
	`temperature_sensor_name`	Temperature sensor name.
	`power_sensor_name`	Power sensor name.

5.4.19.8.6 Execution Phase

To retrieve telemetry data during the execution phase, pass your established context and the target output structure to the appropriate retrieval API.

> **Warning: Memory Allocation**
>
> For every function listed below, you must ensure that memory is properly allocated for the output structure *before* calling the API.

Information Category	Retrieval API
Operation	`doca_telemetry_phy_get_operation_info(context, &operation_info);`
Supported	`doca_telemetry_phy_get_supported_info(context, &supported_info);`
Troubleshooting	`doca_telemetry_phy_get_troubleshooting_info(context, &troubleshooting_info);`
Module	`doca_telemetry_phy_get_module_info(context, &module_info);`

Information Category	Retrieval API
Counter and BER	`doca_telemetry_phy_get_counter_and_ber_info(context, &counter_and_ber_info);`
FEC histogram	`doca_telemetry_phy_get_fec_histogram_info(context, &fec_histogram_info);`
Cable single page	`doca_telemetry_phy_get_management_cable_single_page_info(context, &management_cable_raw_info);`
Cable dump	`doca_telemetry_phy_get_management_cable_dump_info(context, &management_cable_raw_info);`
Cable DDM	`doca_telemetry_phy_get_management_cable_ddm_info(context, &management_cable_ddm_info);`
System Temperature and Power	`doca_telemetry_phy_get_system_temp_and_pwr_info(context, &system_temp_and_pwr_info);`

5.4.19.8.7 State Machine

The **doca_telemetry_phy** context transitions through specific operational states. This section outlines these states, the operations permitted within them, and how to transition between them.

5.4.19.8.7.1 Idle

The context has been created and is idle.

In this state, it is expected for the application to:

- Destroy the context
- Start the context for processing

Allowed operations:

- Configuring the context according to section "Configuration Phase"

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop

5.4.19.8.7.2 Running

In this state it is expected for the application to:

1. Stop the context.
2. Retrieve operation information.
3. Retrieve supported information.
4. Retrieve troubleshooting information.
5. Retrieve module information.
6. Retrieve counter and BER information.
7. Retrieve FEC histogram information.
8. Retrieve management cable single page information.
9. Retrieve management cable dump information.
10. Retrieve management cable DDM information.
11. Retrieve system temperature and power information.

Allowed operations:

- Calling stop, moving the context to "Idle" state

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Successfully start the context

There are currently no state restrictions on the majority of API functions.

5.4.19.8.8 Alternative Datapath Options

DOCA Telemetry PHY supports only CPU-based datapaths.

5.4.19.8.9 DOCA Telemetry Phy Sample

5.4.19.8.9.1 Running the Sample

1. Refer to the following documents:
   a. DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   b. NVIDIA BlueField Platform Software Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

```
cd /opt/mellanox/doca/samples/doca_telemetry/telemetry_phy
meson /tmp/build
ninja -C /tmp/build
```

The binary **doca_telemetry_phy** is created under **/tmp/build/**.

Sample usage:

```
Usage: doca_telemetry_phy [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>

Program Flags:
  -p, --pci-addr                    DOCA device PCI device address
  -oi, --get-operation-info         Retrieve operation info
  -si, --get-supported-info         Retrieve supported info
  -ti, --get-troubleshooting-info   Retrieve troubleshooting info
  -mi, --get-module-info            Retrieve module info
  -bi, --get-counter-ber-info       Retrieve counter and BER info
  -fi, --get-fec-histogram-info     Retrieve FEC Histogram info
  -mcspi, --get-management-cable-single-page-info Retrieve management cable single page info
  -mcdi, --get-management-cable-dump-info Retrieve management cable dump info
  -mcddmi, --get-management-cable-ddm-info Retrieve management cable DDM info
  -stpi, --get-system-temp-and-power-info Retrieve system temperature and power info
```

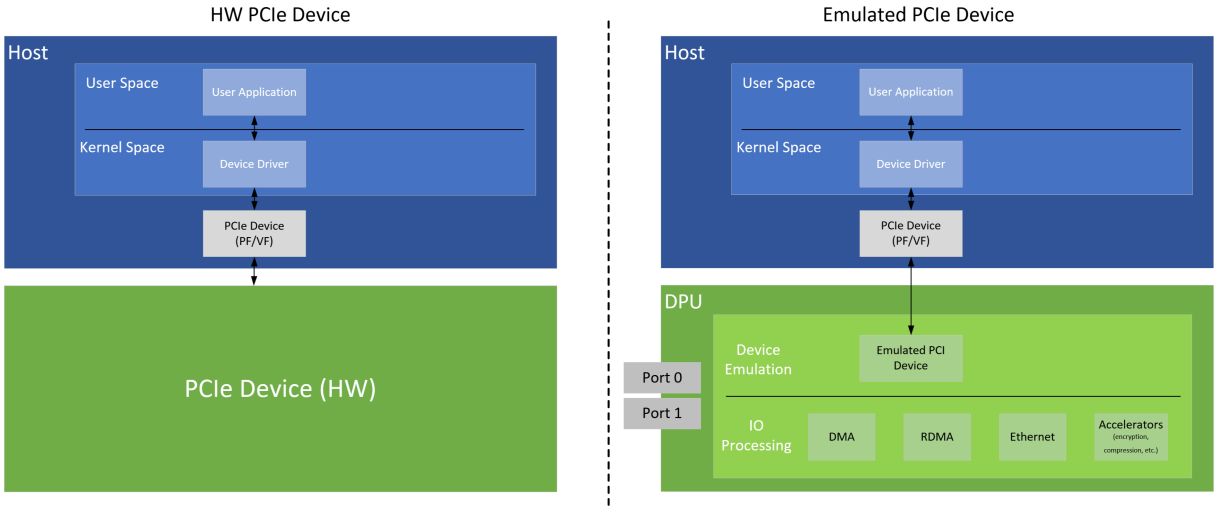
The sample includes:

1. Locating and opening a DOCA device.
2. Creating a **doca_telemetry_phy** instance.
3. Retrieving and printing operation information.
4. Retrieving and printing supported information.
5. Retrieving and printing troubleshooting information.
6. Retrieving and printing module information.
7. Retrieving and printing counter and BER information.
8. Retrieving and printing FEC histogram information.
9. Retrieving and printing management cable single page information.
10. Retrieving and printing management cable dump information.
11. Retrieving and printing management cable DDM information.
12. Retrieving and printing system temperature and power information.
13. Destroying the **doca_telemetry_phy** context.

5.4.20 DOCA Device Emulation

5.4.20.1 Introduction

NVIDIA® BlueField® networking platforms (DPUs or SuperNICs) provide the ability to emulate a PCIe device. The DOCA Device Emulation subsystem provides a low-level software API for users to develop PCIe devices and their controllers. These APIs include discovery, configuration, hot plugging/unplugging, management, and IO path handling. In simpler terms, the libraries enable the user to implement a hardware PCIe function using software, such that the host is not aware that the PCIe function is emulated, and all interactions from the host are routed to software on the BlueField instead of actual hardware.



The diagram shows the potential for device emulation to replace a regular PCIe function of some PCIe device.

- On the left is a conventional setup where the host is connected to a PCIe device (e.g., NVMe SSD). On the host, user applications interact with the kernel driver of that device, using some software interface, and the driver communicates with the hardware/firmware of the device.
- On the right is a setup where the PCIe device is replaced with a BlueField with an application using DOCA Device Emulation. The application can use the DOCA DevEmu PCI library to control the device, and intercept any IOs written by the host to the PCIe device. Additionally, the application can use other DOCA libraries to perform IO processing (e.g., copying data from host memory using DMA, sending RDMA/Ethernet traffic) and other acceleration libraries for encryption, compression, etc.

5.4.20.2 Known Limitations

- This library is supported at alpha level; backward compatibility is not guaranteed
- VFs are not currently supported
- Some limitations apply when creating a generic emulated function, for more details refer to DOCA DevEmu PCI Generic Limitations.
- Consult your NVIDIA representative for limitations on the emulated device's behavior

5.4.20.3 DOCA DevEmu PCI

5.4.20.3.1 Introduction

> The quality status of DOCA libraries is listed here.

DOCA DevEmu PCI is part of the DOCA Device Emulation subsystem. It provides low-level software APIs that allow management of an emulated PCIe device using the emulation capability of NVIDIA® BlueField® networking platforms.

It is a common layer for all PCIe emulation modules, such as DOCA DevEmu PCIe Generic Emulation, and DOCA DevEmu Virtio subsystem emulation.

5.4.20.3.2 Prerequisites

This library follows the architecture of a DOCA Core Context. It is recommended read the following sections beforehand:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

Generic device emulation is part of DOCA device emulation. It is recommended to read the following guides beforehand:

- DOCA Device Emulation

5.4.20.3.3 Environment

DOCA DevEmu PCI Emulation is supported only on the BlueField target. The BlueField must meet the following requirements

- DOCA version 2.7.0 or greater
- BlueField-3 firmware 32.41.1000 or higher

> Please refer to the DOCA Compatibility Policy.

The library must be run with root privileges.

Perform the following:

1. Configure the BlueField to work in DPU mode as described in *BlueField Modes of Operation*.
2. Enable the PCIe switch emulation capability needed for hot plugging emulated PCIe devices. This can be done by running the following command on the host or BlueField:

```
host/bf> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s PCI_SWITCH_EMULATION_ENABLE=1
```

3. Perform a BlueField system-level reset for the **mlxconfig** settings to take effect.

To support hot-plug feature, the host must have the following boot parameters:

- Intel CPU:

```
intel_iommu=on iommu=pt pci=realloc
```

- AMD CPU:

```
iommu=pt pci=realloc
```

This can be done using the following steps:

This process may vary depending on the host OS. Users can find multiple guides online describing this process.

1. Add the boot parameters:

```
host> sudo nano /etc/default/grub
Find the variable
GRUB_CMDLINE_LINUX_DEFAULT="<existing-params>"
Add the params at the end
GRUB_CMDLINE_LINUX_DEFAULT="<existing-params> intel_iommu=on iommu=pt pci=realloc"
```

2. Update configuration.
   - For Ubuntu:

     ```
     host> update-grub
     ```

   - For RHEL:

     ```
     host> grub2-mkconfig -o /boot/grub2/grub.cfg
     ```

3. Perform warm boot.
4. Confirm that the parameters are in effect:

```
host> cat /proc/cmdline
<existing-params> intel_iommu=on iommu=pt pci=realloc
```

5.4.20.3.4 Architecture

The DOCA DevEmu PCI library provides two main software abstractions:

- The PCIe type
- The PCIe endpoint

The PCIe type represents the configurations of the emulated device, while the PCIe endpoint represents an instance of an emulated device. Furthermore, any PCIe endpoint instance must be associated with a single PCIe type, while a PCIe type can be associated with many PCIe endpoints.

The DOCA DevEmu PCI endpoint serves as a base structure for all DOCA DevEmu library devices (such as the DOCA DevEmu PCI Device for generic emulation and the DOCA DevEmu PCI TLP Device for TLP emulation). The endpoint can be retrieved using the **`doca_devemu_pci_*dev_as_ep`** API.

The rest of this document mostly describes the features and characteristics of the DOCA DevEmu PCI Device object (generic emulation). Only the following sections apply to all objects:

- Pre-defined PCI Type vs. Generic/TLP PCI Type
- PCIe Type Name
- Create Emulated Device
- Emulated Device Discovery
- Objects Lifecycle and Persistency
- SR-IOV
- Device Support

5.4.20.3.4.1 Pre-defined PCI Type vs. Generic/TLP PCI Type

A PCIe type object can be acquired in 2 different ways:

- Acquire a pre-defined type, using emulation libraries of existing protocols such as DOCA DevEmu Virtio FS library
- Create from scratch using the DOCA DevEmu PCI Generic library or the DOCA DevEmu PCI TLP library

In case of pre-defined type, the configurability of the type is limited.

5.4.20.3.4.2 PCIe Type Name

As part of the DOCA PCIe emulation, every type has a name assigned to it. This property is not part of the PCIe specification, but rather a mechanism in DOCA that uniquely identifies the PCIe type.

There cannot be 2 different PCIe types with the same name, even across different processes, unless the type in the second process is configured in identical manner to the first one. Furthermore, attempting to configure the second type with same name but with slight configuration difference will fail.

5.4.20.3.4.3 Create Emulated Device

After configuring the desired DOCA DevEmu PCIe type, it is possible to create an emulated device based on the configured type using **doca_devemu_pci_type_create_rep**. This sequential process ensures that the DOCA DevEmu PCIe endpoint is created with the specified parameters and configuration defined by the PCIe type object. Furthermore, it is possible to destroy the emulated device using **doca_devemu_pci_type_destroy_rep**.

Specifically when the device is a DOCA DevEmu PCI Device, the created device representor starts in "power_off" state and is not visible to the host until hot-plug sequence is issued by the user, see Hot-plug Emulated Device. The device can then be destroyed only while in "power_off" state.

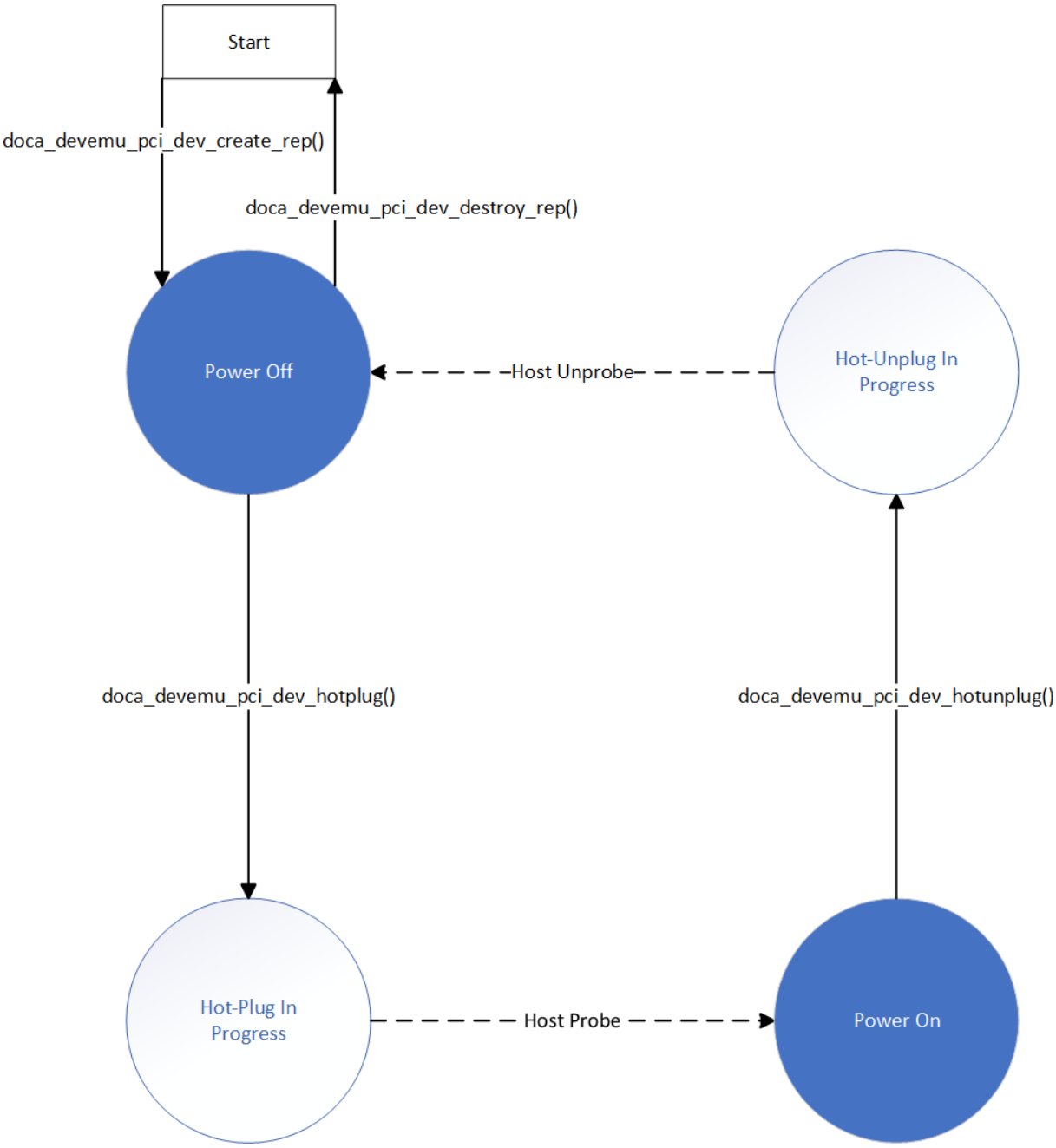
> The created emulated device may outlive the application that created it, see Objects Lifecycle and Persistency.

Extension

The emulated device can be created with extended configurations, such as setting an Expansion ROM BAR and configuring an SR-IOV PF, using **doca_devemu_pci_type_create_rep_ex**.

5.4.20.3.4.4 Hot-plug Emulated Device

Hot-plugging refers to the process of emulating the physical attachment of a PCIe device to the host PCIe subsystem after the system has been powered on and initialized. Note that some operating systems require additional settings to enable the process of hot-plugging a PCIe device. For supported systems, this feature proves particularly advantageous for systems that need to remain operational at all times while expanding their hardware resources, such as additional storage and networking capabilities. DOCA DevEmu PCI provides software APIs that allow users to emulate this process in an asynchronous manner.



When creating a PCIe device object, if it starts in "power off" state, then the device is not yet visible to the host. It is possible then, from the BlueField, to hot-plug the device. This starts an async process of the device getting hot-plugged towards the host. Once the process completes, the emulated device transitions to "power on" and becomes visible to the host. Usually at this stage, the emulated device receives its BDF address. The hot-unplug process works in similar async manner.

Using DOCA API, the BlueField Arm can register to any changes to the hot-plug state of each emulated device using **doca_devemu_pci_dev_event_hotplug_state_change_register**.

5.4.20.3.4.5 Emulated Device Discovery

The emulated device is represented as a `doca_devinfo_rep`. It is possible to iterate through all the emulated devices as explained in DOCA Core Representor Discovery.

There are 2 ways of filtering the list of emulated devices:

- Get all emulated devices – use `DOCA_DEVINFO_REP_FILTER_EMULATED` as the filter argument in `doca_devinfo_rep_create_list`
- Get all emulated devices that belong to a certain type – `doca_devemu_pci_type_create_rep_list`

5.4.20.3.4.6 Objects Lifecycle and Persistency

This section creates distinction between firmware resources and software resources:

- Firmware resources persist until the next power cycle, and can be accessible from different processes on the BlueField Arm. Such resources are not cleared once the application exits.
- Software resources are representations of firmware resources, and are only relevant for the same process.

Using this terminology, it is possible to describe the objects as follows:

- The PCIe type object `doca_devemu_pci_type` represents a PCIe type firmware resource. The resource persists if any of the following apply:
  - There is at least 1 process holding reference to the PCIe type
  - There is at least 1 PCIe device firmware resource belonging to this type
- The emulated device representor, `doca_devinfo_rep`, represents an emulated PCIe function firmware resource:
  - `doca_devemu_pci_type_create_rep` can be used to create such firmware resource
  - To destroy the firmware resource, `doca_devemu_pci_type_destroy_rep` can be used
  - For static functions, the representor resource persists until configured otherwise in NVCONFIG
  - To find existing PCIe device firmware resources, use `doca_devemu_pci_type_create_rep_list`

5.4.20.3.4.7 Function-level Reset

The created emulated devices support PCIe function level reset (FLR).

Using DOCA API, the BlueField Arm can register to FLR event using `doca_devemu_pci_dev_event_flr_register`. Once the driver requests FLR, this event is triggered, calling the user provided callback.

Once FLR is detected, it is expected for the BlueField Arm to do the following:

- Destroy all resources related to the PCIe device. For information on such resources, refer to the guide of concrete PCIe type (generic/virtiofs).
- Stop the PCIe device
- Start the PCIe device again

5.4.20.3.4.8 PCIe Resources

It is possible to query the number of available PCIe emulation resources. The resources that can be queried are:

- Number of doorbells
- Number of MSI-X

These resources are globally shared across the system between all emulated devices that are created using the same **doca_dev**.

5.4.20.3.4.9 SR-IOV

The library allows exposing an emulated Physical Function (PF) that carries an SR-IOV Extended Capability, allowing the host to enable a number of Virtual Functions (VFs) on top of the PF. SR-IOV support is a per-type capability and is queried using **doca_devemu_pci_*cap_type_is_dynamic_sriov_supported**.

Creating an SR-IOV PF

An SR-IOV PF is created by configuring a set of SR-IOV attributes on a **doca_devemu_pci_rep_init_attr** object and then creating the PF representor with **doca_devemu_pci_type_create_rep_ex**. Configuring any of these attributes requires **doca_devemu_pci_*cap_type_is_dynamic_sriov_supported** to return 1. The available attributes are:

- TotalVFs – **doca_devemu_pci_rep_init_attr_set_total_vfs** – the number of VFs the PF can support, reflected in its SR-IOV TotalVFs field. Bounded by **doca_devemu_pci_cap_get_max_total_vfs**. All attributes below are valid only when TotalVFs > 0.
- VF stride – **doca_devemu_pci_rep_init_attr_set_vf_stride** – bounded by **doca_devemu_pci_cap_get_max_vf_stride**.
- First VF offset – **doca_devemu_pci_rep_init_attr_set_first_vf_offset** – bounded by **doca_devemu_pci_cap_get_max_first_vf_offset**.
- First dynamic VF index – **doca_devemu_pci_rep_init_attr_set_first_dynamic_vf_idx** – the starting index of the PF's contiguous range within the global dynamic VF pool (see below). The PF is assigned the VF indices [first_dynamic_vf_idx, first_dynamic_vf_idx + TotalVFs - 1].
- MSI-X vectors per VF – **doca_devemu_pci_rep_init_attr_set_vf_num_msix** – the initial number of MSI-X vectors allocated to each VF. bounded by **doca_devemu_pci_cap_type_get_max_num_msix**.
- Doorbells per VF – **doca_devemu_pci_rep_init_attr_set_vf_num_db** – the initial number of doorbells allocated to each VF. bounded by **doca_devemu_pci_cap_type_get_max_num_db**.

Dynamic VF Pool

VFs are drawn from a global dynamic VF pool of indices [0, pool_size - 1] that is shared across all PF representors. The pool size is queried with **doca_devemu_pci_cap_get_dynamic_vf_pool_size**. Each PF is assigned a contiguous sub-range of this pool via its first dynamic VF index and TotalVFs, and the same VF index cannot be assigned to more than one PF at the same time.

The MSI-X vectors and doorbells assigned to VFs are taken from the globally shared PCIe emulation resources (see PCIe Resources). The total number of MSI-X vectors and doorbells available on the device can be obtained by acquiring a **doca_devemu_pci_resources** object with **doca_devemu_pci_get_available_resources** and reading **doca_devemu_pci_resources_get_num_msix** and **doca_devemu_pci_resources_get_num_db**.

Dedicated Operations

Once an SR-IOV PF endpoint is started, the following operations are available on it:

- **doca_devemu_pci_ep_get_total_vfs** – the PF's TotalVFs value (0 indicates the endpoint is not an SR-IOV PF).
- **doca_devemu_pci_ep_get_num_vfs** – the current NumVFs value as last written by the host, i.e., how many VFs are currently enabled.
- **doca_devemu_pci_ep_get_num_free_dynamic_vf_msix** – the number of MSI-X vectors the PF can still assign to its VFs.
- **doca_devemu_pci_ep_get_num_free_dynamic_vf_db** – the number of doorbells the PF can still assign to its VFs.
- **doca_devemu_pci_ep_reset_msix_resources** – reset the MSI-X resources of the PF's VFs. Requires SR-IOV to be disabled and all associated VF endpoints to be stopped.
- **doca_devemu_pci_ep_reset_db_resources** – reset the doorbell resources of the PF's VFs, under the same preconditions.

5.4.20.3.5 Device Support

DOCA PCIe Device emulation requires a device to operate. For picking a device, see DOCA Core Device Discovery.

The device emulation library is supported only on BlueField-3 and later.

As device capabilities may change in the future (see Capability Checking), it is recommended that users choose a device using the following method:

- **doca_devemu_*_cap_is_*hotplug_supported** – for create and hot-plug support
- **doca_devemu_*_cap_is_*type_supported** – for device discovery only

5.4.20.3.6 PCIe Device

5.4.20.3.6.1 Configuration Phase

To start using the DOCA DevEmu PCI Device, users must first go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context to allow retrieval of events.

Configurations

The context can be configured to match the application use case.

To find if a configuration is supported or what its min/max value is, refer to Device Support.

Mandatory Configurations

All mandatory configurations are provided during the creation of the PCIe device.

These configurations are as follows:

- A DOCA DevEmu PCIe type object
- A DOCA Device Representor, representing an emulated function with the same type as the provided PCIe object type
- A DOCA Progress Engine object

Optional Configurations

These configurations are optional. If not set, then a default value is used:

- Registering to events as described in the "Events" section. By default, the user does not receive events
- The PCIe device ID. By default, it is derived from the PCIe type.
- The PCIe vendor ID. By default, it is derived from the PCIe type.
- The PCIe subsystem ID. By default, it is derived from the PCIe type.
- The PCIe subsystem vendor ID. By default, it is derived from the PCIe type.
- The PCIe revision ID. By default, it is derived from the PCIe type.
- The PCIe class code. By default, it is derived from the PCIe type.
- The number of MSI-X vectors for MSI-X capability. By default, it is derived from the PCIe type.

5.4.20.3.6.2 Execution Phase

This section describes execution on CPU using DOCA Core Progress Engine.

Events

The DOCA DevEmu PCI device exposes asynchronous events to notify about sudden changes according to DOCA Core architecture.

Common events are described in DOCA Core Event.

Hotplug State Change

The hotplug state change event allows users to receive notifications whenever the hotplug state of the emulated device changes. See section "Hot-plug Emulated Device".

Event Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	`doca_devemu_pci_dev_event_hotplug_state_change_register`	`doca_devemu_*_cap_is_*hotplug_supported`

Event Trigger Condition

The event is triggered anytime an asynchronous transition happens as follows:

- `DOCA_DEVEMU_PCI_HP_STATE_PLUG_IN_PROGRESS` → `DOCA_DEVEMU_PCI_HP_STATE_POWER_ON`

- `DOCA_DEVEMU_PCI_HP_STATE_UNPLUG_IN_PROGRESS` → `DOCA_DEVEMU_PCI_HP_STATE_POWER_OFF`
- `DOCA_DEVEMU_PCI_HP_STATE_POWER_ON` → `DOCA_DEVEMU_PCI_HP_STATE_UNPLUG_IN_PROGRESS` (when initiated by the host)

Any transition initiated by user is not triggered (e.g., calling hotplug to transition from `POWER_OFF` to `PLUG_IN_PROGRESS`).

The following APIs can be used to initiate hotplug or hot-unplug transition processes:

- `doca_devemu_pci_dev_hotplug`
- `doca_devemu_pci_dev_hotunplug`

Event Output

Common output as described in DOCA Core Event.

Additionally, the internal cached hotplug state is updated and can be fetched using `doca_devemu_pci_dev_get_hotplug_state`.

Event Handling

Once the event is triggered, it means that the hotplug state has changed. The application is expected to do the following:

- Retrieve the new hotplug state using `doca_devemu_pci_dev_get_hotplug_state`

Function-level Reset

The FLR event allows users to receive notifications whenever the host initiates an FLR flow. See section "Function Level Reset".

Event Configuration

Description	API to Set the Configuration
Register to the event	`doca_devemu_pci_dev_event_flr_register`

Event Trigger Condition

The event is triggered anytime the host driver initiates an FLR flow. See section "Function Level Reset".

Event Output

Common output as described in DOCA Core Event.

Additionally, the internal cached FLR indicator is updated and can be fetched using `doca_devemu_pci_dev_is_flr`.

Event Handling

Once the event is triggered, it means that the host driver has initiated the FLR flow.

The user must handle the FLR flow by doing the following:

1. Flush all the outstanding requests back to the associated resource

2. Release all the PCIe device resources dynamically created after device start
3. Stop the PCIe device – **doca_ctx_stop**
4. Start the PCIe device again – **doca_ctx_start**
    - Call **doca_pe_progress** repeatedly until the PCIe device transitions to "running" state

For more information on starting the PCIe device again, refer to section "State Machine".

SR-IOV

The SR-IOV event allows users to receive notifications whenever the host enables or disables SR-IOV on the PF by writing the NumVFs register. See section "SR-IOV".

Event Configuration

Description	API to Set the Configuration
Register to the event	**doca_devemu_pci_dev_event_sriov_register**

Event Trigger Condition

The event is triggered whenever the host writes the SR-IOV NumVFs register – when enabling SR-IOV (writing a non-zero number of VFs) or disabling it (writing 0).

Event Output

Common output as described in DOCA Core Event.

Additionally, the handler receives the previous and current values of the NumVFs register. The current value can also be fetched using **doca_devemu_pci_ep_get_num_vfs** .

Event Handling

Once the event is triggered, the number of VFs visible on the PCIe bus has changed. Based on the previous and current NumVFs values, the application should bring up the resources for the newly enabled VFs, or tear down the resources of VFs that were disabled.

5.4.20.3.6.3 State Machine

The DOCA DevEmu PCI device object follows the context state machine as described in DOCA Core Context State Machine.

The following section describes how to transition to any state and what is allowed in each state.

Idle

In this state, it is expected that application either:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to section "Configurations"
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop after making sure all resources have been destroyed
Stopping	Call progress until all resources have been destroyed

Starting

In this state, it is expected that application:

- Calls progress to allow transition to next state
- Keeps context in this state until FLR flow is complete

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after receiving FLR event (i.e., while FLR is in progress)

Running

In this state, it is expected that application:

- Calls progress to receive events
- Creates/destroys PCIe device resources

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after configuration
Starting	Call progress until FLR flow is completed

Stopping

In this state, it is expected that application:

- Destroys all emulated device resources as described in section "Function Level Reset".

Allowed operations:

- Destroying PCIe device resources

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call stop without freeing emulated device resources

5.4.20.3.7 DOCA DevEmu PCI Generic

This library is supported at the alpha level; backward compatibility is not guaranteed.

This guide provides instructions on building and developing applications that require emulation of a generic PCIe device.

5.4.20.3.7.1 Introduction

DOCA DevEmu PCI Generic is part of the DOCA Device Emulation subsystem. It provides low-level software APIs that allow creation of a custom PCIe device using the emulation capability of NVIDIA® BlueField®.

For example, it enables emulating an NVMe device by creating a generic emulated device, configuring its capabilities and BAR to be compliant with the NVMe spec, and operating it from the DPU as necessary.

5.4.20.3.7.2 Prerequisites

This library follows the architecture of a DOCA Core Context. It is recommended read the following sections beforehand:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

Generic device emulation is part of DOCA PCIe device emulation. It is recommended to read the following guides beforehand:

- DOCA Device Emulation
- DOCA DevEmu PCI

5.4.20.3.7.3 Environment

DOCA DevEmu PCI Generic Emulation is supported only on the BlueField target. The BlueField must meet the following requirements:

- DOCA version 2.7.0 or greater
- BlueField-3 firmware 32.41.1000 or higher

> Please refer to the DOCA Compatibility Policy.

Library must be run with root privileges.

Please refer to DOCA DevEmu PCI Environment, for further necessary configurations.

5.4.20.3.7.4 Architecture

DOCA DevEmu PCI Generic allows the creation of a generic PCI type. The PCI Type is part of the DOCA DevEmu PCI library. It is the component responsible for configuring the capabilities and bar layout of emulated devices.

The PCI Type can be considered as the template for creating emulated devices. Such that the user first configures a type, and then they can use it to create multiple emulated devices that have the same configuration.

For a more concrete example, consider that you would like to emulate an NVMe device, then you would create a type and configure its capabilities and BAR to be compliant with the NVMe spec, after that you can use the same type, to generate multiple NVMe emulated devices.

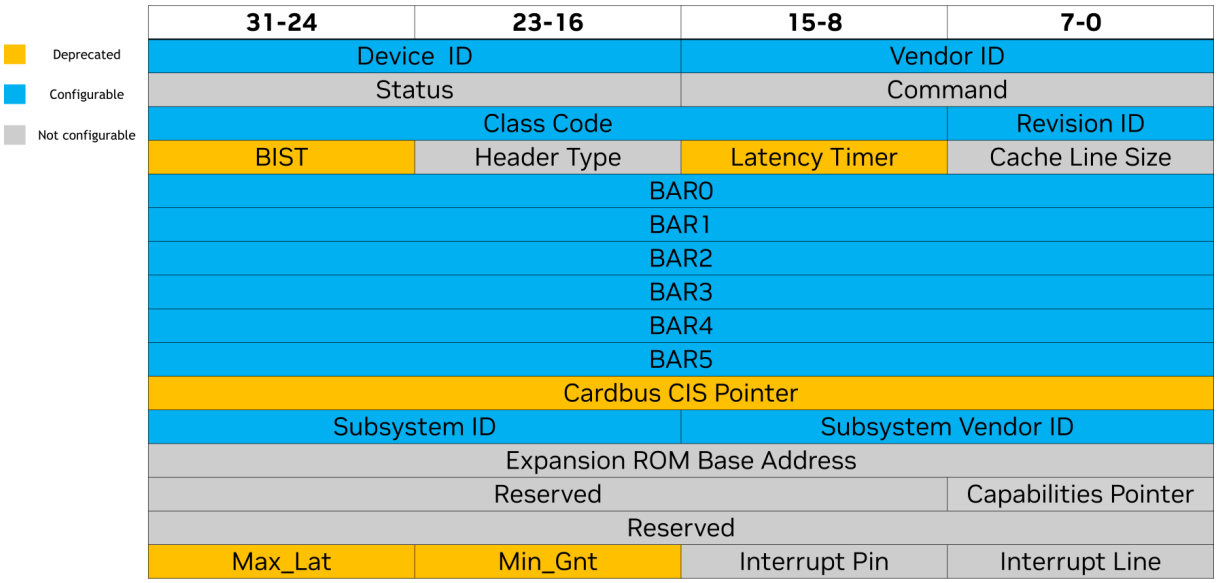
5.4.20.3.7.5 PCIe Configuration Space

The PCIe configuration space is 256 bytes long and has a header that is 64 bytes long. Each field can be referred to as a register (e.g., device ID).

Every PCIe device is required to implement the PCIe configuration space as defined in the PCIe specification.

The host can then read and/or write to registers in the PCIe configuration space. This allows the PCIe driver and the BIOS to interact with the device and perform the required setup.

It is possible to configure registers in the PCIe configuration space header as shown in the following diagram:



0x0 is the only supported header type (general device).

The following registers are read-only, and they are used to identify the device:

Register Name	Description	Example
Class Code	Defines the functionality of the device Can be further split into 3 values {class : subclass: prog IF}	0x020000 Class: 0x02 (Network Controller) Subclass: 0x00 (Ethernet Controller) Prog IF: 0x00 (N/A)
Revision ID	Unique identifier of the device revision Vendor allocates ID by itself	0x01 (Rev 01)

Vendor ID	Unique identifier of the chipset vendor Vendor allocates ID from the PCI-SIG	0x15b3 Nvidia
Device ID	Unique identifier of the chipset Vendor allocates ID by itself	0xa2dc BlueField-3 integrated ConnectX-7 network controller
Subsystem Vendor ID	Unique identifier of the card vendor Vendor allocates ID from the PCI-SIG	0x15b3 Nvidia
Subsystem ID	Unique identifier of the card Vendor allocates ID by itself	0x0051

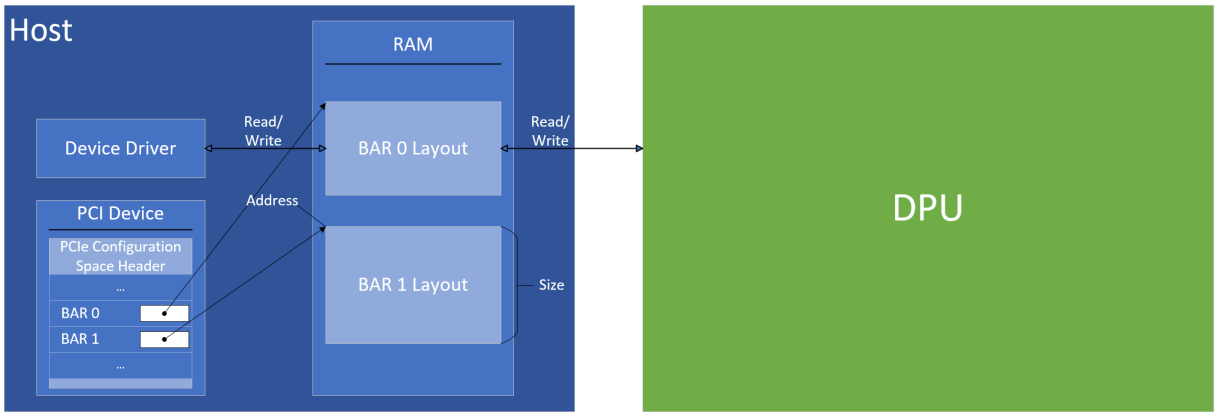
5.4.20.3.7.6 BAR

While the PCIe configuration space can be used to interact with the PCIe device, it is not enough to implement the functionality that is targeted by the device. Rather, it is only relevant for the PCIe layer.

To enable protocol-specific functionality, the device configures additional memory regions referred to as base address registers (BARs) that can be used by the host to interact with the device. Different from the PCIe configuration space, BARs are defined by the device and interactions with them is device-specific. For example, the PCIe driver interacts with an NVMe device's PCIe configuration space according to the PCIe spec, while the NVMe driver interacts with the BAR regions according to the NVMe spec.

Any read/write requests on the BAR are typically routed to the hardware, but in case of an emulated device, the requests are routed to the software.

The DOCA DevEmu PCI type library provides APIs that allow software to pick the mechanism used for routing the requests to software, while taking into consideration common design patterns utilized in existing devices.



Each PCIe device can have up to 6 BARs with varying properties. During the PCIe bus enumeration process, the PCIe device must be able to advertise information about the layout of each BAR. Based on the advertised information, the BIOS/OS then allocates a memory region for each BAR and assigns the address to the relevant BAR in the PCIe configuration space header. The driver can then use the assigned memory address to perform reads/writes to the BAR.

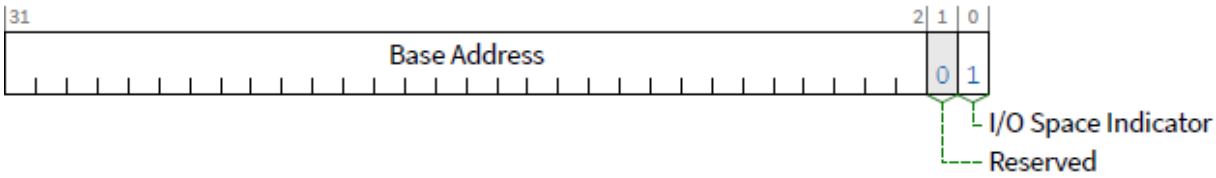
BAR Layout

The PCIe device must be able to provide information with regards to each BAR's layout.

The layout can be split into 2 types, each with their own properties as detailed in the following subsections.

I/O Mapped

According to the PCIe specification, the following represents the I/O mapped BAR:



Additionally, the BAR register is responsible for advertising the requested size during enumeration.

> The size must be a power of 2.

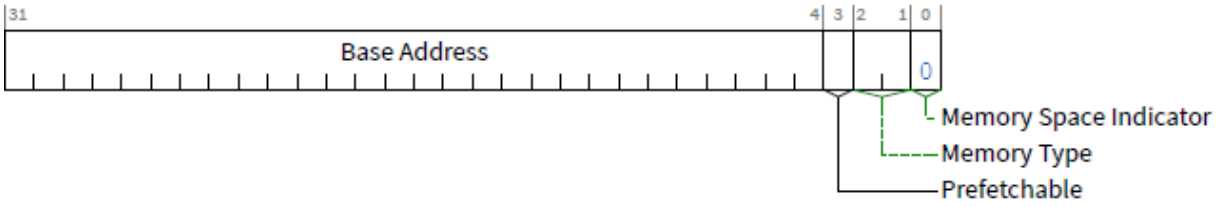
Users can use the following API to set a BAR as I/O mapped:

```
doca_devemu_pci_type_set_io_bar_conf(struct doca_devemu_pci_type *pci_type, uint8_t id, uint8_t log_sz)
```

- **id** – the BAR ID
- **log_sz** – the log of the BAR size

Memory Mapped

According to the PCIe specification, the following represents the memory mapped BAR:



Additionally, the BAR register is responsible for advertising the requested size during enumeration.

> The size must be a power of 2.

The memory mapped BAR allows a 64-bit address to be assigned. To achieve this, users must specify the bar Memory Type as 64-bit and then set the next BAR's (BAR ID + 1) size to be 0.

Setting the pre-fetchable bit indicates that reads to the BAR have no side effects.

Users can use the following API to set a BAR as memory mapped:

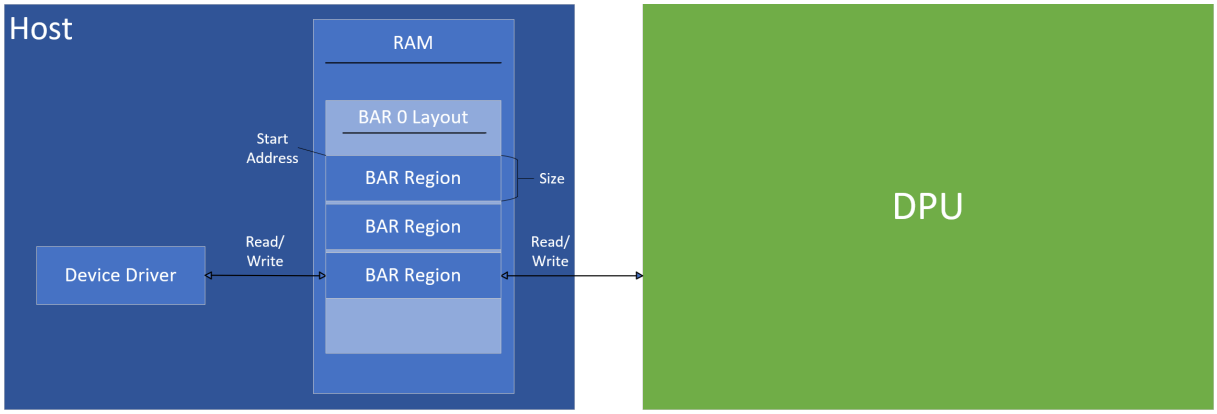
```
doca_devemu_pci_type_set_memory_bar_conf(struct doca_devemu_pci_type *pci_type, uint8_t id, uint8_t log_sz, enum
 doca_devemu_pci_bar_mem_type memory_type, uint8_t prefetchable)
```

- **id** – the BAR ID
- **log_sz** – the log of the BAR size. If set to 0, then the size is considered as 0 (instead of 1).
- **memory_type** – specifies the memory type of the BAR. If set to 64-bit, then the next BAR must have **log_sz** set to 0.
- **prefetchable** – indicates whether the BAR memory is pre-fetchable or not (a value of 1 or 0 respectively)

BAR Regions

BAR regions refer to memory regions that make up a BAR layout. This is not something that is part of the PCIe specification, rather it is a DOCA concept that allows the user to customize behavior of the BAR when interacted with by the host.

The BAR region defines the behavior when the host performs a read/write to an address within the BAR, such that every address falls in some memory region as defined by the user.



Common Configuration

All BAR regions have these configurations in common:

- **id** – the BAR ID that the region is part of
- **start_addr** – the start address of the region within the BAR layout relative to the BAR. 0 indicates the start of the BAR layout.
- **size** – the size of the BAR region

Currently, there are 4 BAR region types, defining different behavior:

- Stateful
- DB by offset
- DB by data
- MSIX table

- MSIX PBA

Generic Control Path (Stateful BAR Region)

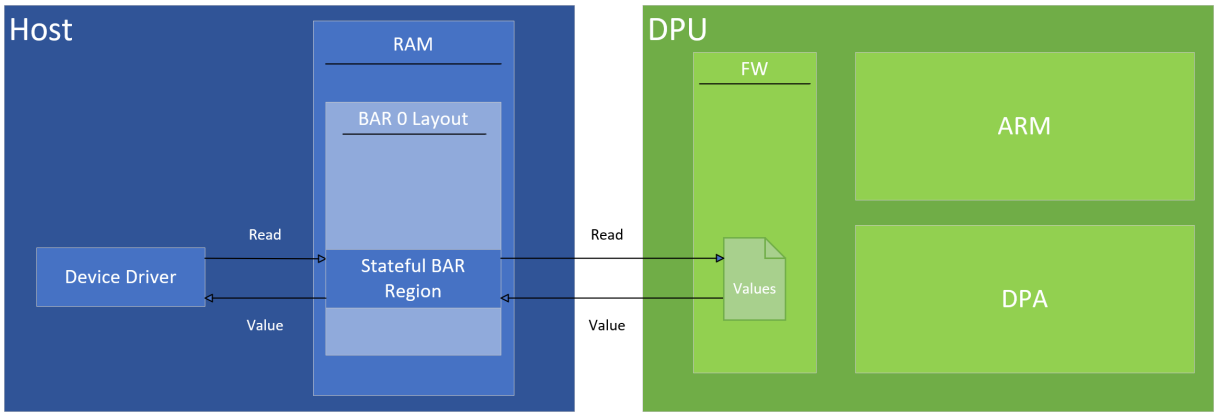
Stateful region can be used as a shared memory, such that the contents are maintained in firmware. A read from the driver returns the latest value, while a write updates the value and triggers an event to software running on the DPU.

This can be useful for communication between the driver and the device, during the control path (e.g., exposing capabilities, initialization).

> Some limitations apply, please see Limitations section

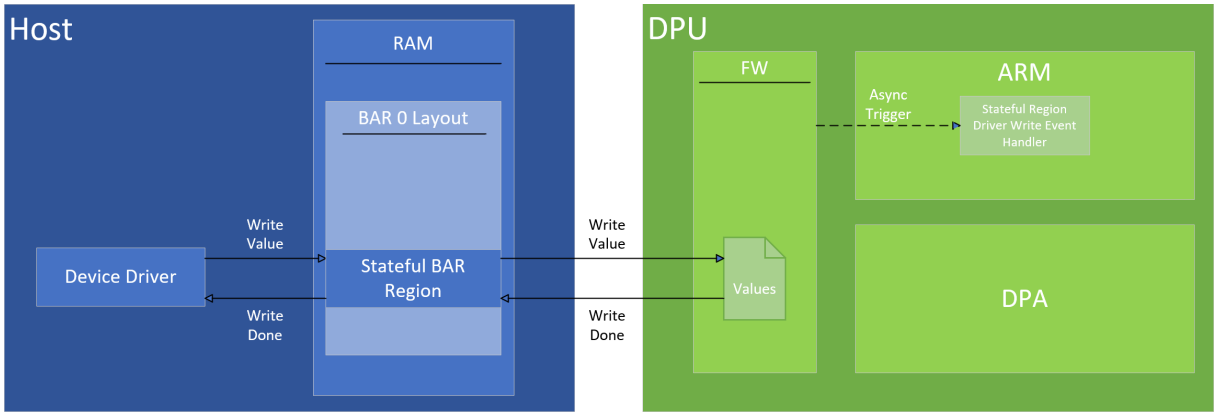
Driver Read

A read from the driver returns the latest value written to the region, whether written by the host or by the driver itself.



Driver Write

A write from the driver updates the value at the written address and notifies software running on the Arm that a write has occurred. The notification on the Arm arrives as an asynchronous event (see `doca_devemu_pci_dev_event_bar_stateful_region_driver_write`).



The event that arrives to Arm software is asynchronous such that it may arrive after the driver has completed the write.

DPU Read

The DPU can read the values of the stateful region using **doca_devemu_pci_dev_query_bar_stateful_region_values**. This returns the latest snapshot of the stateful region values. It can be particularly useful to find what was written by the driver after the "stateful region driver write event" occurs.

DPU Write

The DPU can write the values of the stateful region using **doca_devemu_pci_dev_modify_bar_stateful_region_values**. This updates the values such that subsequent reads from the driver or the DPU returns these values.

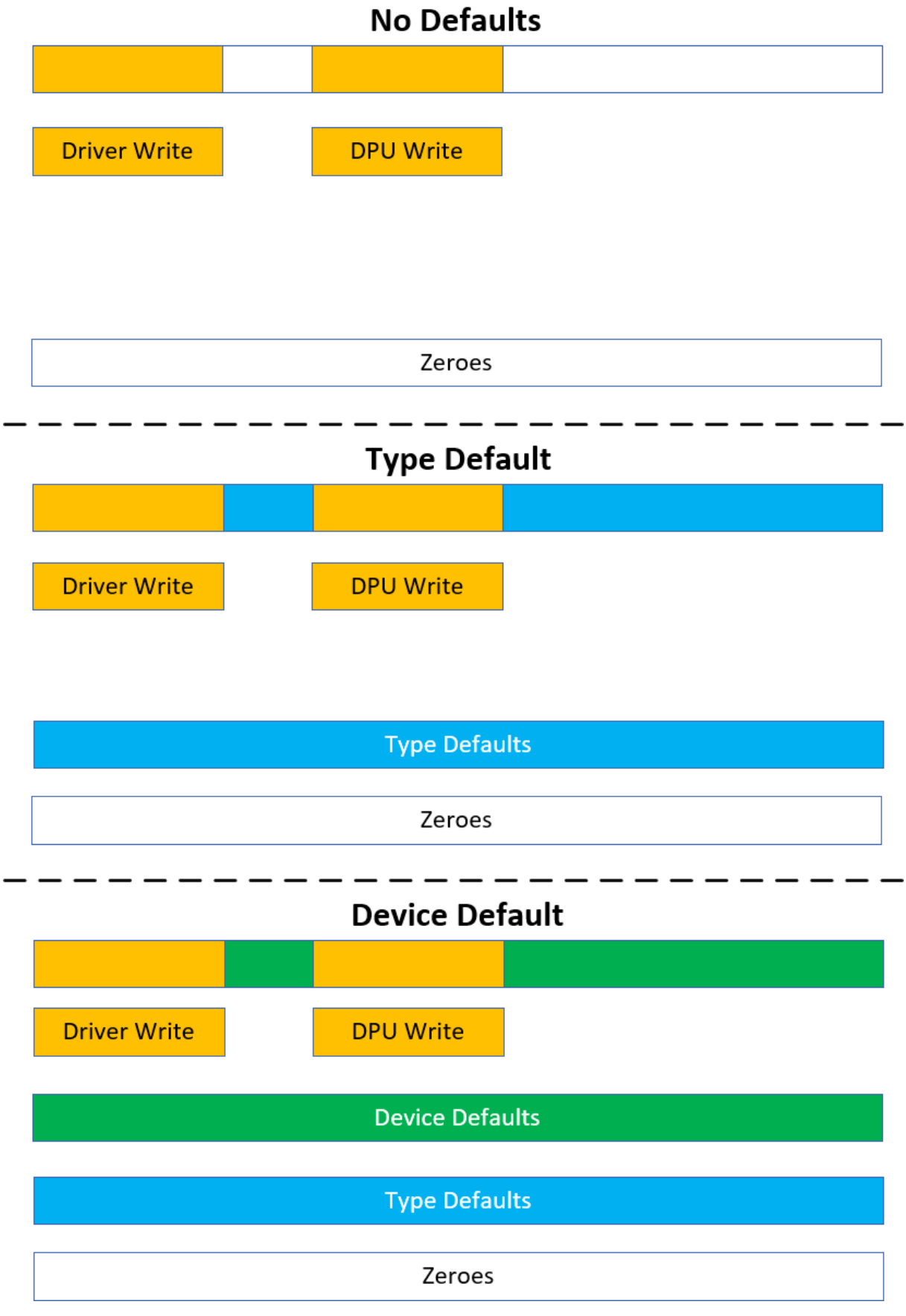
Default Values

The DPU can set default values to the stateful region. Default values come in 2 layers:

- Type default values – these values are set for all devices that have the same type. This can be set only if no device currently exists.
- Device default values – these values are set for a specific device and take effect on the next function-level reset (FLR) cycle or the next hotplug of the device

A read of the stateful region follows the following hierarchy:

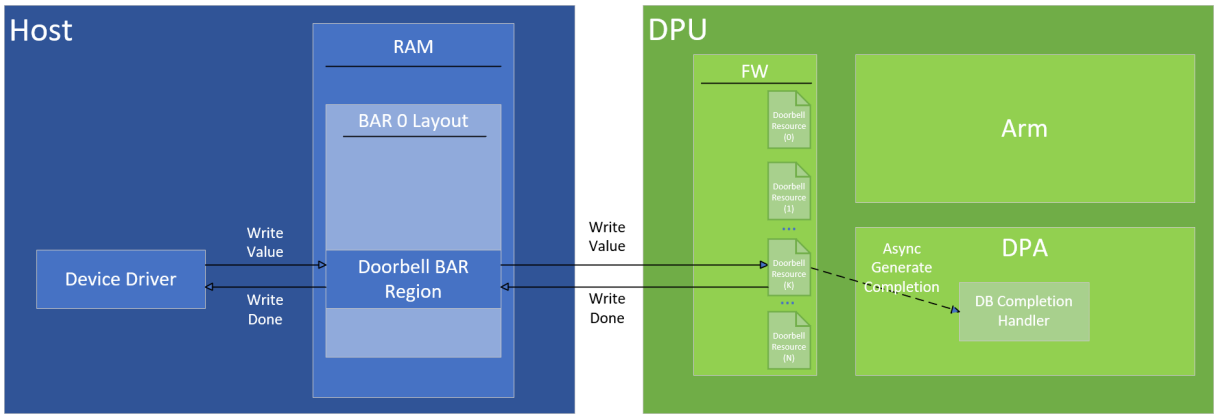
1. Return the latest value as written by the host or driver (whichever was done last).
2. Return the device default values.
3. Return the type default values.
4. Return 0.


No Defaults
Driver Write
DPU Write
Zeroes
Type Default
Driver Write
DPU Write
Type Defaults
Zeroes
Device Default
Driver Write
DPU Write
Device Defaults
Type Defaults
Zeroes

Generic Data Path (Doorbell BAR Region)

Doorbell (DB) regions can be used to implement a consumer-producer queue between the driver and the DPU, such that a write from the driver would trigger an event on the DPU through DPA, allowing it to fetch the written value. This can be useful for communication between the driver and the device, during the data path allowing IO processing.

While DBs are not part of the PCIe specification, it is a widely used mechanism by vendors (e.g., RDMA QP, NVMe SQ, virtio VQ, etc).



The same DB region can be used to manage multiple DBs, such that each DB can be used to implement a queue.

The DPU software can utilize DB resources individually:

- Each DB resource has a unique zero-based index referred to as DB ID
- DB resource can be managed (create/destroy/modify/query) individually
- Each DB resource has a separate notification mechanism. That is, the notification on DPU is triggered for each DB separately.

Driver Write

The DB usually consists of a numeric value (e.g., **`uint32_t`**) representing the consumer/producer index of the queue.

When the driver writes to the DB region, the related DB resource gets updated with the written value, and a notification is sent to the DPU.

When driver writes to the DB BAR region it must adhere to the following:

- The size of the write must match the size of the DB value (e.g., **`uint32_t`**)
- The offset within the region must be aligned to the DB stride size or the DB size

The flow would look like the following:

- Driver performs a write of the DB value at some offset within the DB BAR region
- DPU calculates the DB ID that the write is intended for. Depending on the region type:
  - DB by offset – DPU calculates the DB ID based on the write offset relative to the DB BAR region
  - DB by data – DPU parses the written DB value and extracts the DB ID from it

- DPU updates the DB resource with the matching DB ID to the value written by the driver
- DPU sends a notification to the DPA application, informing it that the value of DB with DB ID has been updated by the driver

Driver Read

The driver should not attempt to read from the DB region. Doing so results in anomalous behavior.

BlueField Write

BlueField can update the value of each DB resource individually using **doca_devemu_pci_db_modify_value**. This produces similar side effects as though the driver updated the value using a write to the DB region.

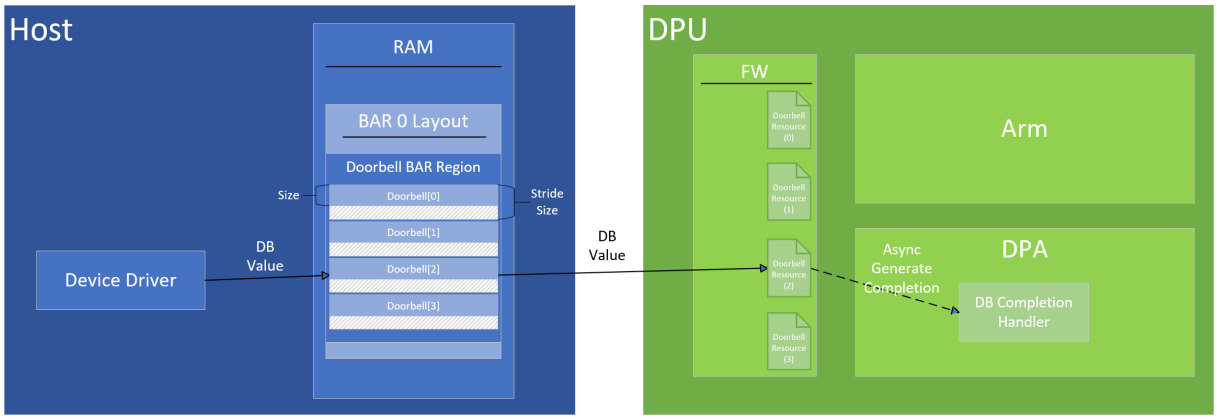
BlueField Read

BlueField can read the value of each DB resource individually using one of the following methods:

- Read the value from the BlueField Arm using **doca_devemu_pci_db_query_value**
- Read the value from the DPA using **doca_dpa_dev_devemu_pci_db_get_value**

The first option is a time-consuming operation and is only recommended for the control path. In the data path, it is recommended to use the second option only.

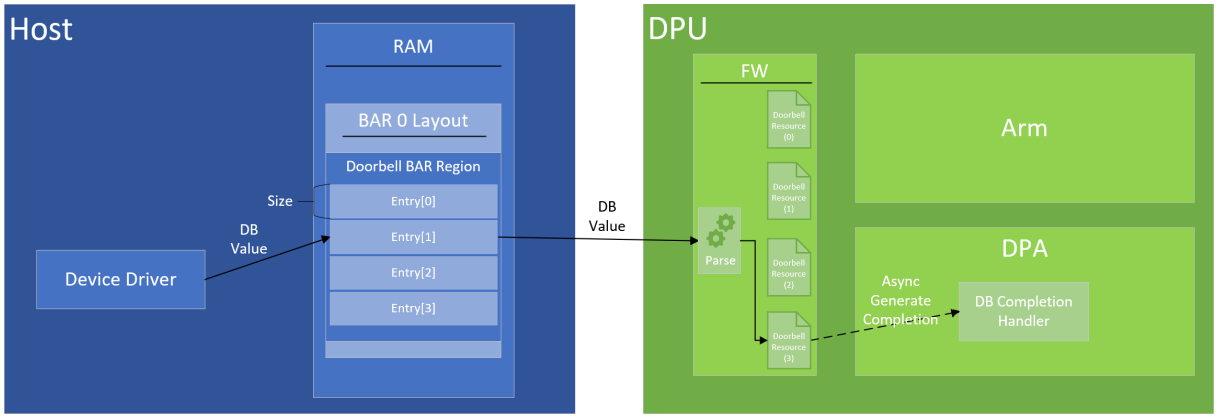
DB by Offset



The API **doca_devemu_pci_type_set_bar_db_region_by_offset_conf** can be used to set up DB by offset region. When the driver writes a DB value using this region, the DPU receives a notification for the relevant DB resource, based on the write offset, such that the DB ID is calculated as follows: **db_id** = **write_offset** / **db_stride_size**.

> The area that is part of the stride but not part of the doorbell should not be used for any read/write operation, doing so will result in anomalous behavior.

DB by Data



The API **doca_devemu_pci_type_set_bar_db_region_by_data_conf** can be used to set up DB by data region. When the driver writes a DB value using this region, the DPU receives a notification for the relevant DB resource based on the written DB value, such that there is no relation between the write offset and the DB triggered. This DB region assumes that the DB ID is embedded within the DB value written by the driver. When setting up this region, the user must specify where the Most Significant Byte (MSB) and Least Significant Byte (LSB) of the DB ID are embedded in the DB value.

The DPU follows these steps to extract the DB ID from the DB value:

- Driver writes the DB value
- BlueField extracts the bytes between MSB and LSB
- DPU compares MSB index with LSB index
  - If MSB index greater than LSB index: The extracted value is interpreted as Little Endian
  - If LSB index greater than MSB index: The extracted value is interpreted as Big Endian

Example:

DB size is 4 bytes, LSB is 1, and MSB is 3.

- Driver writes value **0xCCDDEEFF** to DB region at index 0 in Little Endian
  - The value is written to memory as follows: **[0]=FF [1]=EE [2]=DD [3]=CC**
- The relevant bytes, are the following: **[1]=EE [2]=DD [3]=CC**
- Since MSB (3) is greater than LSB (1), the value is interpreted as Little Endian: **db_id = 0xCCDDEE**

MSI-X Capability (MSI-X BAR Region)

Message signaled interrupts extended (MSI-X) is commonly used by PCIe devices to send interrupts over the PCIe bus to the host driver. DOCA APIs allow users to expose the MSI-X capability as per the PCIe specification, and to later use it to send interrupts to the host driver.

To configure it, users must provide the following:

- The number of MSI-X vectors which can be done using **doca_devemu_pci_type_set_num_msix**
- Define an MSI-X table

- Define an MSI-X PBA

MSI-X Table BAR Region

As per the PCIe specification, to expose the MSI-X capability, the device must designate a memory region within its BAR as an MSI-X table region. In DOCA, this can be done using `doca_devemu_pci_type_set_bar_msix_table_region_conf`.

MSI-X PBA BAR Region

As per the PCIe specification, to expose the MSI-X capability, the device must designate a memory region within its BAR as an MSI-X pending bit array (PBA) region. In DOCA, this can be done using `doca_devemu_pci_type_set_bar_msix_pba_region_conf`.

Raising MSI-X From DPU

It is possible to raise an MSI-X for each vector individually. This can be done using `doca_devemu_pci_msix_raise` or `doca_dpa_dev_devemu_pci_msix_raise` depending on the API used for the creation of the MSI-X (see Raising MSI-X).

DMA Memory

Some operations require accessing I/O memory that is set up by the host driver. DOCA's device emulation APIs allow users to access this memory using the DOCA mmap (see DOCA Core Memory Subsystem).

After starting the PCIe device, you can acquire an mmap that references the host memory by using `doca_devemu_pci_ep_mmap_create`. Once this mmap is created, it must be configured with the following:

- Access permissions
- The host memory range it will reference
- The DOCA devices that are permitted to access this memory

This mmap can then be used to create `doca_buf` buffers that reference the memory on the host.

> **I/O Addresses**
>
> The addresses for these buffers are not locally accessible and cannot be dereferenced by the local CPU. They are I/O addresses as defined by the host driver.

These buffers, created from the mmap, can then be used with other DOCA libraries that accept a `doca_buf` as input. This includes:

- DOCA DMA
- DOCA RDMA
- DOCA Ethernet
- DOCA AES-GCM

Function Level Reset

FLR can be handled as described in DOCA DevEmu PCI FLR. Additionally, users must ensure that the following resources are destroyed before stopping the PCIe device:

- Doorbells created using `doca_devemu_pci_ep_create_db_on_dpa`

- MSI-X vectors created using `doca_devemu_pci_ep_create_msix` or `doca_devemu_pci_ep_create_msix_on_dpa`
- Memory maps created using `doca_devemu_pci_ep_mmap_create`

Limitations

Based on explanation in "Driver Write", user can assume that DOCA DevEmu PCI Generic supports creating emulated PCI devices with the limitation that when a driver writes to a register, the value is immediately available for subsequent reads from the same register. However, this immediate availability does not ensure that any required internal actions triggered by the write have been completed. It is recommended to rely on specific different register values to confirm completion of the write action. For instance, when implementing a write-to-clear operation, e.g. writing 1 to register A to clear register B, it is advisable to poll register B until it indicates the desired state. This approach ensures that the write action has been successfully executed. If a device specification requires certain actions to be completed before exposing written values for subsequent reads, such a device cannot be emulated using the DOCA DevEmu PCI generic framework.

5.4.20.3.7.7 Device Support

DOCA PCI Device emulation requires a device to operate. For information on picking a device, see DOCA DevEmu PCI Device Support.

Some devices can allow different capabilities as follows:

- The maximum number of emulated devices
- The maximum number of different PCIe types
- The maximum number of BARs
- The maximum BAR size
- The maximum number of doorbells
- The maximum number of MSI-X vectors
- Whether adding the device to a DOCA mmap associated with a DOCA Devemu PCI device is supported
- For each BAR region type, there are capabilities for:
  - Whether the region is supported
  - The maximum number of regions with this type
  - The start address alignment of the region
  - The size alignment of the region
  - The min/max size of the region

As the list of capabilities can be long, it is recommended to use the DOCA Capabilities Print Tool to get an overview of all the available capabilities.

Run the tool as root user as follows:

```
$ sudo /opt/mellanox/doca/tools/doca_caps -p <pci-address> -b devemu_pci
Example output:
PCI: 0000:03:00.0
    devemu_pci
        max_hotplug_devices                         15
        max_pci_types                               2
        mmap_add_dev_supported                      supported
        type_log_min_bar_size                       12
        type_log_max_bar_size                       30
        type_max_num_msix                           11
        type_max_num_db                             64
```

```
type_log_min_db_size                        1
type_log_max_db_size                        2
type_log_min_db_stride_size                 2
type_log_max_db_stride_size                 12
type_max_bars                               2
bar_max_bar_regions                         12
type_max_bar_regions                        12
bar_db_region_identify_by_offset            supported
bar_db_region_identify_by_data              supported
bar_db_region_block_size                    4096
bar_db_region_max_num_region_blocks         16
type_max_bar_db_regions                     2
bar_max_bar_db_regions                      2
bar_db_region_start_addr_alignment          4096
bar_stateful_region_block_size              64
bar_stateful_region_max_num_region_blocks   4
type_max_bar_stateful_regions               1
bar_max_bar_stateful_regions                1
bar_stateful_region_start_addr_alignment    64
bar_msix_table_region_block_size            4096
bar_msix_table_region_max_num_region_blocks 1
type_max_bar_msix_table_regions             1
bar_max_bar_msix_table_regions              1
bar_msix_table_region_start_addr_alignment  4096
bar_msix_pba_region_block_size              4096
bar_msix_pba_region_max_num_region_blocks   1
type_max_bar_msix_pba_regions               1
bar_max_bar_msix_pba_regions                1
bar_msix_pba_region_start_addr_alignment    4096
bar_is_32_bit_supported                     unsupported
bar_is_1_mb_supported                       unsupported
bar_is_64_bit_supported                     supported
pci_type_hotplug                            supported
pci_type_mgmt                               supported
```

5.4.20.3.7.8 PCI Type

Configurations

This section describes the configurations of the DOCA DevEmu PCI Type object, that can be provided before start.

To find if a configuration is supported or what its min/max value is, refer to Device Support.

Mandatory Configurations

The following are mandatory configurations and must be provided before starting the PCI type:

- A DOCA device that is an emulation manager or hotplug manager. See Device Support.

Optional Configurations

The following configurations are optional:

- The PCIe device ID
- The PCIe vendor ID
- The PCIe subsystem ID
- The PCIe subsystem vendor ID
- The PCIe revision ID
- The PCIe class code
- The number of MSI-X vectors for MSI-X capability
- One or more memory mapped BARs
- One or more I/O mapped BARs
- One or more DB region
- An MSI-X table and PBA regions
- One or more stateful regions

If these configurations are not set, then a default value is used.

5.4.20.3.7.9 PCI Device

Configuration Phase

This section describes additional configuration options, on top of the ones already described in DOCA DevEmu PCI Device Configuration Phase.

Configurations

The context can be configured to match the application's use case.

To find if a configuration is supported or what its min/max value is, refer to Device Support.

Optional Configurations

The following configurations are optional:

- Setting the stateful regions' default values – If not set, then the type default values are used. See stateful region default values for more.

Execution Phase

This section describes additional events, on top of the ones already described in DOCA DevEmu PCI Device Events.

Events

DOCA DevEmu PCI Device exposes asynchronous events to notify about changes that happen suddenly according to the DOCA Core architecture.

Common events are described in DOCA Core Event.

BAR Stateful Region Driver Write

The stateful region driver write event allows you to receive notifications whenever the host driver writes to the stateful BAR region. See section "Driver Write" for more information.

Configuration

Description	API to Set the Configuration	API to Query Support
Register to the event	`doca_devemu_pci_dev_event_bar_stateful_region_driver_write_register`	`doca_devemu_pci_cap_type_get_max_bar_stateful_regions`

If there are multiple stateful regions for the same device, then registration is done separately for each region. The details provided on registration (i.e., `bar_id` and start address) must match a region previously configured for PCIe type.

Trigger Condition

The event is triggered anytime the host driver writes to the stateful region. See section "Driver Write" for more information.

Output

Common output as described in DOCA Core Event.

Additionally, the event callback receives an event object of type **struct doca_devemu_pci_dev_event_bar_stateful_region_driver_write** which can be used to retrieve:

- The DOCA DevEmu PCI Device representing the emulated device that triggered the event – **doca_devemu_pci_dev_event_bar_stateful_region_driver_write_get_pci_dev**
- The ID of the BAR containing the stateful region – **doca_devemu_pci_dev_event_bar_stateful_region_driver_write_get_bar_id**
- The start address of the stateful region – **doca_devemu_pci_dev_event_bar_stateful_region_driver_write_get_bar_region_start_addr**

Event Handling

Once the event is triggered, it means that the host driver has written to someplace in the region.

The user must perform either of the following:

- Query the new values of the stateful region – **doca_devemu_pci_dev_query_bar_stateful_region_values**
- Modify the values of the stateful region – **doca_devemu_pci_dev_modify_bar_stateful_region_values**

It is possible also to do both. However, it is important that the memory areas that the host wrote to are either queried or overwritten with a modify operation.

Failure to do so results in a recurring event. For example, if the host wrote to the first half of the region, BlueField Arm only queries the second half of the region after receiving the event. Then, if the user did not handle the event, the library retriggers the event.

5.4.20.3.7.10 PCI Device DB

After the PCIe device has been created, the underlying DOCA DevEmu PCI Endpoint can be used to create DB objects. Each DB object represents a DB resource identified by a DB ID. See Generic Data Path (Doorbell BAR Region).

When creating the DB, the DB ID must be provided, this can hold different meaning for DB by offset and DB by data. The DB object can then be used to get a notification to the DPA once a driver write occurs, and to fetch the latest value using the DPA.

To enable creation of a Doorbell on the DPA, the DOCA DevEmu PCI Endpoint data path must be set on the DPA.

Configuration

Creating and Configuring DB

The flow for creating and configuring a DB should be as follows:

1. Create the DB object:

```
arm> doca_devemu_pci_ep_create_db_on_dpa
```

2. (Optional) Query the DB value:

```
arm> doca_devemu_pci_db_query_value
```

3. (Optional) Modify the DB value:

```
arm> doca_devemu_pci_db_modify_value
```

4. Get the DB DPA handle for referencing the DB from the DPA:

```
arm> doca_devemu_pci_db_get_dpa_handle
```

5. Bind the DB to the DB completion context using the handle from the previous step:

```
dpa> doca_dpa_dev_devemu_pci_db_completion_bind_db
```

It is important to perform this step before the next one. Otherwise, the DB completion context will start receiving completions for an unbound DB.

6. Start the DB to start receiving completions on DPA:

```
arm> doca_devemu_pci_db_start
```

Once DB is started, a completion is immediately generated on the DPA.

Destroying DB

The flow for destroying a DB would look as follows:

1. Stop the DB to stop receiving completions:

```
arm> doca_devemu_pci_db_stop
```

This step ensures that no additional completions will arrive for this DB

2. Acknowledge all completions related to this DB:

```
dpa> doca_dpa_dev_devemu_pci_db_completion_ack
```

This step ensures that existing completions have been processed.

3. Unbind the DB from the DB completion context:

```
dpa> doca_dpa_dev_devemu_pci_db_completion_unbind_db
```

Make sure not to perform this step more than once.

4. Destroy the DB object:

```
arm> doca_devemu_pci_db_destroy
```

Fetching DBs on DPA

To fetch DBs on DPA, a DB completion context can be used. The DB completion context serves the following purposes:

- Notifying a DPA thread that a DB value has been updated (wakes up thread)
- Providing information about which DB has been updated

The following flow shows how to use the same DB completion context to get notified whenever any of the DBs are updated, and to find which DBs were actually updated, and finally to get the DBs' values:

1. Get DB completion element:

```
doca_dpa_dev_devemu_pci_get_db_completion
```

2. Get DB from completion:

```
doca_dpa_dev_devemu_pci_db_completion_element_get_db_properties
```

3. Store the DB (e.g., in an array).
4. Repeat steps 1-3 until there are no more completions.
5. Acknowledge the number of received completions:

```
doca_dpa_dev_devemu_pci_db_completion_ack
```

6. Request notification on DPA for the next completion:

```
doca_dpa_dev_devemu_pci_db_completion_request_notification
```

7. Go over the DBs stored in step 3 and for each DB:
    a. Request a notification for the next time the host driver writes to this DB:

```
doca_dpa_dev_devemu_pci_db_request_notification
```

    b. Get the most recent value of the DB:

```
doca_dpa_dev_devemu_pci_db_get_value
```

Query/Modify DB from Arm

It is possible to query the DB value of a particular DB using **doca_devemu_pci_db_query_value** on the Arm. Similarly, it is possible to modify the DB value using **doca_devemu_pci_db_modify_value**. When modifying the DB value, the side effects of such modification is the same as if the host driver updated the DB value.

Querying and modifying operations from the Arm are time consuming and should be used in the control path only. Fetching DBs on DPA is the recommended approach for retrieval of DB values in the data path.

5.4.20.3.7.11 PCIe Device MSI-X Vector

After the PCIe device is created, the underlying DOCA DevEmu PCI Endpoint can be used to create MSI-X objects. Each MSI-X object represents an MSI-X vector identified by the vector index. The MSI-X object can be used to send a notification to the host driver from either the DPU or the DPA.

Configuration

There are two ways to create the MSI-X object:

- On the DPU using `doca_devemu_pci_ep_create_msix`.
- On the DPA using `doca_devemu_pci_ep_create_msix_on_dpa`.

An MSI-X vector index must be provided during creation. This is a value in the range `[0, num_msix)`, where `num_msix` is the value previously set using `doca_devemu_pci_type_set_num_msix`.

When created on the DPA, `doca_devemu_pci_msix_get_dpa_handle` can be used to get a DPA handle for use within the DPA.

> ⚠ To enable the creation of an MSI-X on the DPA, the DOCA DevEmu PCI Endpoint data path must be set on the DPA.

Raising MSI-X

You can use the MSI-X object to raise an MSI-X vector via the following APIs:

- Use `doca_devemu_pci_msix_raise` if the MSI-X object was created using `doca_devemu_pci_ep_create_msix`.
- Use `doca_dpa_dev_devemu_pci_msix_raise` if the MSI-X object was created using `doca_devemu_pci_ep_create_msix_on_dpa`.

5.4.20.3.7.12 DOCA DevEmu Generic Samples

This section describes DOCA DevEmu Generic samples.

The samples illustrate how to use the DOCA DevEmu Generic API to do the following:

- List details about emulated devices with same generic type
- Create and hot-plug/hot-unplug an emulated device with a generic type
- Handle Host driver write using stateful region
- Handle Host driver write using DB region
- Raise MSI-X to the Host driver
- Perform DMA operation to copy memory buffer between the Host driver and the DPU Arm

Structure

All the samples utilize the same generic PCI type. The configurations of the type reside in `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h`

The structure for some samples is as follows:

- **/opt/mellanox/doca/samples/doca_devemu/<sample directory>**
  a. **dpu**
    i. **host**
    ii. **device**
  b. **host**

Samples following this structure will have two binaries: **dpu** (1) and **host** (2), the former should be run on the BlueField and represents the controller of the emulated device, while the latter should be run on the host and represents the host driver.

For simplicity, the host (2) side is based on the VFIO driver, allowing development of a driver in user-space.

Within the **dpu** (a) directory, there is a **host** (a) and **device** (b) directories. **host** in this case refers to the BlueField Arm processor, while **device** refers to the DPA processor. Both directories are compiled into a single binary.

Running the Samples

1. Refer to the following documents:
   - DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - DOCA Troubleshooting for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

```
cd /opt/mellanox/doca/samples/doca_devemu/<sample_name>[/dpu or /host]
meson /tmp/build
ninja -C /tmp/build
```

> The binary **doca_<sample_name>[_dpu or _host]** is created under **/tmp/build/**.

3. Sample (e.g., **doca_devemu_pci_device_db**) usage:
   a. BlueField side (**doca_devemu_pci_device_db_dpu**):

```
Usage: doca_devemu_pci_device_db_dpu [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse all command flags from an input json file

Program Flags:
  -p, --pci-addr                    The DOCA device PCI address. Format: XXXX:XX:XX.X or XX:XX.X
  -u, --vuid                        DOCA Devemu emulated device VUID. Sample will use this device
to handle Doorbells from Host
  -r, --region-index                The index of the DB region as defined in
 devemu_pci_type_config.h. Integer
  -i, --db-id                       The DB ID of the DB. Sample will listen on DBs related to this
DB ID. Integer
```

   b. Host side (**doca_devemu_pci_device_db_host**):

```
Usage: doca_devemu_pci_device_db_host [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse all command flags from an input json file

Program Flags:
  -p, --pci-addr                    PCI address of the emulated device. Format: XXXX:XX:XX.X
  -g, --vfio-group                  VFIO group ID of the device. Integer
  -r, --region-index                The index of the DB region as defined in
 devemu_pci_type_config.h. Integer
  -d, --db-index                    The index of the Doorbell to write to. The sample will write at
byte offset (db-index * db-stride)
  -w, --db-value                    A 4B value to write to the Doorbell. Will be written in Big
Endian
```

4. For additional information per sample, use the **-h** option:

```
/tmp/build/<sample_name> -h
```

Additional sample setup:

- The BlueField samples require the emulated device to be already hot-plugged:
  - Such samples expect the VUID of the hot-plugged device (**-u, --vuid**)
  - The list sample can be used to find if any hot-plugged devices exist and what their VUID is
  - The hot-plug sample can be used to hot plug a device if no such device already exists
- The host samples require the emulated device to be already hot-plugged and that the device is bound to the VFIO driver:
  - The samples expect 2 parameters **-p** (**--pci-addr**) and **-g** (**--vfio-group**) of the emulated device as seen by the host
  - The PCI Device List sample can be used from the BlueField to find the PCIe address of the emulated device on the host
  - Once the PCIe address is found, the host can use the script **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_vfio_bind.py** to bind the VFIO driver

    ```
    $ sudo python3 /opt/mellanox/doca/samples/doca_devemu/devemu_pci_vfio_bind.py <pcie-address-of-emulated-dev>
    ```

    - The script is a python3 script which expects the PCIe address of the emulated device as a positional argument (e.g., **0000:3e:00.0**)
    - The script outputs the VFIO group ID
    - The script must be used only once after the device is hot plugged towards the host for the first time
- The hot-unplug sample requires that the device be unbound from the VFIO driver:
  - Use the script located at **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_vfio_bind.py** from the host to unbind the VFIO driver as follows:

    ```
    $ sudo python3 /opt/mellanox/doca/samples/doca_devemu/devemu_pci_vfio_bind.py <pcie-address-of-emulated-dev> --unbind
    ```

    - This python3 script expects the PCIe address of the emulated device as a positional argument (e.g., **0000:3e:00.0**) along with the **--unbind** argument

Samples

These samples are also available on GitHub.

PCI Device List

This sample illustrates how to list all emulated devices that have the generic type configured in `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h` .

The sample logic includes:

1. Initializing the generic PCIe type based on `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h` .
2. Creating a list of all emulated devices belonging to this type.
3. Iterating over the emulated devices.
4. Dumping their VUID.
5. Dumping their PCIe address as seen by the host.
6. Releasing the resources.

References:

- `/opt/mellanox/doca/samples/doca_devemu/`
  - `devemu_pci_device_list/`
    - `devemu_pci_device_list_sample.c`
    - `devemu_pci_device_list_main.c`
    - `meson.build`
  - `devemu_pci_common.h` ; `devemu_pci_common.c`
  - `devemu_pci_type_config.h`

PCI Device Hot-Plug

This sample illustrates how to create and hot-plug/hot-unplug an emulated device that has the generic type configured in `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h` .

The sample logic includes:

1. Initializing the generic PCIe type based on `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h` .
2. Acquiring the emulated device representor:
   - If the user did not provide VUID as input, then creating and using a new emulated device.
   - If the user provided VUID as an input, then searching for an existing emulated device with a matching VUID and using it.
3. Creating a PCIe device context to manage the emulated device and connecting it to a progress engine (PE).
4. Registering to the PCIe device's hot-plug state change event.
5. Initializing hot-plug/hot-unplug of the device:
   a. If the user did not provide VUID as input, then initializing hot-plug flow of the device.

b. If the user provided VUID as input, then initializing hot-unplug flow of the device.

6. Using the PE to poll for hot-plug state change event.
7. Waiting until hot-plug state transitions to expected state (power on or power off).
8. Cleaning up resources.
    - If hot-unplug was requested, then the emulated device is destroyed as well.
    - Otherwise, the emulated device persists.

References:

- **/opt/mellanox/doca/samples/doca_devemu/**
    - **devemu_pci_device_hotplug/**
        - **devemu_pci_device_hotplug_sample.c**
        - **devemu_pci_device_hotplug_main.c**
        - **meson.build**
    - **devemu_pci_common.h** ; **devemu_pci_common.c**
    - **devemu_pci_type_config.h**

PCI Device Stateful Region

This sample illustrates how the host driver can write to a stateful region, and how the BlueField Arm can handle the write operation.

This sample consists of a host sample and BlueField sample. It is necessary to follow the additional sample setup detailed previously.

The BlueField sample logic includes:

1. Initializing the generic PCIe type based on **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h** .
2. Acquiring the emulated device representor that matches the provided VUID.
3. Creating a PCIe device context to manage the emulated device and connecting it to a progress engine (PE).
4. For each stateful region configured in **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h** , registering to the PCIe device's stateful region write event.
5. Using the PE to poll for driver write to any of the stateful regions.
    - Every time the host driver writes to the stateful region, the handler is invoked and performs the following:
        i. Queries the values of the stateful region that the host wrote to.
        ii. Logs the values of the stateful region.
    - The sample polls indefinitely until the user presses [Ctrl+c] to close the sample.
6. Cleaning up resources.

The host sample logic includes:

1. Initializing the VFIO device with a matching PCIe address and VFIO group.
2. Mapping the stateful memory region from the BAR to the process address space.
3. Writing the values provided as input to the beginning of the stateful region.

References:

- **/opt/mellanox/doca/samples/doca_devemu/**

- `devemu_pci_device_stateful_region/dpu/`
  - `devemu_pci_device_stateful_region_dpu_sample.c`
  - `devemu_pci_device_stateful_region_dpu_main.c`
  - `meson.build`
- `devemu_pci_device_stateful_region/host/`
  - `devemu_pci_device_stateful_region_host_sample.c`
  - `devemu_pci_device_stateful_region_host_main.c`
  - `meson.build`
- `devemu_pci_common.h` ; `devemu_pci_common.c`
- `devemu_pci_host_common.h` ; `devemu_pci_host_common.c`
- `devemu_pci_type_config.h`

PCI Device DB

This sample illustrates how the host driver can ring the doorbell and how the BlueField can retrieve the doorbell value. The sample also demonstrates how to properly handle FLR.

This sample consists of a host sample and BlueField sample. It is necessary to follow the additional sample setup detailed previously.

BlueField Sample Logic

The BlueField portion of this sample executes logic on both the Host (BlueField Arm) and the Device (BlueField DPA).

Host (BlueField Arm) Logic

1. Initialize the generic PCIe type based on the `/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h` configuration.
2. Initialize DPA resources: Create a DPA instance and associate it with the DPA application, create a DPA thread and associate it with the DPA DB handler, and create a DB completion context and associate it with the DPA thread.
3. Acquire the emulated device representor that matches the provided VUID.
4. Create a PCIe device context to manage the emulated device and connect it to a progress engine (PE).
5. Register to the context state changes event.
6. Register to the PCIe device FLR event.
7. Use the PE to poll for any of the following state changes or events:
   - Transition to running: Every time the PCIe device context state transitions to running, the handler creates a DB object and makes an RPC to the DPA to initialize the DB object.
   - Transition to stopping: Every time the PCIe device context state transitions to stopping, the handler makes an RPC to the DPA to un-initialize the DB object, then explicitly destroys the DB object.
   - FLR triggered: Every time the host driver initializes or destroys the VFIO device, an FLR event is triggered. The FLR handler immediately destroys the DB object, stops the PCIe device context, and then starts the PCIe device context again.

- Continuous polling: The sample polls indefinitely until the user presses **[Ctrl+c]** to close the application.

  > ⚠ During this polling time, the DPA may start receiving DBs from the host.

8. Clean up resources.

Device (BlueField DPA) Logic

1. Set the global context to point to the DB completion context DPA handle, then bind the DB to the doorbell completion context.
2. Unbind the DB from the doorbell completion context.
3. When triggered, the handler gets the DB completion element from the completion context, gets the DB handle from the DB completion element, acknowledges the DB completion element, requests notification from the DB completion context, requests notification from the DB, and finally gets the DB value from the DB.

Host Sample Logic

The host portion of the sample is responsible for triggering the doorbell:

1. Initialize the VFIO device using its matching PCIe address and VFIO group.
2. Map the DB memory region from the BAR to the process address space.
3. Write the value provided as input to the DB region at the specified offset.

References

BlueField files:

- **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_device_db/dpu/host/devemu_pci_device_db_dpu_sample.c**
- **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_device_db/dpu/device/devemu_pci_device_db_dpu_kernels_dev.c**
- **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_device_db/dpu/device/devemu_pci_device_db_attributes.yaml**
- **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_device_db/dpu/devemu_pci_device_db_dpu_main.c**

Host files:

- **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_device_db/host/devemu_pci_device_db_host_sample.c**
- **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_device_db/host/devemu_pci_device_db_host_main.c**

Shared libraries:

- **devemu_pci_common.h** / **devemu_pci_common.c**
- **devemu_pci_host_common.h** / **devemu_pci_host_common.c**
- **devemu_pci_type_config.h**

PCI Device MSI-X

This sample illustrates how the BlueField can raise an MSI-X vector, either from the DPA or the DPU, sending a signal towards the host, and shows how the host can retrieve this signal.

> ⚠ This sample consists of a host sample and a BlueField sample. It is necessary to follow the additional sample setup detailed previously before proceeding.

BlueField Sample Logic

The BlueField portion of this sample executes logic on both the Host (BlueField Arm) and the Device (BlueField DPA).

Host (BlueField Arm) Logic

1. Initialize the generic PCIe type based on the **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h** configuration.
2. Depending on the selected MSI-X datapath, proceed with one of the following configurations:
    - Option A: For DPA (default)
        i. Initialize DPA resources: Create a DPA instance and associate it with the DPA application, then create a DPA thread and associate it with the DPA MSI-X handler.
        ii. Set the PCI device context datapath on the DPA.
        iii. Acquire the emulated device representor that matches the provided VUID.
        iv. Create a PCIe device context to manage the emulated device and connect it to a PE.
        v. Create an MSI-X vector on the DPA and acquire its DPA handle.
        vi. Send an RPC to the DPA to raise the MSI-X vector.
    - Option B: For DPU
        i. Acquire the emulated device representor that matches the provided VUID.
        ii. Create a PCIe device context to manage the emulated device and connect it to a PE.
        iii. Create an MSI-X vector on the DPU.
        iv. Raise the MSI-X vector directly from the DPU.
3. Clean up resources.

Device (BlueField DPA) Logic

1. Raise the MSI-X RPC by using the MSI-X vector handle.

Host Sample Logic

The host portion of the sample is responsible for receiving the signal:

1. Initialize the VFIO device using the matching PCIe address and VFIO group.
2. Map each MSI-X vector to a different File Descriptor (FD).
3. Read events from the FDs in a continuous loop:
    - Once the DPU raises the MSI-X, the FD matching the MSI-X vector returns an event which is then printed to the screen.

- The sample polls the FDs indefinitely until the user presses **[Ctrl+c]** to close the application.

References

BlueField files:

- **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_device_msix/dpu/host/devemu_pci_device_msix_dpu_sample.c**
- **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_device_msix/dpu/device/devemu_pci_device_msix_dpu_kernels_dev.c**
- **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_device_msix/dpu/device/devemu_pci_device_msix_attributes.yaml**
- **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_device_msix/dpu/devemu_pci_device_msix_dpu_main.c**

Host files:

- **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_device_msix/host/devemu_pci_device_msix_host_sample.c**
- **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_device_msix/host/devemu_pci_device_msix_host_main.c**

Shared libraries:

- **devemu_pci_common.h** / **devemu_pci_common.c**
- **devemu_pci_host_common.h** / **devemu_pci_host_common.c**
- **devemu_pci_type_config.h**

PCI Device DMA

This sample illustrates how the host driver can set up memory for DMA, then the DPU can use that memory to copy a string from the BlueField to the host and from the host to the BlueField.

This sample consists of a host sample and a BlueField sample. It is necessary to follow the additional sample setup detailed previously.

The BlueField sample logic includes:

1. Initializing the generic PCIe type based on **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h** .
2. Acquiring the emulated device representor that matches the provided VUID.
3. Creating a PCIe device context to manage the emulated device and connecting it to a progress engine (PE).
4. Creating a DMA context to use for copying memory across the host and BlueField.
5. Setting up an mmap representing the host driver memory buffer.
6. Setting up an mmap representing a local memory buffer.
7. Use the DMA context to copy memory from host to BlueField.
8. Use the DMA context to copy memory from BlueField to host.
9. Cleaning up resources.

The host sample logic includes:

1. Initializing the VFIO device with the matching PCIe address and VFIO group.
2. Allocating memory buffer.
3. Mapping the memory buffer to I/O memory. The BlueField can now access the memory using the I/O address through DMA.
4. Copying the string provided by user to the memory buffer.
5. Waiting for the BlueField to write to the memory buffer.
6. Un-mapping the memory buffer.
7. Cleaning up resources.

References:

- **/opt/mellanox/doca/samples/doca_devemu/**
  - **devemu_pci_device_dma/dpu/**
    - **devemu_pci_device_dma_dpu_sample.c**
    - **devemu_pci_device_dma_dpu_main.c**
    - **meson.build**
  - **devemu_pci_device_dma/host/**
    - **devemu_pci_device_dma_host_sample.c**
    - **devemu_pci_device_dma_host_main.c**
    - **meson.build**

5.4.20.3.8 DOCA DevEmu PCI TLP

This guide provides instructions on building and developing applications that require emulation of a PCIe device at the PCI Transaction Layer Packet level.

5.4.20.3.8.1 Introduction

DOCA DevEmu PCI TLP is part of the DOCA Device Emulation subsystem. It provides low-level software APIs that allow emulation of a custom PCIe device at the PCI Transaction Layer Packet (TLP) level using the emulation capability of NVIDIA® BlueField®.

PCIe transactions from the host that target the application's emulated devices are forwarded to the application running on the DPU, which processes the TLPs in accordance with the PCI specification, allowing full management of the configuration space of the PCIe device (endpoint or bridge), and construction of a custom PCIe topology consisting of endpoints and switches.

5.4.20.3.8.2 Prerequisites

This library follows the architecture of a DOCA Core Context. It is recommended read the following sections beforehand:

- DOCA Core Execution Model
- DOCA Core Device
- DOCA Core Memory Subsystem

TLP device emulation is part of DOCA PCIe device emulation. It is recommended to read the following guides beforehand:

- DOCA Device Emulation

- DOCA DevEmu PCI
- DOCA DevEmu PCI Generic

5.4.20.3.8.3 Environment

DOCA DevEmu PCI TLP Emulation is supported only on the BlueField target. The BlueField must meet the following requirements:

- DOCA version 3.4.0 or higher
- BlueField-3 firmware 32.49.0288 or higher

> Please refer to the DOCA Compatibility Policy.

Library must be run with root privileges.

Perform the following:

1. Configure the BlueField to work in DPU mode as described in BlueField Modes of Operation.
2. Set the required device configurations by running the following commands on the host or BlueField:

   Enable the PCIe switch emulation capability

   ```
   host/bf> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s PCI_SWITCH_EMULATION_ENABLE=1
   ```

   Enable TLP emulation

   ```
   host/bf> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s TLP_EMULATION_ENABLE=1
   ```

   Set at least 1 NVIDIA emulation switch port for TLP emulation

   ```
   host/bf> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s PCI_SWITCH_EMULATION_NUM_TLP_PORT=1
   ```

   Set at least (PCI_SWITCH_EMULATION_NUM_TLP_PORT + 1) NVIDIA emulation switch ports

   ```
   host/bf> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s PCI_SWITCH_EMULATION_NUM_PORT=2
   ```

   Set at least 1 PF for TLP emulation

   ```
   host/bf> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s TLP_EMULATION_NUM_PF=1
   ```

3. Perform a BlueField system-level reset for the **mlxconfig** settings to take effect.

To support the hot-plug feature please follow the instructions described at DOCA DevEmu PCI.

5.4.20.3.8.4 Architecture

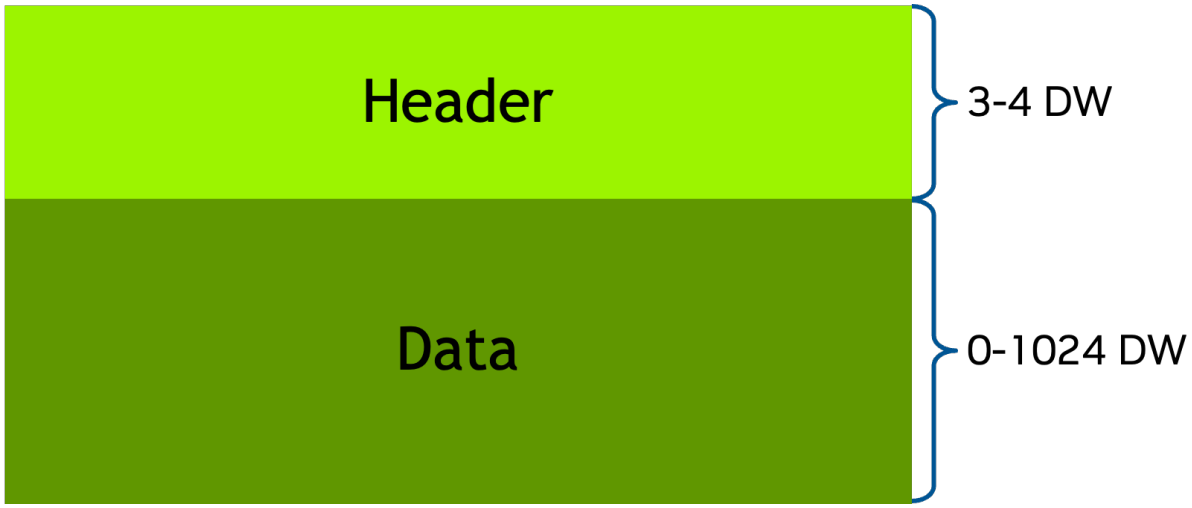
DOCA DevEmu PCI TLP allows the creation of a PCI TLP type which essentially serves the same purpose of a generic PCI type as described at DOCA DevEmu PCI Generic, only it places the responsibility of processing and responding to PCIe traffic (Transaction layer packets) on the user

application, and some other exceptions, e.g., a PCI TLP type has its own capabilities, and APIs only partially overlap (some apply to both, while others are type-specific).

Transaction Layer Packets

The transaction layer packet (TLP) is the fundamental unit of information exchange in a PCIe fabric, carrying requests and responses between the host and the PCIe device.

Each TLP starts with a 3 or 4 DW (Double Word) header and ends with up to 1024 DW of data:



DOCA DevEmu PCI TLP supports 3 types of TLPs:

- Configuration read/write - access a PCIe device configuration space.
- MMIO (or memory) read/write - access memory within the BARs address range.
- Completion - response from a PCIe device back to the requester.

All TLP requests in this scope require a response with a completion TLP except MMIO write TLPs.

> For a detailed breakdown of the Transaction Layer Packet, please refer to the **PCI Express Base Specification**.

DOCA DevEmu PCI TLP provides APIs that allow the application to fully handle and respond to incoming TLP requests from the host.

PCIe Configuration Space

Unlike generic emulation where the configuration space is maintained internally, with TLP emulation the application is responsible for the following:

- Keeping track of the configuration space of each device (Type 0 for an endpoint, Type 1 for a bridge) explicitly (for example, by preserving a dedicated data structure of its fields).
- Handling all configuration read and write TLP requests.
- Constructing the appropriate completion responses.

This enables the application to dynamically create robust, custom PCIe topologies consisting of switches and endpoints.

For a detailed breakdown of the PCIe device Configuration Space and its types, please refer to the PCI Express Base Specification.

Base Address Register (BAR)

DOCA DevEmu PCI TLP utilizes most of the generic emulation BAR and BAR regions functionality, for example, configure I/O and memory mapped BAR layouts, and BAR regions like Doorbell and MSI-X. The only exception is the **Stateful BAR Region**, which is not supported by TLP emulation.

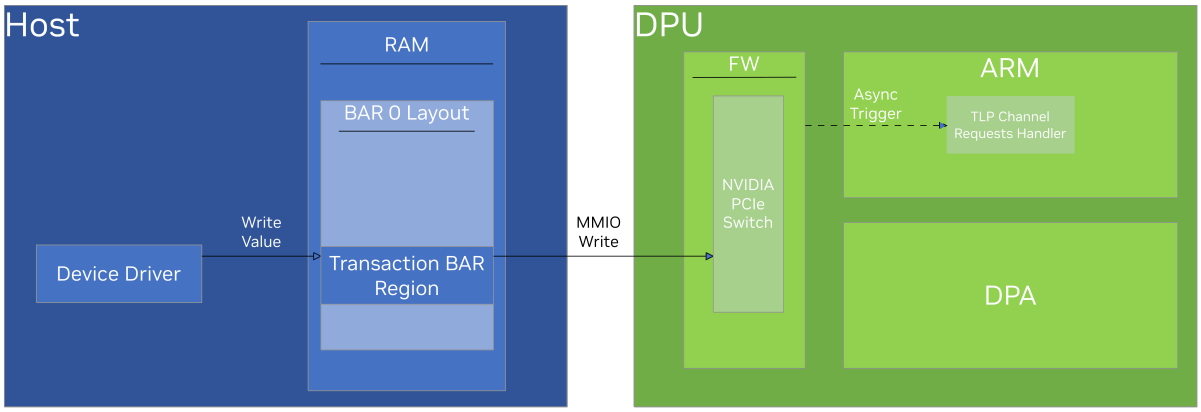
TLP Control Path (Transaction BAR Region)

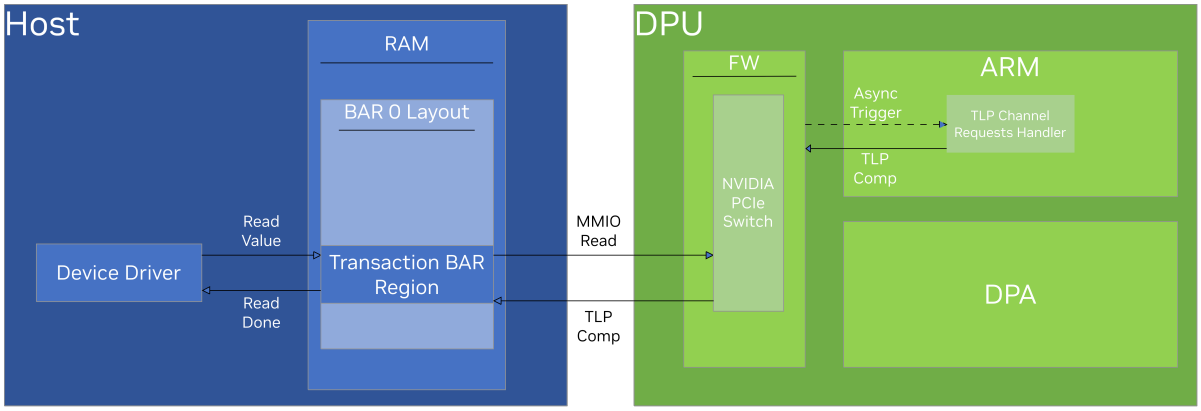
The Transaction region is a general-purpose BAR region for host driver ↔ device communication, typically during the configuration phase (e.g., exposing capabilities, initialization, etc.). Any read or write to an address within this region by the host driver is passed to the application running on the DPU in the form of a memory read or memory write (respectively) TLP request, and therefore it is the application's responsibility to maintain and update the Transaction region state consistently.

The application has full control over the Transaction region and can define any desired behavior for any transaction from the host driver that targets the region's memory range.

Although maintained by the application, to expose a Transaction region the application must configure it explicitly using **doca_devemu_pci_tlp_type_set_bar_transaction_region_conf**

The following diagram depicts how MMIO Write and Read operations targeting the Transaction BAR Region are executed:





TLP Data Path (Doorbell BAR Region)

DOCA DevEmu PCI TLP shares the same Doorbell BAR region logic as DOCA DevEmu PCI Generic (see DOCA DevEmu PCI Generic Doorbell BAR Region).

MSI-X Capability (MSI-X BAR Region)

DOCA DevEmu PCI TLP shares the same MSI-X BAR region logic as DOCA DevEmu PCI Generic (see DOCA DevEmu PCI Generic MSI-X BAR Region).

DMA Memory

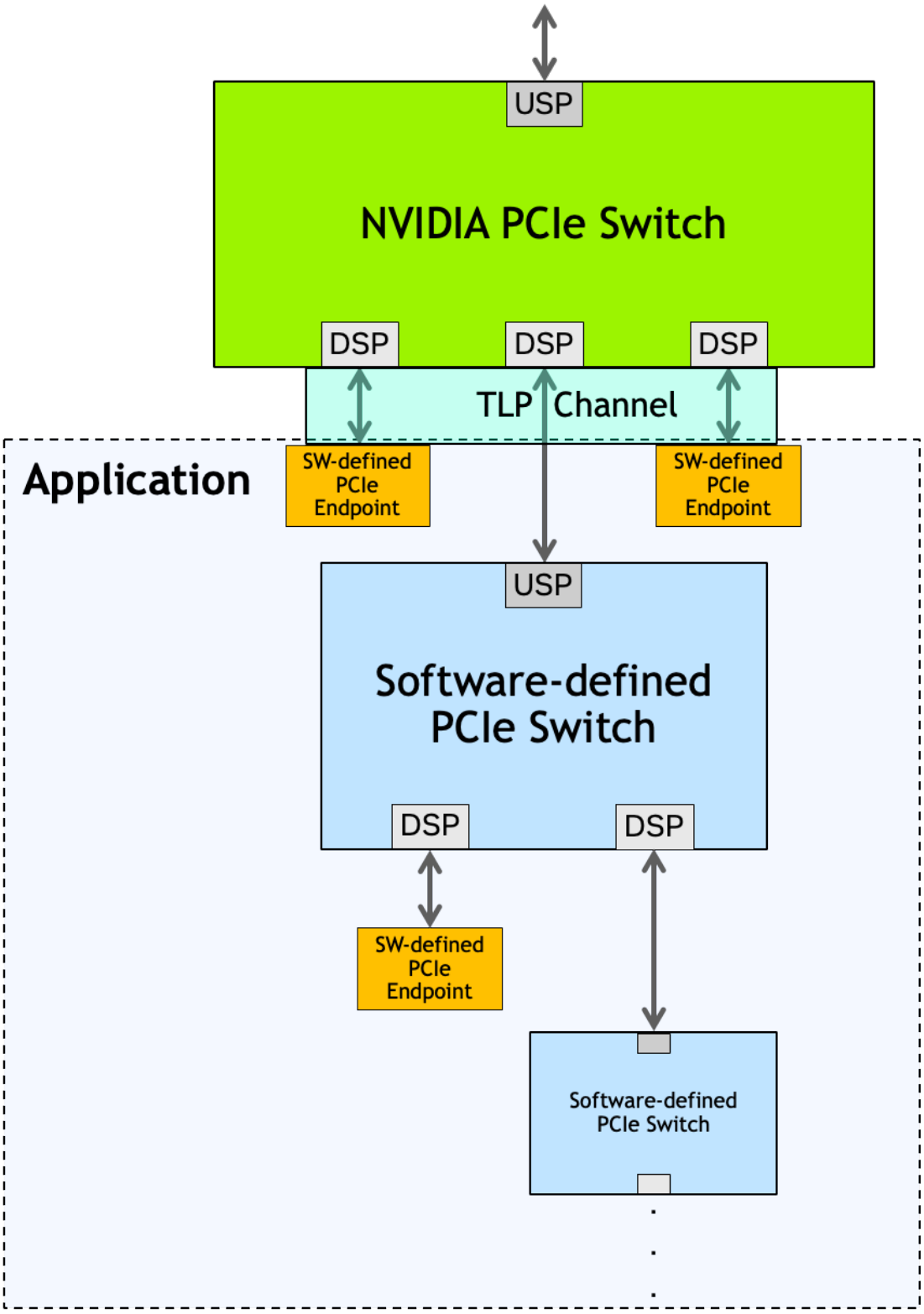
DOCA DevEmu PCI TLP shares the same I/O memory access logic as DOCA DevEmu PCI Generic (see DOCA DevEmu PCI Generic DMA Memory).

TLP Channel

DOCA DevEmu PCI TLP introduces a bidirectional channel for receiving requests and sending responses (completing requests). A single channel can deliver transactions from one or more NVIDIA emulation switch downstream ports, and the application has the flexibility to define the PCIe entity connected to each port (either an endpoint or a switch). The number of downstream ports associated with a given channel can be retrieved using **`doca_devemu_pci_tlp_channel_get_num_dsp`**. See TLP Channel Request Downstream Port Identifier for more details on how an application can identify the downstream port associated with any given transaction.

The application has to follow the PCIe specification to handle incoming TLP requests properly.

The following diagram shows an example of a TLP channel and a software PCIe topology built underneath:



In addition to TLP requests, the TLP channel also supports the following:

- Asynchronous Credit Grant (ACG) requests - allows the application to transmit PCI-compliant MMIO write TLP to the host, e.g., use MSI to notify the host on a device hot-plug. For more information, see Asynchronous Credit Grant requests.
- PCI Event requests - notify the application on state changes that may require specific actions, e.g., PERST# assertion and de-assertion. For more information, see PCI Event requests.

A TLP channel can also enforce a response timeout for non-posted TLP requests (MMIO Read and Configuration Read/Write), all of which require a completion. If the application does not complete such a request within the configured time window, the device autonomously returns error completions to the host for all outstanding non-posted requests, as mandated by the PCIe specification, and transitions the channel to a *fatal state*. A channel in a fatal state can no longer serve traffic, and it must be recreated together with its associated devices. See TLP Channel Non-Posted Response Timeout for the configuration and recovery flow.

SR-IOV

> Before reading this section, please refer to the SR-IOV section of the DOCA DevEmu PCI guide.

With TLP emulation, the responsibility for the SR-IOV capability rests with the application. When exposing an SR-IOV PF over TLP emulation, the application must implement the SR-IOV Extended Capability for each SR-IOV PF, and maintain the configuration space and BARs of the associated VFs – all in accordance with the PCI specification.

Whether SR-IOV is supported for a PCI TLP type can be queried using `doca_devemu_pci_tlp_cap_type_is_dynamic_sriov_supported`.

Using SR-IOV over TLP emulation has some limitations:

1. The following APIs are not supported when provided with a DOCA DevEmu PCI EP that represents a TLP PF device:
   - `doca_devemu_pci_ep_get_num_free_dynamic_vf_msix` (will be supported in the future)
   - `doca_devemu_pci_ep_get_num_free_dynamic_vf_db` (will be supported in the future)
   - `doca_devemu_pci_ep_reset_msix_resources` (will be supported in the future)
   - `doca_devemu_pci_ep_reset_db_resources` (will be supported in the future)
   - `doca_devemu_pci_ep_get_num_vfs` (should be read from the SR-IOV capability at the PF's configuration space)
2. When configuring 1 or more Doorbells to a TLP PF's VFs using `doca_devemu_pci_rep_init_attr_set_vf_num_db`, the TLP PF must be set with same number of Doorbells using `doca_devemu_pci_ep_set_num_db` or `doca_devemu_pci_type_set_num_db`. This limitation will be removed in a future release.

5.4.20.3.8.5 Device Support

DOCA PCI TLP Device emulation requires a device to operate. For information on picking a device, see DOCA DevEmu PCI Device Support.

Some devices can allow different capabilities. For example:

- The maximum number of different PCI TLP types
- The maximum number of different TLP emulation physical functions (endpoints)
- The maximum number of Asynchronous Credit Grants
- The maximum number of BARs
- The maximum number of BAR regions
- The maximum BAR size
- The maximum number of doorbells
- The maximum number of MSI-X vectors
- The maximum configurable non-posted response timeout for a TLP channel

As the list of capabilities can be long, it is recommended to use the DOCA Capabilities Print Tool to get an overview of all the available capabilities.

Run the tool as root user as follows:

```
$ sudo /opt/mellanox/doca/tools/doca_caps -p <pci-address> -b devemu_pci
Example output:
PCI: 0000:03:00.0
    devemu_pci_tlp
        max_tlp_types                                8
        max_pf_devices                               5
        max_payload_size_per_tlp_channel_comp        32
        max_acg                                      1
        min_log_exp_bar_size                         12
        max_log_exp_bar_size                         24
        type_max_bars                                2
        bar_max_bar_regions                          15
        type_max_bar_regions                         15
        type_log_min_bar_size                        12
        type_log_max_bar_size                        30
        type_max_num_msix                            256
        type_max_num_db                              256
        bar_db_region_block_size                     4096
        bar_db_region_max_num_region_blocks          16
        type_max_bar_db_regions                      2
        bar_max_bar_db_regions                       2
        bar_db_region_start_addr_alignment           4096
        bar_msix_table_region_block_size             4096
        bar_msix_table_region_max_num_region_blocks  1
        type_max_bar_msix_table_regions              1
        bar_max_bar_msix_table_regions               1
        bar_msix_table_region_start_addr_alignment   4096
        bar_msix_pba_region_block_size               4096
        bar_msix_pba_region_max_num_region_blocks    1
        type_max_bar_msix_pba_regions                1
        bar_max_bar_msix_pba_regions                 1
        bar_msix_pba_region_start_addr_alignment     4096
        bar_transaction_region_block_size            64
        bar_transaction_region_max_num_region_blocks 1024
        type_max_bar_transaction_regions             4
        bar_max_bar_transaction_regions              4
        bar_transaction_region_start_addr_alignment  64
        bar_is_32_bit_supported                      unsupported
        bar_is_1_mb_supported                        unsupported
        bar_is_64_bit_supported                      supported
        tlp_type_supported                           supported
        ingress_mmio_read_fmt                        valid
        ingress_mmio_write_fmt                       valid
        ingress_mmio_read_type                       valid
        ingress_mmio_write_type                      valid
        ingress_mmio_read_t9                         invalid
        ingress_mmio_write_t9                        invalid
        ingress_mmio_read_tc                         invalid
        ingress_mmio_write_tc                        invalid
        ingress_mmio_read_t8                         invalid
        ingress_mmio_write_t8                        invalid
        ingress_mmio_read_attr_2                     invalid
        ingress_mmio_write_attr_2                    invalid
        ingress_mmio_read_ln                         invalid
        ingress_mmio_write_ln                        invalid
        ingress_mmio_read_th                         invalid
        ingress_mmio_write_th                        invalid
        ingress_mmio_read_td                         valid
        ingress_mmio_write_td                        valid
        ingress_mmio_read_ep                         valid
        ingress_mmio_write_ep                        valid
        ingress_mmio_read_attr_1_0                   invalid
        ingress_mmio_write_attr_1_0                  invalid
        ingress_mmio_read_at                         invalid
        ingress_mmio_write_at                        invalid
        ingress_mmio_read_length                     valid
        ingress_mmio_write_length                    valid
        ingress_mmio_read_req_id                     valid
        ingress_mmio_write_req_id                    valid
        ingress_mmio_read_tag                        invalid
        ingress_mmio_write_tag                       invalid
```

```
ingress_mmio_read_last_dw_be                valid
ingress_mmio_write_last_dw_be               valid
ingress_mmio_read_first_dw_be               valid
ingress_mmio_write_first_dw_be              valid
ingress_mmio_read_address_31_2              valid
ingress_mmio_write_address_31_2             valid
ingress_mmio_read_address_63_32             valid
ingress_mmio_write_address_63_32            valid
ingress_mmio_read_tlp_ph                    invalid
ingress_mmio_write_tlp_ph                   invalid
```

5.4.20.3.8.6 PCI TLP Type

Configurations

This section describes the configurations of the DOCA DevEmu PCI TLP Type object, that can be provided before start.

To find if a configuration is supported or what its min/max value is, refer to Device Support.

Mandatory Configurations

The following are mandatory configurations that must be provided before starting the PCI TLP type:

- A DOCA device that supports PCI TLP type. See Device Support.
- At least one BAR (either memory mapped or I/O mapped).

Optional Configurations

The following configurations are optional:

- One or more PCI capabilities
- One or more PCI extended capabilities
- The number of MSI-X vectors for MSI-X capability
- The number of Doorbells
- One or more memory mapped BARs (in addition to the mandatory one)
- One or more I/O mapped BARs (in addition to the mandatory one)
- One or more Doorbell regions
- An MSI-X table and PBA regions
- One or more Transaction regions

If these configurations are not set, then a default value is used.

5.4.20.3.8.7 PCI TLP Device

The following sections from DOCA DevEmu PCI Generic apply for a TLP device as well:

- PCI Device DB
- PCIe Device MSI-X Vector

Configurations

This section describes the configurations of the DOCA DevEmu PCI TLP Device object, that can be provided before start.

To find if a configuration is supported or what its min/max value is, refer to Device Support.

Mandatory Configurations

All mandatory configurations are provided during the creation of the PCI TLP device as follows:

- A DOCA DevEmu PCI TLP type object
- A DOCA Device Representor, representing an emulated function with the same type as the provided PCI TLP object type.

Optional Configurations

The following configurations are optional:

- The number of MSI-X vectors for MSI-X capability. By default, it is derived from the PCIe type.
- The number of Doorbells. By default, it is derived from the PCIe type.
- Set data path on DPA.

5.4.20.3.8.8 PCI TLP Channel

Configurations

This section describes the configurations of the DOCA DevEmu PCI TLP Channel object, that can be provided before start.

To start using the TLP Channel, users must first go through a configuration phase as described in DOCA Core Context Configuration Phase.

To find if a configuration is supported or what its min/max value is, refer to Device Support.

Mandatory Configurations

The following are mandatory configurations and must be provided before starting the TLP channel:

- A DOCA Device - provided at the TLP channel creation.
- A TLP channel requests handler (callback).

Optional Configurations

The following configurations are optional:

- Channel request user data size.
- Enable/Disable Asynchronous Credit Grant requests.
- A non-posted response timeout for non-posted TLP requests (see TLP Channel Non-Posted Response Timeout).
- A shared memory directory path (see TLP Channel Live Upgrade).
- Channel role, primary or secondary (see TLP Channel Live Upgrade).

Execution Phase

TLP Channel Events

When the TLP channel requests handler triggers (invoked by a preceding call to `doca_pe_progress`), ownership of the request is transferred to the application.

Each request provides an operation code for classification:

```
/**
 * @brief DOCA devemu PCI TLP channel request opcodes.
 */
enum doca_devemu_pci_tlp_channel_req_opcode {
    /**< Incoming TLP request opcode. */
    DOCA_DEVEMU_PCI_TLP_CHANNEL_REQ_OPCODE_TLP = 0x00,
    /**< Asynchronous Credit Grant (ACG) request opcode. */
    DOCA_DEVEMU_PCI_TLP_CHANNEL_REQ_OPCODE_ACG = 0x01,
    /**< PCI event request opcode. */
    DOCA_DEVEMU_PCI_TLP_CHANNEL_REQ_OPCODE_PCI_EVENT = 0x02,
    /**< INVALID request opcode. */
    DOCA_DEVEMU_PCI_TLP_CHANNEL_REQ_OPCODE_INVALID = 0x10000,
};
```

The application can retrieve the opcode using **doca_devemu_pci_tlp_channel_req_get_opcode**.

TLP Channel Request Downstream Port Identifier

The application can retrieve the downstream port identifier associated with a given request using **doca_devemu_pci_tlp_channel_req_get_dsp_id**. Since there are cases where there is no explicit ID associated with the request (e.g., when the request is associated with all the channel's downstream ports), this API may return one of the following special values:

```
/**
 * @brief The specific downstream port is implicitly determined from the request data.
 */
#define DOCA_DEVEMU_PCI_TLP_CHANNEL_REQ_DSP_ID_IMPLICIT 0xFD
/**
 * @brief The request is not associated with any specific downstream port.
 */
#define DOCA_DEVEMU_PCI_TLP_CHANNEL_REQ_DSP_ID_NONE 0xFE
/**
 * @brief The request is associated with all downstream ports associated with the corresponding TLP channel.
 */
#define DOCA_DEVEMU_PCI_TLP_CHANNEL_REQ_DSP_ID_ALL 0xFF
```

When supported by the device, the application can also obtain the host-visible BDF of the downstream port associated with a request using **doca_devemu_pci_tlp_channel_req_get_dsp_bdf()**. Whether the downstream port BDF is provided in TLP channel requests is queried using **doca_devemu_pci_tlp_cap_is_channel_req_dsp_bdf_supported()**. The BDF is only meaningful for requests whose downstream port identifier is a concrete (non-special) value as returned by **doca_devemu_pci_tlp_channel_req_get_dsp_id()**; it is returned in the standard PCI format, encoding bus[15:8], device[7:3], and function[2:0]. For example, it can be used to complete requests with an Unsupported Request (UR) status on behalf of the associated downstream port, when the application has not implemented any PCI entity beyond that port.

Handling and completing TLP Channel requests

Transaction Layer Packet requests (DOCA_DEVEMU_PCI_TLP_CHANNEL_REQ_OPCODE_TLP)

Processing

To parse an incoming TLP request, the application should use **doca_devemu_pci_tlp_channel_req_get_tlp_header** and read the the relevant TLP header fields according to the PCIe specification.

For write transactions, the payload can be retrieved using **doca_devemu_pci_tlp_channel_req_get_tlp_data**.

Completion

For non-posted requests (i.e. require a completion), the application retrieves a TLP completion header and data buffers using **doca_devemu_pci_tlp_channel_req_get_tlp_cpl_header** and **doca_devemu_pci_tlp_channel_req_get_tlp_cpl_data** and populate them according to the PCIe specification.

> ⚠ The completion header buffer size is always 3 DW (12 Bytes) to conform with the PCIe specification. Combined with the completion data, the total size cannot exceed **max_payload_size_per_tlp_channel_comp** .
>
> Populating more than the maximum size may result in an undefined behavior

Returning ownership of the request back to the channel is done by using **doca_devemu_pci_tlp_channel_req_complete_tlp** .

Completion of configuration space transactions in particular can be optimized by using **doca_devemu_pci_tlp_channel_req_complete_config_read** and **doca_devemu_pci_tlp_channel_req_complete_config_write** .

Asynchronous Credit Grant requests (DOCA_DEVEMU_PCI_TLP_CHANNEL_REQ_OPCODE_ACG)

When the application enables ACG requests using **doca_devemu_pci_tlp_channel_set_acg_enabled** , the TLP channel will forward all the ACG requests to the application right after the channel is started.

Typically, the application stores the ACG requests and only completes them at a later time, utilizing them for one of the functionalities introduced by the ACG completion operation mode:

```
/**
 * @brief DOCA devemu PCI TLP Asynchronous Credit Grant (ACG) completion operation modes.
 */
enum doca_devemu_pci_tlp_channel_req_acg_comp_opmode {
    /**< Flush ACG completion. */
    DOCA_DEVEMU_PCI_TLP_CHANNEL_REQ_ACG_COMP_OPMODE_FLUSH = 0x00,
    /**< MMIO WRITE ACG completion. */
    DOCA_DEVEMU_PCI_TLP_CHANNEL_REQ_ACG_COMP_OPMODE_MMIO_WRITE = 0x01,
};
```

Completion

Returning ownership of the request back to the channel is done by using **doca_devemu_pci_tlp_channel_req_complete_acg** .

> ⚠ When completing an ACG request while the channel is still running, the channel returns the request back to the application for reuse.

Flush (DOCA_DEVEMU_PCI_TLP_CHANNEL_REQ_ACG_COMP_OPMODE_FLUSH)

Flushing the ACG request simply returns ownership back to the channel without any effect.

This is used during a TLP Channel Handover for passing ACGs from one process to another (see TLP Channel handover Switchover stage).

Memory Write request (DOCA_DEVEMU_PCI_TLP_CHANNEL_REQ_ACG_COMP_OPMODE_MMIO_WRITE)

To send a memory write TLP request to the host, the application retrieves the ACG buffer using **doca_devemu_pci_tlp_channel_req_get_acg_buf** and populates it with a memory write request (both header and data) according to the PCIe specification.

PCI Event requests (DOCA_DEVEMU_PCI_TLP_CHANNEL_REQ_OPCODE_PCI_EVENT)

Processing

Each PCI Event request provides an operation mode for classification:

```
/**
 * @brief DOCA devemu PCI TLP PCI_EVENT request operation modes.
 */
enum doca_devemu_pci_tlp_channel_req_pci_event_opmode {
    /**< PERST# is asserted (enters reset). */
    DOCA_DEVEMU_PCI_TLP_CHANNEL_REQ_PCI_EVENT_OPMODE_PERST_ASSERT = 0x00,
    /**< PERST# is deasserted (released from reset). */
    DOCA_DEVEMU_PCI_TLP_CHANNEL_REQ_PCI_EVENT_OPMODE_PERST_DEASSERT = 0x01,
    /**< INVALID PCI_EVENT request operation mode. */
    DOCA_DEVEMU_PCI_TLP_CHANNEL_REQ_PCI_EVENT_OPMODE_INVALID = 0x10000,
};
```

The application can retrieve the opmode using **doca_devemu_pci_tlp_channel_req_get_pci_event_opmode**, and act accordingly (e.g., if opmode is **DOCA_DEVEMU_PCI_TLP_CHANNEL_REQ_PCI_EVENT_OPMODE_PERST_ASSERT**, put the relevant PCIe entities (endpoints/switches) into reset as defined by the PCIe specification).

Completion

Returning ownership of the request back to the channel is done by using **doca_devemu_pci_tlp_channel_req_complete_pci_event**.

TLP Channel Non-Posted Response Timeout

A TLP channel can be configured to bound the time the application is allowed to take when responding to a non-posted TLP request — that is, an MMIO Read or a Configuration Read/Write, all of which require a completion. Whether the device supports this mechanism, and the maximum value that may be configured, is queried using **doca_devemu_pci_tlp_cap_get_max_np_rsp_timeout()**. A returned maximum of 0 milliseconds indicates that the mechanism is not supported by the device.

When supported, the timeout is configured per channel, in milliseconds, using **doca_devemu_pci_tlp_channel_set_np_rsp_timeout()**, and read back using **doca_devemu_pci_tlp_channel_get_np_rsp_timeout()**. The configured value must not exceed the maximum reported by **doca_devemu_pci_tlp_cap_get_max_np_rsp_timeout()**, and a value of 0 disables the timeout. The timeout must be set while the channel is idle; once the channel has been started with **doca_ctx_start()**, the configuration is fixed for the remainder of the channel's lifetime. This configuration is not available for a channel created with **doca_devemu_pci_tlp_channel_create_from_export()**.

If the application fails to complete a non-posted request within the configured window, the device autonomously transitions the channel to a fatal state and, in accordance with the PCIe specification, returns error completions to the host for all outstanding non-posted TLP requests.

Detecting the fatal state

The application can detect the transition in one of two ways:

- Register a handler using **doca_devemu_pci_tlp_channel_event_fatal_state_register()** while the channel is idle. The registered **doca_devemu_pci_tlp_channel_event_fatal_state_handler_cb_t** callback is then invoked, during a call to **doca_pe_progress**, when the channel enters the fatal state.
- Poll **doca_devemu_pci_tlp_channel_get_is_fatal_state()**, which reports whether a started channel is currently in a fatal state. If no handler is registered, the fatal state is not signaled and must be detected by polling.

Recovery

A channel in a fatal state cannot be reused. Before it can be recreated, the application must perform the following steps, in order:

1. Stop and destroy all DOCA DevEmu PCI TLP devices associated with the channel (the devices whose TLP requests were received and processed through it).
2. Destroy the representor devices that were used to create those TLP devices, using **doca_devemu_pci_type_destroy_rep()**.
3. Stop and destroy the channel, following the destruction flow described for **doca_devemu_pci_tlp_channel_create()**.

Once these steps are complete, the channel, its representor devices, and its TLP devices can be recreated.

TLP Channel Live Upgrade

This section describes the process of a TLP channel handover between a running (source) application and a new upgraded (destination) application.

In a channel handover, the source and destination applications use dedicated APIs and an inter-process communication channel to export and import a running TLP Channel.

This process is fully transparent to the host driver, which can keep initiating transactions towards the NVIDIA switch ports associated with the channel.

Requirements

Channel

For a non-imported source channel:

1. Must be set with a valid shared memory directory path.
2. Must be started.

For an imported source channel:

1. Must be started.

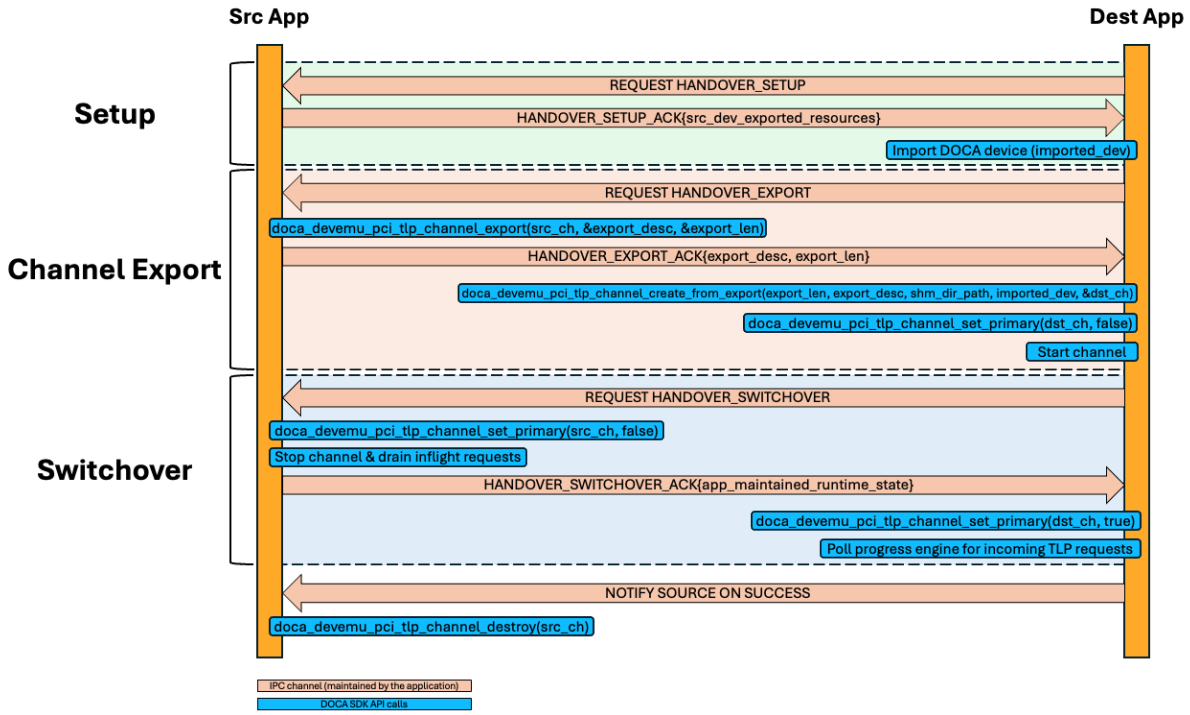
Application

The source and destination applications are required to share an inter-process communication channel for passing resources and for synchronization.

Exporting the DOCA device associated with the channel includes passing a file descriptor, which requires a special treatment (e.g., using dup(), SCM_RIGHTS, etc.).

TLP Channel Handover Stages

The following diagram shows all the TLP Channel Handover stages in high level:



Setup

The source application retrieves the **ibv_pd** associated with the DOCA device that used for the TLP channel creation using **doca_rdma_bridge_get_dev_pd**, and sends the associated resources to the destination application.

The destination application imports the **ibv_pd** and creates the DOCA device using **doca_rdma_bridge_open_dev_from_pd**.

For a detailed breakdown of the export and import process of an ibv_pd, please refer to the relevant Linux Manual Pages.

Channel Export

The source application creates a channel export descriptor using **doca_devemu_pci_tlp_channel_export**, and sends it to the destination application.

The destination application imports the channel using **doca_devemu_pci_tlp_channel_create_from_export**, sets the channel as secondary using

doca_devemu_pci_tlp_channel_set_primary, sets other configurations (e.g., request user data size), and starts the channel using **doca_ctx_start**.

> Setting the destination channel as secondary ensures only the source channel can forward requests to the (source) application, avoiding race conditions and undefined behavior. Later in the process, the source and destination channels switch roles.

> Some configurations are not supported when the channel is created from export, e.g., setting a shared memory directory path.

Switchover

The source application sets the channel as secondary using **doca_devemu_pci_tlp_channel_set_primary**, stops the channel using **doca_ctx_stop** and drains all inflight requests.

The destination application sets the channel as primary using **doca_devemu_pci_tlp_channel_set_primary**, and can start receiving incoming TLP Channel Requests using **doca_pe_progress**.

> Inflight ACG requests flushed during the draining stage can be retrieved by the destination application once it polls the channel for incoming requests.

> If the destination application faces any failures at this stage, the source application can rollback by re-starting the (stopped) source channel.

> It is the source application responsibility to pass all the necessary maintained PCIe-related resources to the destination application at this stage (e.g., a data structure with the latest configuration space state of a PCIe device).

5.4.20.3.8.9 DOCA DevEmu PCI TLP Samples

This section describes DOCA DevEmu PCI TLP samples.

The samples illustrate how to use the DOCA DevEmu PCI TLP API to do the following:

- Create a single PCIe endpoint and handle raw Transaction Layer Packets at the BlueField DPU level
- Carry out a TLP Channel Handover (Live Upgrade)
- Emulate a complete PCIe switch topology with multiple bridges and endpoints using (handling) raw Transaction Layer Packets.
- Hot-plug and hot-unplug TLP devices.

Structure

All the samples utilize the same TLP type configurations, which can be found at **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_type_config.h**

The structure for some samples is as follows:

- **/opt/mellanox/doca/samples/doca_devemu/<sample directory>**
  a. **dpu**
  b. **host**

Samples following this structure will have two binaries: **dpu** (1) and **host** (2), the former should be run on the BlueField and represents the controller of the emulated TLP devices, while the latter should be run on the host and represents the host driver.

For simplicity, the host (2) side is based on the VFIO driver, allowing development of a driver in user-space.

Running the Samples

1. Refer to the following documents:
   - DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - DOCA Troubleshooting for any issue you may encounter with the installation, compilation, or execution of DOCA samples.
2. To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file

   The binary **doca_<sample_name>[_dpu or _host]** is created under **/tmp/build/**.

```
cd /opt/mellanox/doca/samples/doca_devemu/<sample_name>[/dpu or /host]
meson /tmp/build
ninja -C /tmp/build
```

3. Sample usage example (**devemu_pci_device_tlp_handler**):
   a. BlueField side (**devemu_pci_device_tlp_handler_dpu**):

```
Usage: doca_devemu_pci_device_tlp_handler_dpu [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -p, --pci-addr                    The DOCA device PCI address. Format: XXXX:XX:XX.X or XX:XX.X
  -d, --handover-destination        Run as a TLP channel handover destination: connect to a source
 app and take over the TLP channel
  -s, --shm-dir-path                Absolute directory path for TLP channel shared memory. When
provided, enables live-upgrade handover (source: exports channel; destination: imports from this
path). When omitted, the sample runs in standalone mode without handover support.
```

   b. Host side (**devemu_pci_device_tlp_handler_host**):

```
Usage: devemu_pci_device_tlp_handler_host [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
```

```
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse all command flags from an input json file

Program Flags:
  -p, --pci-addr                    PCI address of the emulated device. Format: XXXX:XX:XX.X
  -g, --vfio-group                  VFIO group ID of the device. Integer
  -r, --region-index                The index of the transaction region as defined in
 devemu_pci_type_config.h. Integer
  -w, --write-data                  Data to write to the transaction region, if not provided then
 the sample will do a read instead. ASCII string
```

4. For additional information per sample, use the **-h** option:

```
/tmp/build/<sample_name> -h
```

Additional sample setup:

- The host samples require the emulated device to be discovered by the host side and the device is bound to the VFIO driver:
  - The samples expect 2 parameters **-p** (**--pci-addr**) and **-g** (**--vfio-group**) of the emulated device as seen by the host.
  - The host can use the script **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_vfio_bind.py** to bind the VFIO driver.

    ```
    $ sudo python3 /opt/mellanox/doca/samples/doca_devemu/devemu_pci_vfio_bind.py <pcie-address-of-emulated-dev>
    ```

    - The script is a python3 script which expects the PCIe address of the emulated device as a positional argument (e.g., **0000:3e:00.0**)
    - The script outputs the VFIO group ID
    - The script should be used only once after the device is discovered at the host for the first time
- To unbind the VFIO driver, use the script located at **/opt/mellanox/doca/samples/doca_devemu/devemu_pci_vfio_bind.py** from the host as follows:

  ```
  $ sudo python3 /opt/mellanox/doca/samples/doca_devemu/devemu_pci_vfio_bind.py <pcie-address-of-emulated-dev> --unbind
  ```

  - This python3 script expects the PCIe address of the emulated device as a positional argument (e.g., **0000:3e:00.0**) along with the **--unbind** argument

Samples

These samples are also available on GitHub.

PCI Device TLP Handler

This sample illustrates how to handle raw Transaction Layer Packets (TLPs) at the BlueField DPU level, providing low-level PCIe protocol processing capabilities for custom device emulation.

It also demonstrates how the host driver can interact with transaction regions for testing TLP processing functionality.

In addition, the sample supports TLP channel handover (live upgrade): a running source instance can transfer its active TLP channel to a newly started destination instance with zero host-visible disruption, enabling seamless in-service software upgrades.

The sample can also optionally expose an SR-IOV capability on the PF and serve the configuration and memory TLPs of each VF over the same endpoint.

This sample implements a single PCIe endpoint and is limited to a single TLP channel downstream port.

When the device supports querying the downstream port BDF for TLP channel requests, the sample uses it to serve TLPs that are associated with unused downstream ports: such requests are completed with an Unsupported Request (UR) status, using the downstream port's BDF as the completer ID. This lets the single-endpoint sample run even when more than one TLP downstream port is configured; otherwise, the number of TLP ports must be set to 1 via mlxconfig before running the sample.

The BlueField sample logic includes:

1. Initializing a custom PCI TLP type with specific device and vendor IDs.
2. Creating and configuring a TLP channel to receive and process raw PCIe transactions.
3. Setting up PCI configuration space structure with proper headers, BARs, and device capabilities.
4. Registering a TLP request handler callback function to process incoming transactions.
5. Creating a device representor to enable host-side device enumeration.
6. Initializing a TLP device context for transaction processing.
7. Optionally enabling SR-IOV: exposing an SR-IOV capability on the PF, bringing up the requested VFs, and routing each function's configuration and memory TLPs.
8. Processing various types of TLP requests including:
    a. Configuration space read/write operations (Type 0)
    b. Memory read/write operations
    c. Completion handling with appropriate status codes
9. Handling PCIe protocol-specific fields such as:
    a. Transaction tags and request IDs
    b. Bus/Device/Function (BDF) addressing
    c. Byte enable masks and data alignment
    d. Completion status codes (Success, Unsupported Request, etc.)
10. Parsing TLP headers to extract transaction information and routing details.
11. Generating appropriate TLP completion responses for posted and non-posted transactions.
12. Managing device state and configuration space updates.
13. Supporting TLP channel handover for live upgrade between two instances of the sample (running on 2 different processes).
14. The sample polls indefinitely for TLP requests until the user presses `[Ctrl+c]` to close the sample.

The Host sample logic includes:

1. Initializing the VFIO device with a matching PCIe address and VFIO group.
2. Mapping the transaction memory region from the BAR to the process address space.
3. Writing the values provided as input to the beginning of the transaction region, or reading existing values if no input is provided.

4. The sample demonstrates interaction with transaction regions that can be processed by the TLP handler on the BlueField DPU.

References:

- **/opt/mellanox/doca/samples/doca_devemu/**
  - **devemu_pci_device_tlp_handler/dpu/**
    - **devemu_pci_device_tlp_handler_dpu_sample.c**
    - **devemu_pci_device_tlp_handler_dpu_main.c**
    - **meson.build**
  - **devemu_pci_device_tlp_handler/host/**
    - **devemu_pci_device_tlp_handler_host_sample.c**
    - **devemu_pci_device_tlp_handler_host_main.c**
    - **meson.build**
  - **devemu_pci_common.h**
  - **devemu_pci_common.c**
  - **devemu_pci_host_common.h**
  - **devemu_pci_host_common.c**
  - **devemu_pci_type_config.h**

PCI Device TLP Bridge Handler

This sample illustrates how to emulate a complete PCIe switch topology with multiple bridges and endpoints using raw Transaction Layer Packets (TLPs). It demonstrates advanced PCIe fabric emulation capabilities by implementing a hierarchical bus architecture with upstream/downstream ports and multiple endpoint devices, in either a static mode or a hot-plug mode.

The sample logic includes:

1. Initializing a custom PCI TLP type for bridge and multi-endpoint topology.
2. Creating and configuring a TLP channel to receive and process PCIe transactions across the entire topology.
3. Initializing a complex PCIe device topology with one sub-topology per TLP channel DSP, each consisting of:
   a. 1 Upstream Switch Port (USP)
   b. N Downstream Switch Ports (DSPs) connected to the USP, where N is the number of endpoints assigned to this channel DSP
   c. N Single-PF endpoint devices, each connected to a DSP
4. Setting up PCI configuration space structures for both Type 0 (endpoint) and Type 1 (bridge) headers.
5. Implementing PCIe switch routing logic:
   a. Type 0 configuration requests routing to USPs
   b. Type 1 configuration requests routing to DSPs and endpoints
   c. Bus number management and secondary bus assignment
   d. Device number mapping across the switch fabric
6. Creating device representors for all endpoint devices to enable host enumeration, or delaying creation in case of hot-plug mode.
7. Initializing TLP device contexts for hardware-accelerated endpoint processing.
8. Allocating independent transaction memory regions for each endpoint device.

9. Processing various types of TLP requests with topology-aware routing:
    a. Configuration space read/write operations (Type 0 and Type 1)
    b. Memory read/write operations with BDF-based device lookup
    c. Bridge-specific configuration (bus numbers, memory windows)
    d. Endpoint-specific configuration (BARs, capabilities)
10. Handling PCIe capabilities for endpoints:
    a. PCIe Express capability
    b. MSI-X capability
    c. VPD (Vital Product Data) capability
    d. Power Management capability
11. Managing bridge and endpoint device states independently.
12. The sample polls indefinitely for TLP requests until the user presses `[Ctrl+c]` to close the sample.
13. In hot-plug mode, the user can hot-plug and hot-unplug on any available downstream port, via the terminal that runs the sample.

References:

- **/opt/mellanox/doca/samples/doca_devemu/**
    - **devemu_pci_device_tlp_bridge_handler/**
        - **devemu_pci_device_tlp_bridge_handler_config.h**
        - **devemu_pci_device_tlp_bridge_handler_sample.c**
        - **devemu_pci_device_tlp_bridge_handler_main.c**
        - **meson.build**
    - **devemu_pci_common.h**
    - **devemu_pci_common.c**
    - **devemu_pci_type_config.h**

5.4.20.4 DOCA DevEmu Virtio

5.4.20.4.1 Introduction

> The quality status of DOCA libraries is listed here.

DOCA DevEmu Virtio, which is part of the DOCA Device Emulation subsystem, introduces low-level software APIs that provide building blocks for developing and manipulating virtio devices using the device emulation capability of NVIDIA® BlueField®. This subsystem incorporates a core library that handles a common logic for various types of virtio devices, such as virtio-FS. One of its key responsibilities is managing the standard "device reset" procedure outlined in the virtio specification. This core library serves as a foundation for implementing shared functionalities across different virtio device types, ensuring consistency and efficiency in device operations and behaviors.

DOCA provides support for emulating virtio devices over the PCIe bus. The PCIe transport is commonly used for virtio devices. Configuration, discovery, and features related to PCIe (such as MSI-X and PCIe device hot plug/unplug) are managed through the DOCA DevEmu PCI APIs. This modular design enables each layer within the DOCA Device Emulation subsystem to manage its

own business logic and facilitates seamless integration with the other layers, ensuring independent functionality and operation throughout the system.

This subsystem also includes device-specific libraries for various virtio device types (e.g., a library for a virtio-FS device).

From the host's perspective, there is no difference between para-virtual, DOCA-emulated, and actual hardware devices. The host uses the same virtio device drivers to operate the device under all circumstances.

5.4.20.4.2 Prerequisites

Virtio device emulation is part of the DOCA Device Emulation subsystem. It is, therefore, recommended to read the following guides beforehand:

- DOCA Device Emulation
- DOCA DevEmu PCI

5.4.20.4.3 Environment

DOCA DevEmu Virtio is supported on the BlueField target only.

The BlueField must meet the following requirements

- DOCA version 2.7.0 or greater
- BlueField-3 firmware 32.41.1000 or higher

Please refer to the DOCA Compatibility Policy.

Library must be run with root privileges.

5.4.20.4.4 Architecture

The DOCA DevEmu Virtio core library provides the following software abstractions:

- Virtio type – extends the PCIe type, represents common/default virtio configurations of emulated virtio devices
- Virtio device – extends the PCIe device, represents an instance of an emulated virtio device
- Virtio IO context – represents a progress context which is responsible for processing virtio descriptors and their associated virtio queues

DOCA DevEmu Virtio library does not provide APIs to configure the entire BAR layout of the virtio device as this configuration is done internally. However, the library offers APIs to configure some of the registers within the common configuration structure (see Virtio Device).

5.4.20.4.4.1 Virtio Common Configuration

According to the virtio specification, the common PCIe configuration structure layout is as follows:

virtio_pci_common_cfg

```
struct virtio_pci_common_cfg {
    /* About the whole device. */
    le32 device_feature_select; /* read-write */
    le32 device_feature; /* read-only for driver */
    le32 driver_feature_select; /* read-write */
    le32 driver_feature; /* read-write */
    le16 config_msix_vector; /* read-write */
    le16 num_queues; /* read-only for driver */
    u8 device_status; /* read-write */
    u8 config_generation; /* read-only for driver */

    /* About a specific virtqueue. */
    le16 queue_select; /* read-write */
    le16 queue_size; /* read-write */
    le16 queue_msix_vector; /* read-write */
    le16 queue_enable; /* read-write */
    le16 queue_notify_off; /* read-only for driver */
    le64 queue_desc; /* read-write */
    le64 queue_driver; /* read-write */
    le64 queue_device; /* read-write */
    le16 queue_notify_data; /* read-only for driver */
    le16 queue_reset; /* read-write */
};
```

The DOCA DevEmu Virtio core library provides the ability to configure some of the listed registers using the appropriate setters.

5.4.20.4.4.2 Virtio Type

The virtio type extends the PCIe type and describes the common/default configuration of emulated virtio devices, including the common virtio configuration space registers (such as **num_queues**, **queue_size**, and others).

Virtio type is currently read-only (i.e., only getter APIs are available to retrieve information). The following methods can be used for this purpose:

- **doca_devemu_virtio_type_get_num_queues** – for getting the default initial value of the **num_queues** register for the associated virtio devices
- **doca_devemu_virtio_type_get_queue_size** – for getting the default initial value of the **queue_size** register for the associated virtio devices
- **doca_devemu_virtio_type_get_device_features_63_0** – for getting the default initial values of the **device_feature** bits (0-63) for the associated virtio devices
- **doca_devemu_virtio_type_get_config_generation** – for getting the default initial value of the **config_generation** register for the associated virtio devices

The default virtio type is extended by a virtio device's specific type (e.g., virtio-FS type) and cannot be created on demand.

5.4.20.4.4.3 Virtio Device

The virtio device extends the PCIe device. Before using the DOCA DevEmu Virtio device, it is recommended to read the guidelines of DOCA DevEmu PCI device and DOCA Core context configuration phase.

The virtio device is extended by a virtio-specific device (e.g., virtio FS device) and cannot be created on demand.

Virtio Device Configurations

The virtio device context can be configured to match the application use case and optimize the utilization of system resources.

Mandatory Configurations

The mandatory configurations are as follows:

- **doca_devemu_virtio_dev_set_num_required_running_virtio_io_ctxs** – to set the number of required running virtio IO contexts to be bound to the virtio device context. The virtio device context does not move to **running** state (according to the DOCA Core context state machine) before having this amount of running virtio IO contexts bound to it.
- **doca_devemu_virtio_dev_event_reset_register** – to register to the virtio device reset event. This configuration is mandatory

Optional Configurations

The optional configurations are as follows:

- **doca_devemu_virtio_dev_set_device_features_63_0** – to set the values of the **device_feature** bits (0-63). If not set, the default value is taken from the virtio type configuration.
- **doca_devemu_virtio_dev_set_num_queues** – to set the value of the **num_queues** register. If not set, the default value is taken from the virtio type configuration.
- **doca_devemu_virtio_dev_set_queue_size** – to set the value of the **queue_size** register for all virtio queues. If not set, the default value is taken from the virtio type configuration.

Events

DOCA DevEmu Virtio device exposes asynchronous events to notify about sudden changes, according to DOCA Core architecture.

> Common events are described in DOCA DevEmu PCI Device events and in DOCA Core Event.

Reset Event

The reset event allows users to receive notifications whenever the device reset flow is initialized by the device driver. Upon receiving this event, it is guaranteed that no further requests are routed to the user via any associated virtio IO context until the reset flow is completed.

To complete the reset flow the user must:

1. Flush all outstanding requests back to the virtio IO context associated with the request.
2. Perform one of the following:
    - Call **doca_devemu_virtio_dev_reset_complete**.
    - Follow FLR flow:
        i. **doca_ctx_stop** – stop the virtio device with its associated virtio IO contexts and wait until the device and its associated virtio IO contexts transition to **idle** state
        ii. **doca_ctx_start** – start the virtio device with its associated virtio IO contexts and wait until the device and its associated virtio IO contexts transition to **running** state

Now, the device and its associated virtio IO contexts should be fully operational again, the device is allowed to route new requests via any associated virtio IO context.

5.4.20.4.4.4 Virtio IO

The virtio IO context extends the DOCA Core context. Before using the DOCA DevEmu Virtio IO, it is recommended to read the guidelines of DOCA Core context configuration phase.

This context is associated with a single DOCA virtio device and is bound to the virtio device context upon start. The virtio IO context is a thread-unsafe object and is progressed by a single DOCA Core progress engine. Usually, users configure a single virtio IO context per BlueField core used by the application service.

The virtio IO context is responsible to route new incoming virtio requests towards the application and to complete handled requests back to the device driver. It can only route requests while in **running** state and when its associated virtio device is also in **running** state.

5.4.20.4.5 DOCA DevEmu Virtio-fs

> ⚠ This library is supported at alpha level; backward compatibility is not guaranteed.

5.4.20.4.5.1 Introduction

The DOCA DevEmu Virtio-fs library, part of the DOCA DevEmu Virtio subsystem, provides low-level software APIs. These APIs serve as the building blocks for developing and manipulating virtio-fs (virtio-filesystem) devices by leveraging the device emulation capabilities of NVIDIA® BlueField® DPUs.

Architecture and Library Interaction

This library is part of a modular subsystem, where each library manages its own logic:

- DOCA DevEmu PCI: Manages the PCIe transport layer, including configuration, discovery, and features like MSI-X and device hot plug/unplug.
- DOCA Virtio Common: Manages generic virtio logic, such as handling common virtio registers and the virtio device reset flow.
- DOCA DevEmu Virtio-fs: Manages the specific logic for the virtio-filesystem device.

This layered design allows for seamless integration and independent operation of each component.

Core Functionality

The DOCA DevEmu Virtio-fs library efficiently handles virtio descriptors that carry FUSE requests from the device driver. It translates these into abstract virtio-fs requests that are then routed to the user application. This process abstracts away the underlying device-specific acceleration details.

The library supports FUSE versions 7.31–7.43.

User Responsibilities

Users of this library are responsible for developing a virtio-fs controller. This controller application must:

1. Manage the underlying DOCA DevEmu Virtio-fs device.
2. Receive the abstracted DOCA Virtio-fs requests.
3. Process these requests according to the virtio-fs and FUSE specifications.
4. Translate the FUSE-based commands into the appropriate protocol for an external backend file system.

   The backend file system itself is outside the scope of DOCA.

5.4.20.4.5.2 Prerequisites

Virtio-fs device emulation is part of DOCA DevEmu Virtio subsystem. It is, therefore, recommended to read the following guides before proceeding:

- DOCA Device Emulation
- DOCA DevEmu PCI
- DOCA DevEmu Virtio

5.4.20.4.5.3 Environment

DOCA DevEmu Virtio-fs is supported on the BlueField target only. The BlueField must meet the following requirements:

- DOCA version 2.7.0 or greater
- BlueField-3 firmware 32.41.1000 or higher

Please refer to the DOCA Compatibility Policy.

Library must be run with root privileges.

Perform the following:

1. Configure BlueField to work in DPU mode as described in BlueField Modes of Operation.
2. Enable emulation by running the following on the host or DPU:

   ```
   host/dpu> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s VIRTIO_FS_EMULATION_ENABLE=1
   ```

3. Configure the number of static virtio-fs physical functions and the number of MSIX for each physical function to expose. This can be done by running the following command on the DPU:

```
host/dpu> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s VIRTIO_FS_EMULATION_NUM_PF=2
VIRTIO_FS_EMULATION_NUM_MSIX=18
```

4. Perform a BlueField system reboot for the **mlxconfig** settings to take effect.

Hot-plug

Host Configuration

With a Linux environment on host OS, additional kernel boot parameters are required to support the hot-plug feature:

- For Intel machines:

```
intel_iommu=on iommu=pt pci=realloc
```

- For AMD machines:

```
iommu=pt pci=realloc
```

> ⚠ On AMD machines, hotplug may not work.

Firmware Configuration

When PCIe switch emulation is enabled, BlueField can support 1 hotplug virtio-fs function. These PCIe functions are shared among all BlueField users and applications and may hold hot-plugged devices of type NVMe, virtio-blk, virtio-fs, or other (e.g., virtio-net).

To enable PCIe switch emulation and configure one hot-plugged port, run:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s PCI_SWITCH_EMULATION_ENABLE=1 PCI_SWITCH_EMULATION_NUM_PORT=2
```

PCI_SWITCH_EMULATION_NUM_PORT equals 1 plus the number of hot-plugged PCIe functions.

5.4.20.4.5.4 Architecture

The DOCA DevEmu Virtio-fs library provides the following main software abstractions:

- The virtio-fs type – extends the virtio type; represents common/default virtio-fs configurations of emulated virtio-fs devices
- The virtio-fs device – extends the virtio device; represents an instance of an emulated virtio-fs device
- The virtio-fs IO context – extends the virtio IO context; represents a progress context responsible for processing virtio descriptors, carrying FUSE requests, and their associated virtio queues (e.g., hiprio, request, admin, and notification queues).
- The virtio-fs request

Virtio-fs Feature Bits

According to the virtio specification, a virtio-fs device may report support for **VIRTIO_FS_F_NOTIFICATION** which indicates the ability to handle FUSE notify messages sent via the notification queue.

> ⚠ Currently, DOCA does not support reporting the **VIRTIO_FS_F_NOTIFICATION** feature to the driver.

Virtio-fs Configuration Layout

According to the virtio specification, the virtio-fs configuration structure layout is as follows:

virtio_fs_config

```
struct virtio_fs_config {
    char tag[36];
    le32 num_request_queues;
    le32 notify_buf_size;
};
```

The **tag** and **num_request_queues** fields are always available. The **notify_buf_size** field is only available when **VIRTIO_FS_F_NOTIFICATION** is set.

> ⚠ Currently, there is no support for reporting the **VIRTIO_FS_F_NOTIFICATION** feature to the driver. Therefore, **notify_buf_size** field is not available.

Virtio-fs Type

The virtio-fs type extends the virtio type and describes the common/default configuration of emulated virtio-fs devices, including some of the virtio-fs configuration space registers (e.g., **num_request_queues**).

Currently, the virtio-fs type is read-only (i.e., only getter APIs are available to retrieve information). The following method can be used for this purpose:

- **doca_devemu_vfs_type_get_num_request_queues** – to get the default initial value of the **num_request_queues** register for the associated virtio-fs devices

DOCA supports the default virtio-fs type. To retrieve the default virtio-fs type, users use the following method:

- **doca_devemu_vfs_is_default_vfs_type_supported** – check if the default DOCA Virtio-fs type is supported by the device. If supported:
  - **doca_dev_open** – open supported DOCA device
  - **doca_devemu_vfs_find_default_vfs_type_by_dev** – get the default DOCA Virtio-fs type associated with the device

Virtio-fs Device

The virtio-fs device extends the virtio device. Before using the DOCA DevEmu Virtio-fs device, it is recommended to read the guidelines of DOCA DevEmu Virtio device, DOCA DevEmu PCI device, and DOCA Core context configuration phase.

This section describes how to create, configure, and operate the virtio-fs device.

Virtio-fs Device Configurations

The virtio-fs emulated device might be in several different visibility levels from the host point of view:

- Visible/non-visible to the PCIe subsystem – If the device is visible to the PCIe subsystem, the user is not able to configure PCIe-related parameters (e.g., number of MSI-X vector, `subsystem_id`).
- Visible/non-visible to the virtio subsystem – If the device is visible to the virtio subsystem, the user is not be able to configure virtio-related parameters (e.g., number of queues, `queue_size`).

The flow for creating and configuring a virtio-fs device is as follows:

1. `doca_devemu_vfs_dev_create` – Create a new DOCA DevEmu Virtio-fs device instance.
2. `doca_devemu_vfs_dev_set_tag` – Set a unique tag for the device according to the virtio specification.
3. `doca_devemu_vfs_dev_set_num_request_queues` – Set the number of request queues for the device.
4. `doca_devemu_vfs_dev_set_vfs_req_user_data_size` – Set the user data size of the virtio-fs request. If set, a buffer with this size is allocated for each DOCA DevEmu Virtio-fs on behalf of the user.
5. Configure virtio-related parameters as described in DOCA Virtio configurations.

   > ⚠ `doca_devemu_virtio_dev_set_num_queues` should be equal to the number of request queues +1 (for the `hiprio` queue) since DOCA does not currently support the virtio-fs notification queue.

6. Configure PCIe-related parameters as described in DOCA DevEmu PCI configurations.
7. `doca_ctx_start` – Start the virtio-fs device context to finalize the configuration phase.
   - The virtio-fs device object follows the DOCA context state machine as described in DOCA Core context state machine
   - The virtio-fs device context moves to `running` state after the initial number of virtio IO contexts is bound to it and turns to `running` state, as described at DOCA DevEmu Virtio configurations

At this point, the DOCA Devemu Virtio-fs context is fully operational.

Mandatory Configurations

The following are mandatory configurations:

- `doca_devemu_vfs_dev_set_tag` – set a unique tag for the device

Optional Configurations

The optional configurations are as follows:

- `doca_devemu_vfs_dev_set_num_request_queues` – set the number of request queues for the device. If not set, the default value is taken from the virtio-fs type configuration.

- **doca_devemu_vfs_dev_set_vfs_req_user_data_size** – set the user data size of the virtio-fs request. If not set, user data size defaults to 0.

Virtio-fs Device Events

DOCA DevEmu Virtio-fs device exposes asynchronous events to notify about changes that happen out of the blue, according to the DOCA Core architecture.

Common events are described in DOCA DevEmu Virtio device events, DOCA DevEmu PCI device events and in DOCA Core event.

Virtio-fs IO

The virtio-fs IO context extends the Virtio IO Context. To start using the DOCA DevEmu Virtio-fs IO it is recommended to read the guidelines of DOCA DevEmu Virtio IO and DOCA Core context configuration phase.

This section describes how to create, configure and operate the virtio-fs IO context.

Virtio-fs IO Configurations

The flow for creating and configuring a virtio-fs IO context should be as follows:

1. **doca_devemu_vfs_io_create** – Create a new DOCA DevEmu Virtio-fs IO instance.
2. **doca_devemu_vfs_io_event_vfs_req_notice_register** – Register event handler for incoming virtio-fs requests.
3. **doca_ctx_start** – Start the virtio-fs IO context to finalize the configuration phase. The virtio-fs IO object follows the DOCA Core context state machine. The virtio-fs device context moves to **running** state after the initial number of virtio-fs IO contexts is bound to it and moves to **running** state (as described at DOCA DevEmu Virtio configurations).

Mandatory Configurations

The following are mandatory configurations:

- **doca_devemu_vfs_io_event_vfs_req_notice_register** – Register event handler for incoming virtio-fs requests is mandatory

Virtio-fs Request

The virtio-fs request object serves as an abstraction for handling requests arriving on virtio-fs queues, including high-priority, request, or notification queues. These requests are initially generated by the device driver through created virtio queues and then routed to the user via a registered event handler, which is set up using **doca_devemu_vfs_io_event_vfs_req_notice_register**, on the associated virtio IO context. This event handler, issued by the DOCA Virtio FS library, ensures that users can receive and process virtio-fs requests effectively within their application. Once the event handler is called, the ownership of the virtio-fs request and the associated request user data move to the user. The request ownership moves back to the associated virtio IO context once it is completed by the user by calling **doca_devemu_vfs_req_complete**.

The following APIs operate a virtio-fs request:

1. **doca_devemu_vfs_req_get_datain** – Get a DOCA buffer representing the data-in of the virtio-fs request. This DOCA buffer represents the host memory for the device-readable part of the request according to the virtio specification.
2. **doca_devemu_vfs_req_get_dataout** – Get a DOCA buffer representing the data-out of the virtio-fs request. This DOCA buffer represents the host memory for the device-writable part of the request according to the virtio specification.
3. **doca_devemu_vfs_req_complete** – Complete the virtio-fs request. The associated virtio-fs IO context completes the request toward the device driver according to the virtio-fs specification.

5.4.20.4.5.5 Discovery

Emulated virtio-fs PCIe functions are represented by a **doca_devinfo_rep**. To find the suitable **doca_devinfo_rep** that is used as the input parameter for **doca_devemu_vfs_dev_create**, users should first discover the existing device representors using the below:

1. **doca_devinfo_create_list** – Get a list of all DOCA devices.
2. **doca_devemu_vfs_is_default_vfs_type_supported** – Check whether the device can manage device associated to virtio-fs type.
3. If supported:
    a. **doca_dev_open** – Get an instance of the DOCA device that can be used as virtio-fs emulation manager.
    b. **doca_devemu_vfs_find_default_vfs_type_by_dev** – Get the default virtio-fs device type.
    c. **doca_devemu_vfs_type_as_pci_type** – Cast virtio-fs type to PCIe type.
    d. **doca_devemu_pci_type_rep_list_create** – Create a list of all available representor devices for the virtio-fs type.
4. At this point, the user can choose the preferred representor device, open it using **doca_dev_rep_open**, and proceed with the flow described in section "Virtio-fs Device Configurations".

5.4.20.4.5.6 Initialization

This section describes the initialization flow of a DOCA DevEmu Virtio-fs device and one or more DOCA DevEmu Virtio-fs IO contexts (4 in this example). In this procedure, the user sets up and prepares the environment before starting to receive control path events (from the virtio-fs device context) and IO requests (from the virtio-fs IO contexts). During initialization, the user should configure various essential components to ensure correct behavior.

The user should perform the following:

1. Choose 4 Arm cores to run the application threads on.
2. Create 4 DOCA Core progress engine (PE) objects (**pe1** , **pe2** , **pe3** , **pe4**).
3. Find the suitable representor device according to the Discovery flow or any other method.
4. Create, configure, and start a new virtio-fs device according to the virtio-fs device configuration flow. Assume **pe1** is associated with the virtio-fs device and **doca_devemu_virtio_dev_set_num_required_running_virtio_io_ctxs** is set to 4.

5. Create, configure, and start 4 new virtio-fs IO contexts according to the virtio-fs IO configuration flow. Assume **pe1**, **pe2**, **pe3**, and **pe4** are associated with each of the 4 virtio-fs IO contexts respectively.
6. At this point, the 4 virtio-fs IO contexts transition to **running** state, followed by the virtio-fs device context transitioning to **running** state.

> ⚠ During the initialization flow, it is guaranteed that no virtio/PCIe control path or IO path events are generated until the virtio-fs device has transitioned to **running** state.

5.4.20.4.5.7 Teardown

This section describes the teardown flow of DOCA DevEmu Virtio-fs device and one or more DOCA DevEmu Virtio-fs IO contexts (4 in this example). In this procedure, the user cleans all the resources allocated in the initialization flow and all the outstanding events and requests.

The user should perform the following:

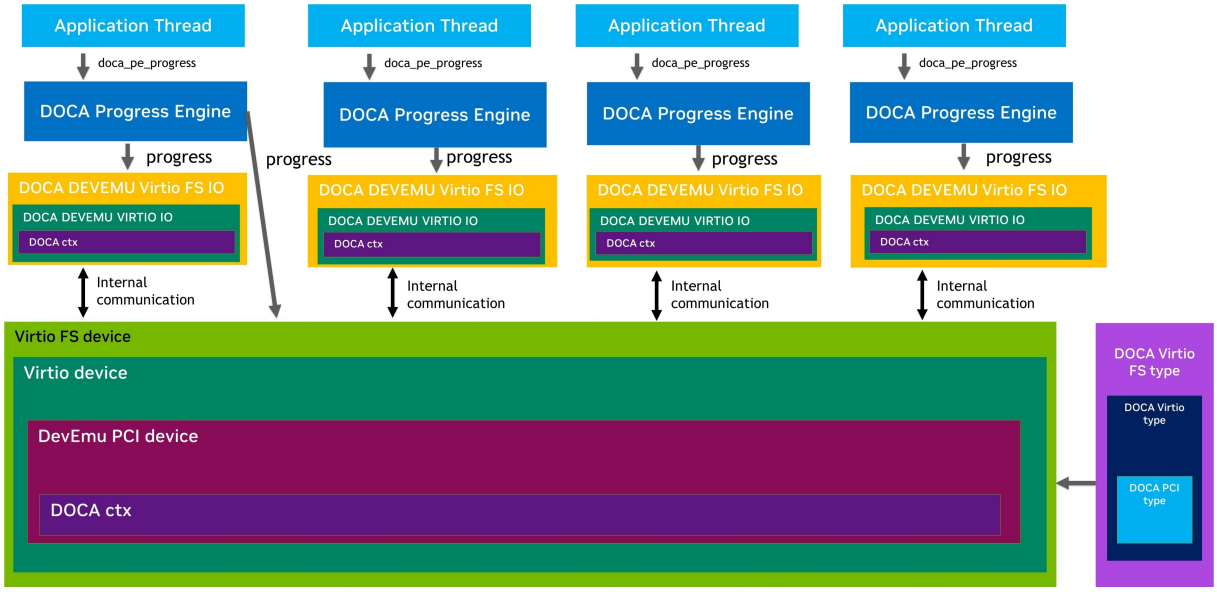
1. Start the teardown flow by calling **doca_ctx_stop**. This causes the DOCA Virtio-fs device context to transition to **stopping** state. It is guaranteed that no virtio/PCIe control path events is generated during this state.
2. Call **doca_ctx_stop** for any DOCA Virtio-fs IO context. This causes the DOCA Virtio-fs IO context to transition to **stopping** state. It is guaranteed that no IO path events are generated during this state.
3. Flush all outstanding virtio-fs requests to the associated virtio-fs IO contexts by calling **doca_devemu_vfs_req_complete**. Upon completing all the requests associated with a virtio-fs IO context, the DOCA Virtio-fs IO context transitions to **idle** state.
4. At this point, it is safe to destroy the virtio-fs IO context by calling **doca_devemu_vfs_io_destroy**. Destroying a virtio-fs IO context not in **idle** state will fail.
5. Once all 4 virtio-fs IO contexts associated with the virtio-fs device transition to **idle** state, the DOCA Virtio-fs device context transitions to **idle** state as well.
6. At this point, it is safe to destroy the virtio-fs device context by calling **doca_devemu_vfs_dev_destroy**. Destroying a virtio-fs device context not in **idle** state will fail.

5.4.20.4.5.8 Execution Phase

This section describes execution on BlueField Arm cores using several DOCA Core PE objects (one per core):

1. Choose 4 Arm cores to run the application threads on.
2. Create 4 DOCA Core PE objects. The application threads should periodically call **doca_pe_progress** to advance all DOCA contexts associated with the PE.
3. Create, configure, and start the DOCA Virtio-fs device.
4. Create, configure, and start 4 DOCA Virtio-fs IO contexts.

The progress of DOCA Virtio-fs objects is illustrated by the following diagram:



Control Path

The DOCA Virtio-fs device context extends the DOCA Virtio device context (which extends the DOCA PCIe device context). This means that the DOCA Virtio-fs device control path is comprised by all the object it extends (i.e., DOCA Context, DOCA DevEmu PCI device, and DOCA DevEmu Virtio device).

The following events can be triggered by a virtio-fs device context:

- DOCA context state change events as described in DOCA Core context state machine and in DOCA DevEmu PCI state machine
- DOCA DevEmu PCI FLR flow
- DOCA DevEmu Virtio reset flow

The DOCA Virtio-fs IO context extends the DOCA Virtio IO context (which extends the DOCA core context). This means that the DOCA Virtio-fs IO context control path is comprised by all the object it extends (i.e., DOCA Context and DOCA DevEmu Virtio IO).

The following events can be triggered by a Virtio-fs IO context:

- DOCA context state change events as described in DOCA Core context state machine

In addition to the control path events, the DOCA DevEmu Virtio-fs IO context also produces IO path events as described in IO path.

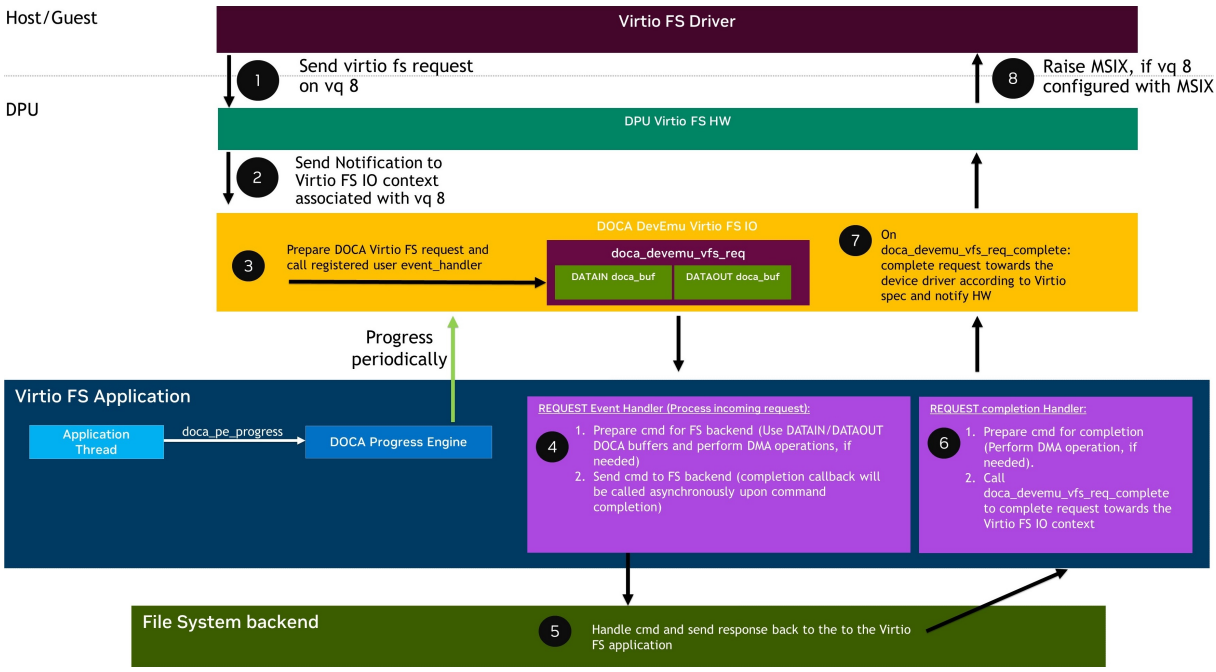
IO Path

This section describes the flow for a single virtio-fs request sent by the device driver until its completion.

It is assumed that the user properly configured an event handler for an incoming virtio-fs request as explained in section "Virtio-fs IO Configurations".

It is also assumed that the user is familiar with the virtio-fs specification and has the ability to perform DMA operations to/from the host using DOCA DMA or any other suitable method.

The DOCA virtio-fs flow is illustrated in the following diagram:



5.4.20.4.6 DOCA DevEmu Virtio-net

5.4.20.4.6.1 1. Introduction

The DOCA DevEmu Virtio-net library provides low-level APIs for building emulated VirtIO network devices on NVIDIA BlueField DPUs. It is part of the DOCA DevEmu Virtio subsystem and is intended to be used with DOCA DevEmu PCI TLP emulation.

The reference application, `vnet_pci_dev`, demonstrates how to build a VirtIO-net PCI device controller around this library. The application owns PCI/TLP emulation, VirtIO feature negotiation, virtqueue lifecycle, control virtqueue handling, hotplug, reset, and live-update coordination. Packet data-path processing is not surfaced as per-packet application callbacks; RX/TX virt-queues are handled by the vNET offload library and hardware path once the offload engine is enabled.

1.1. Architecture and Library Interaction

The DevEmu Virtio-net stack follows the same layered design as other DOCA DevEmu Virtio libraries:

DOCA DevEmu PCI: manages the TLP PCI transport: PCI type creation, representors, endpoints, TLP channel events, MSI-X, and hotplug support.

DOCA DevEmu Virtio Common: manages common VirtIO offload concepts: offload engine lifecycle, virtqueue configuration, virtqueue start/stop/enable/disable, IO binding, virtqueue debug state, and reset-sensitive state transitions.

DOCA DevEmu Virtio-net: manages VirtIO-net-specific objects: network offload engine, RX/TX/control virtqueues, MAC/MTU configuration, representor lookup, control virtqueue request routing, and DevEmu Virtio-net counters.

The reference app adds a complete VirtIO PCI device model around the library. It implements config-space and MMIO TLP handling, tracks VirtIO common configuration, exposes network device configuration, and drives the library according to host driver status transitions.

1.2. Core Functionality

The DevEmu Virtio-net library provides these main capabilities:

- Create a VirtIO-net PCI TLP type.
- Create a per-endpoint VirtIO-net offload engine.
- Configure network-specific parameters such as MAC address and MTU.
- Create RX, TX, and control virtqueues.
- Start, enable, disable, stop, and destroy virtqueues through the common VirtIO API.
- Route control virtqueue commands to the application as `doca_devemu_vnet_ctrl_req`.
- Expose DevEmu Virtio-net virtqueue counters and common VirtIO queue debug state.
- Export/import offload engine state for live update and recovery flows.

A key difference from DevEmu Virtio-blk: DevEmu Virtio-net does not expose per-packet request objects like **doca_devemu_vblk_req** . The application does not receive every packet. Instead, it configures RX/TX virtqueues and lets the DevEmu Virtio-net offload engine handle packet movement. The only request-style event exposed to the app is the control virtqueue request.

1.3. User Responsibilities

A DevEmu Virtio-net controller application is responsible for:

- Creating and configuring the PCI TLP type.
- Creating PCI representors and TLP endpoint devices.
- Emulating VirtIO PCI configuration and device-specific network config.
- Advertising feature bits consistently with what the application and library can support.
- Creating, configuring, starting, binding, enabling, disabling, stopping, and destroying RX/TX/CVQ virtqueues.
- Creating an IO context only when control virtqueue commands need to be routed to the app.
- Handling **doca_devemu_vnet_ctrl_req** events and completing them with VirtIO-net ACK status.
- Handling device reset, hotplug, and teardown without blocking the TLP progress path.
- Managing any backend network policy, representor setup, or system integration outside the DevEmu Virtio-net library.

5.4.20.4.6.2 2. Environment and Prerequisites

VirtIO-net TLP emulation depends on BlueField TLP device emulation support. The environment should be configured for DPU mode and TLP emulation.

1. Configure BlueField to work in DPU mode as described in BlueField Modes of Operation.
2. Enable TLP emulation by running the following on the DPU:

```
[dpu] mlxconfig -y -d /dev/mst/mt41692_pciconf0 set TLP_EMULATION_ENABLE=1
[dpu] mlxconfig -y -d /dev/mst/mt41692_pciconf0 set TLP_EMULATION_NUM_PF=32
```

3. Perform a BlueField system cold reboot for the **mlxconfig** settings to take effect.

For hotplug, PCI switch emulation and host kernel support are also required.

1. for Host platforms with intel CPU, need config grub file with:

```
intel_iommu=on iommu=pt pci=realloc
```

2. for DPU side, need config with:

```
[dpu] mlxconfig -y -d /dev/mst/mt41692_pciconf0 set PCI_SWITCH_EMULATION_ENABLE=1
[dpu] mlxconfig -y -d /dev/mst/mt41692_pciconf0 set PCI_SWITCH_EMULATION_NUM_TLP_PORT=1
[dpu] mlxconfig -y -d /dev/mst/mt41692_pciconf0 set PCI_SWITCH_EMULATION_NUM_PORT=2
```

PCI_SWITCH_EMULATION_NUM_PORT equals 1 plus the number of hot-plugged PCIe functions.
for 32 vNET devices, need config **mlxconfig -y -d /dev/mst/mt41692_pciconf0 s PF_BAR2_ENABLE=0 PER_PF_NUM_SF=1 PF_SF_BAR_SIZE=8 PF_TOTAL_SF=33** .

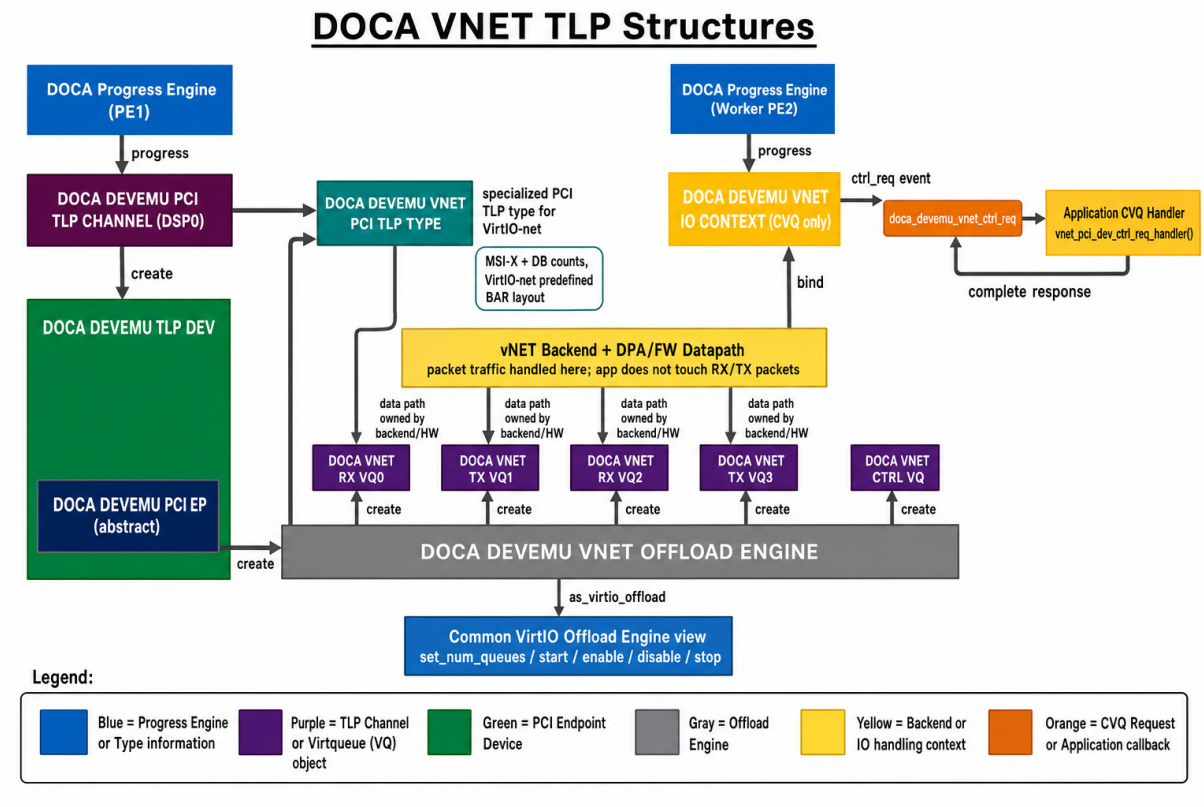
The **vnet_pci_dev** reference app additionally expects exactly one physical NV switch TLP downstream port for its topology.
During initialization it queries the TLP channel and fails if the number of downstream ports is not `1`.

The reference topology is:

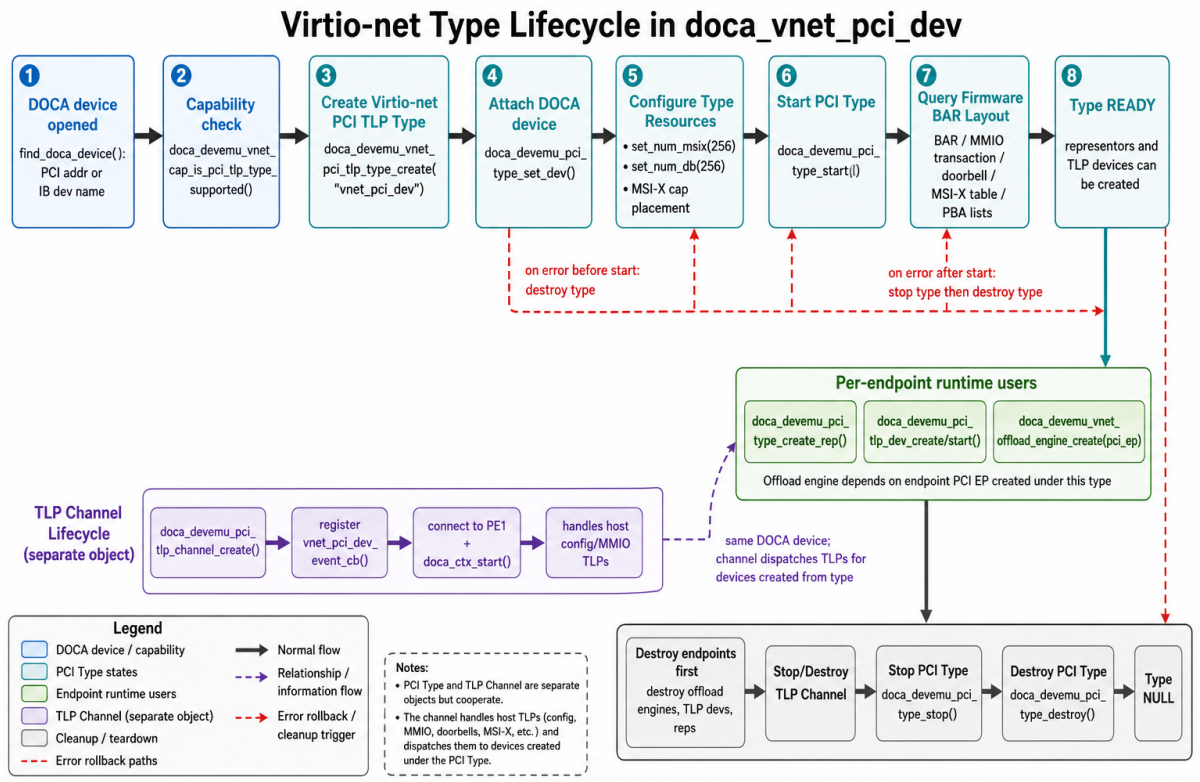
```
USP
|-- DSP[0] -> EP[0] virtio-net
|-- DSP[1] -> EP[1] virtio-net
|-- ...
`-- DSP[N] -> EP[N] virtio-net
```

- One upstream bridge, or USP.
- `N` downstream bridge slots, or DSPs.
- `N` emulated VirtIO-net endpoints.

2.1. Main Abstractions



2.2. Virtio-net Type



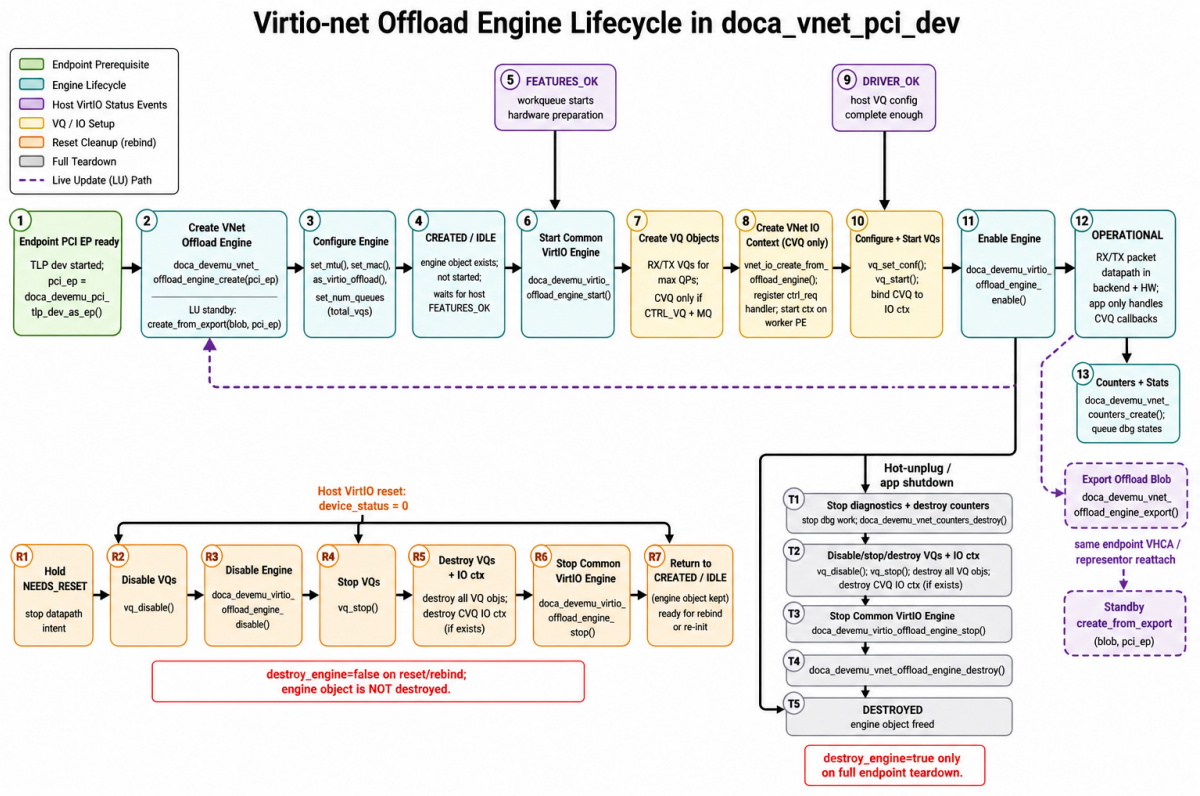
The VirtIO-net type is a PCI TLP type specialized for VirtIO-net emulation.

Important APIs:

- `doca_devemu_vnet_cap_is_pci_tlp_type_supported()`
- `doca_devemu_vnet_pci_tlp_type_create()`
- `doca_devemu_pci_type_set_dev()`
- `doca_devemu_pci_type_set_num_msix()`
- `doca_devemu_pci_type_set_num_db()`
- `doca_devemu_pci_tlp_type_set_pci_cap_conf()`
- `doca_devemu_pci_type_start()`

The reference app creates the type in `vnet_pci_device_init()`, sets MSI-X and doorbell counts, configures MSI-X capability placement, starts the type, and then queries the predefined VirtIO-net BAR layout provided by firmware.

2.3. Virtio-net Offload Engine



The offload engine is the per-endpoint DevEmu Virtio-net object that owns virtqueue state and network-specific configuration.

Important APIs:

- `doca_devemu_vnet_offload_engine_create()`
- `doca_devemu_vnet_offload_engine_create_from_export()`
- `doca_devemu_vnet_offload_engine_export()`
- `doca_devemu_vnet_offload_engine_destroy()`
- `doca_devemu_vnet_offload_engine_set_mtu()`
- `doca_devemu_vnet_offload_engine_set_mac()`

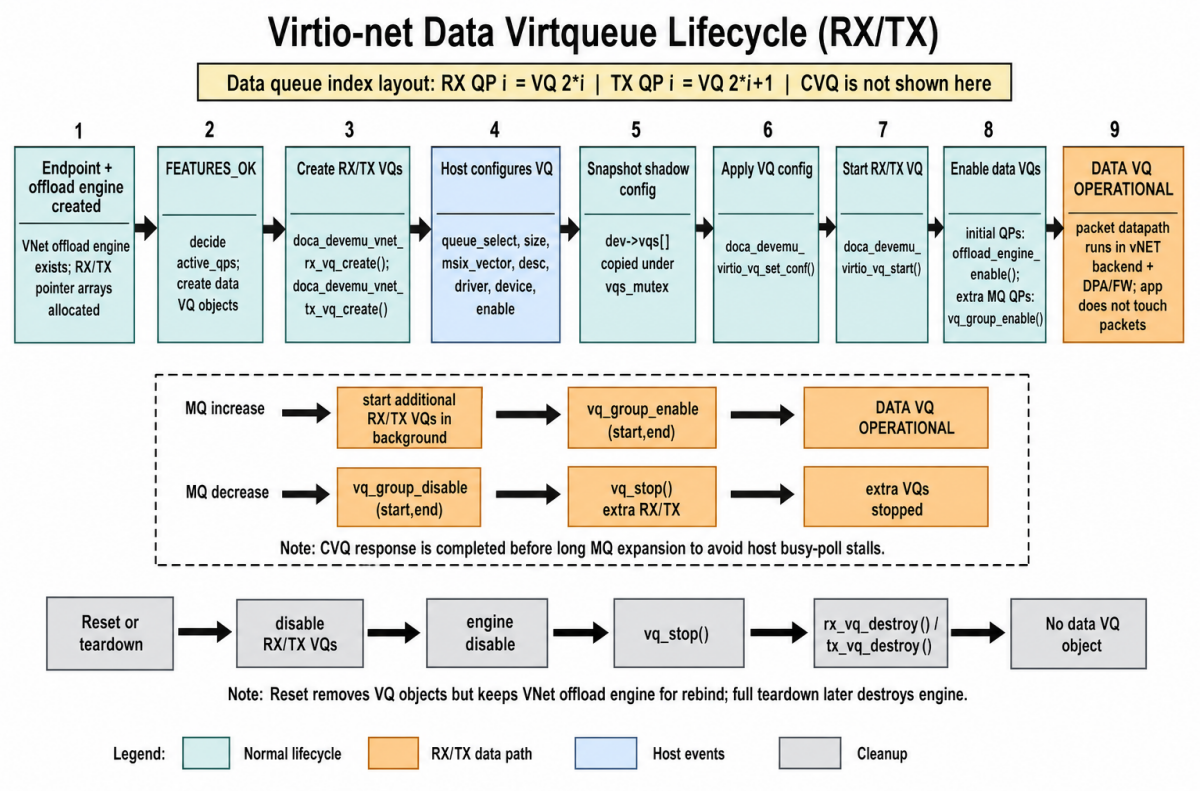
- `doca_devemu_virtio_offload_engine_set_feature_enabled()`
- `doca_devemu_vnet_offload_engine_get_rep()`
- `doca_devemu_vnet_offload_engine_as_virtio_offload()`

After casting to common VirtIO offload:

- `doca_devemu_virtio_offload_engine_set_num_queues()`
- `doca_devemu_virtio_offload_engine_start()`
- `doca_devemu_virtio_offload_engine_enable()`
- `doca_devemu_virtio_offload_engine_disable()`
- `doca_devemu_virtio_offload_engine_stop()`

The reference app creates the offload engine when a PCI endpoint is created, sets MTU and MAC, configures total queue count, but deliberately delays `doca_devemu_virtio_offload_engine_start()` until the host reaches `FEATURES_OK`.

2.4. Virtqueues



VirtIO-net uses separate queue types:

- RX virtqueue: `doca_devemu_vnet_rx_vq`
- TX virtqueue: `doca_devemu_vnet_tx_vq`
- Control virtqueue: `doca_devemu_vnet_ctrl_vq`

Key APIs:

- `doca_devemu_vnet_rx_vq_create()`
- `doca_devemu_vnet_tx_vq_create()`
- `doca_devemu_vnet_ctrl_vq_create()`
- `doca_devemu_vnet_rx_vq_as_vq()`
- `doca_devemu_vnet_tx_vq_as_vq()`
- `doca_devemu_vnet_ctrl_vq_as_vq()`
- `doca_devemu_vnet_rx_vq_destroy()`
- `doca_devemu_vnet_tx_vq_destroy()`
- `doca_devemu_vnet_ctrl_vq_destroy()`

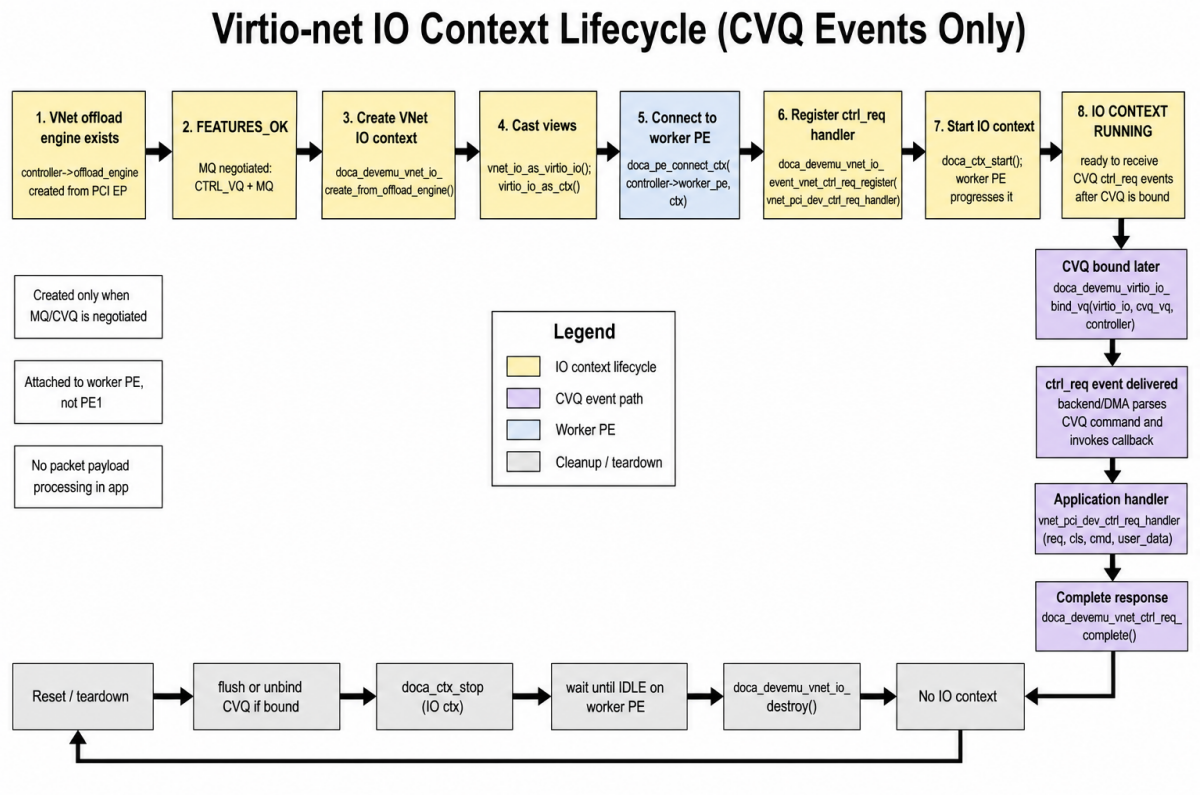
The reference app uses queue index layout:

- RX queue for queue pair `i`: `2 * i`
- TX queue for queue pair `i`: `2 * i + 1`

- Control queue: `max_queue_pairs * 2`
- Total queues: `max_queue_pairs * 2 + 1`

The control queue is created only when both `VIRTIO_NET_F_CTRL_VQ` and `VIRTIO_NET_F_MQ` are negotiated.

2.5. Virtio-net IO Context



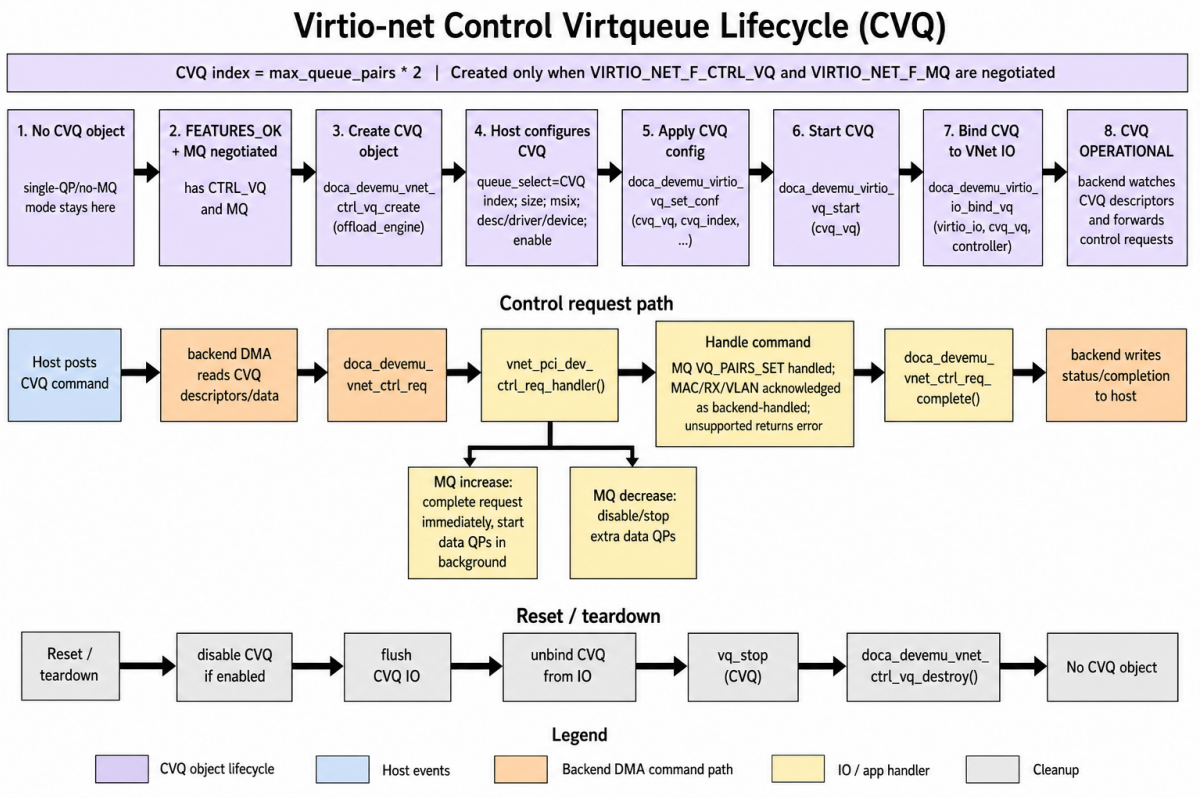
The DevEmu Virtio-net IO context routes control virtqueue requests to the application. It is not a packet datapath callback context.

Key APIs:

- `doca_devemu_vnet_io_create_from_offload_engine()`
- `doca_devemu_vnet_io_destroy()`
- `doca_devemu_vnet_io_as_ctx()`
- `doca_devemu_vnet_io_as_virtio_io()`
- `doca_devemu_vnet_io_event_vnet_ctrl_req_register()`

The reference app creates the IO context only when MQ/CVQ is active. It connects the IO context to a worker progress engine, registers `vnet_pci_dev_ctrl_req_handler()`, starts the context, and later binds the CVQ with `doca_devemu_virtio_io_bind_vq()`.

2.6. Control Request



A control request represents one VirtIO-net CVQ command.

Handler signature:

- void handler(struct doca_devemu_vnet_ctrl_req *req, uint8_t cls, uint8_t cmd, void *req_user_data);

key APIs:

- `doca_devemu_vnet_ctrl_req_get_vnet_io()`
- `doca_devemu_vnet_ctrl_req_get_vq_user_data()`
- `doca_devemu_vnet_ctrl_req_get_data()`
- `doca_devemu_vnet_ctrl_req_get_data_len()`
- `doca_devemu_vnet_ctrl_req_get_result()`
- `doca_devemu_vnet_ctrl_req_get_result_len()`
- `doca_devemu_vnet_ctrl_req_complete()`

The reference app handles `VIRTIO_NET_CTRL_MQ / VIRTIO_NET_CTRL_MQ_VQ_PAIRS_SET`. MAC, RX, and VLAN control classes are acknowledged as backend-handled in the sample.

5.4.20.4.6.3 3. Feature Bits

The reference app advertises these default network features:

- **VIRTIO_NET_F_CSUM**

- VIRTIO_NET_F_MTU
- VIRTIO_NET_F_MAC
- VIRTIO_NET_F_STATUS
- VIRTIO_NET_F_HOST_TSO4
- VIRTIO_NET_F_HOST_TSO6
- VIRTIO_NET_F_SPEED_DUPLEX

It always adds:

- VIRTIO_F_VERSION_1
- VIRTIO_F_ACCESS_PLATFORM
- VIRTIO_NET_F_CTRL_VQ
- VIRTIO_NET_F_MQ
- VIRTIO_F_ADMIN_VQ

Optional offloads features, enabled with CLI argument (see section 14):

- VIRTIO_NET_F_GUEST_CSUM
- VIRTIO_NET_F_GUEST_TSO4
- VIRTIO_NET_F_GUEST_TSO6
- VIRTIO_NET_F_GUEST_ECN
- VIRTIO_NET_F_GUEST_UFO
- VIRTIO_NET_F_HOST_ECN
- VIRTIO_NET_F_HOST_UFO
- VIRTIO_NET_F_MRG_RXBUF

Even though the app advertises control queue and multiqueue support, it still checks what the driver negotiated. If both `CTRL_VQ` and `MQ` are not negotiated, the app skips CVQ creation and IO context initialization and runs with one active queue pair.

5.4.20.4.6.4 4. Device Configuration Layout

The reference app models the VirtIO-net device config as:

```
struct vnet_virtio_net_config {
        uint8_t mac[ETH_ALEN];
        uint16_t status;
        uint16_t max_virtqueue_pairs;
        uint16_t mtu;
        uint32_t speed
        ...
};
```

The app initializes this from CLI/config values:

- MAC address
- Link status
- Maximum queue pairs
- MTU
- Speed

The common VirtIO config and queue shadow state are maintained by the application, not by the vNET library. Host writes to queue addresses, queue size, queue MSI-X vector, queue enable,

device status, and feature selectors are decoded from TLP MMIO/config accesses and reflected into the app's shadow structures.

5.4.20.4.6.5 5. Global Configuration

The current DevEmu Virtio-net public API has a smaller global configuration surface than DevEmu Virtio-blk.

Key APIs:

- `doca_devemu_vnet_add_dev()`
- `doca_devemu_vnet_rm_dev()`
- `doca_devemu_vnet_set_vnet_ctrl_req_user_data_size()`
- `doca_devemu_vnet_get_vnet_ctrl_req_user_data_size()`
- `doca_devemu_vnet_init()`
- `doca_devemu_vnet_teardown()`

Unlike DevEmu Virtio-blk, the DevEmu Virtio-net headers do not expose global `seg_max`, queue-size, or datapath-on-DPA configuration APIs.

The reference app performs:

1. Add DOCA device to DevEmu Virtio-net.
2. Initialize the DevEmu Virtio-net subsystem.
3. Parse and store MAC/MTU/queue-pair settings.
4. Create per-endpoint offload engines later.

5.4.20.4.6.6 6. Discovery and PCI Setup

A typical DevEmu Virtio-net TLP setup flow is:

1. Open a DOCA device by PCI address or IB device name.
2. Query `doca_devemu_vnet_cap_is_pci_tlp_type_supported()`.
3. Create the DevEmu Virtio-net PCI TLP type with `doca_devemu_vnet_pci_tlp_type_create()`.
4. Attach the DOCA device to the PCI type.
5. Configure MSI-X and doorbell counts.
6. Configure PCI capabilities such as MSI-X.
7. Start the PCI type.
8. Create a TLP channel with `doca_devemu_pci_tlp_channel_create()`.
9. Register the TLP request handler.
10. Connect the TLP channel context to the main progress engine.
11. Start the TLP channel context.
12. Create endpoint representors and TLP devices.

The reference app uses a main progress engine for TLP handling and one worker progress engine per controller for slower DOCA operations.

5.4.20.4.6.7 7. Initialization Flow

7.1. Application Startup

At process startup, the app:

1. Allocates `tlp_context`.
2. Opens the DOCA device.

3. Creates the main progress engine.
4. Initializes the DevEmu Virtio-net PCI framework and workqueue.
5. Creates and starts the vNET PCI TLP type.
6. Creates and starts the TLP channel.
7. Connects the TLP channel to the main PE.
8. Initializes software topology.
9. Calls DevEmu Virtio-net global init.
10. Creates the software VirtIO-net device model.
11. In static mode, creates all endpoint representors and TLP devices.
12. Creates one DevEmu Virtio-net controller per endpoint.
13. Creates one worker PE per controller.
14. Enters the progress loop.

7.2. Per-endpoint Controller Creation

For each endpoint, the app:

1. Creates or reattaches a PCI representor.
2. Creates a TLP device.
3. Starts the TLP device.
4. Creates a vNET offload engine from the PCI endpoint.
5. Sets MTU.
6. Sets MAC address.
7. Sets total number of queues on the common VirtIO offload engine.
8. Allocates RX/TX VQ pointer arrays.
9. Queries the associated representor for logging/debug.

The offload engine is created before host driver probing, but not started until `FEATURES_OK`.

7.3. FEATURES_OK

When the host writes `FEATURES_OK`, the app queues asynchronous work:

1. Determine whether `CTRL_VQ` and `MQ` were both negotiated.
2. Set `num_active_qps` to `1`.
3. Set `max_queue_pairs` to the configured maximum if MQ is active, otherwise `1`.
4. Apply optional features if negotiated.
5. Start the offload engine.
6. Create RX/TX VQs for all configured queue pairs.
7. Create CVQ only if MQ/CVQ was negotiated.
8. Create and start the IO context only if CVQ is needed.

This work is queued to avoid blocking TLP completions on slow DOCA or firmware operations.

7.4. DRIVER_OK

When the host writes `DRIVER_OK`, the app:

1. Verifies the first queue pair is fully configured.
2. Applies queue configuration with `doca_devemu_virtio_vq_set_conf()`.
3. Starts active RX/TX VQs with `doca_devemu_virtio_vq_start()`.
4. Starts the CVQ if configured.
5. Binds the CVQ to the IO context.
6. Enables the offload engine with `doca_devemu_virtio_offload_engine_enable()`.
7. Creates DevEmu Virtio-net counters for debug/statistics.

At this point the device is operational.

5.4.20.4.6.8 8. Execution Phase

8.1. Control Path

The control path includes:

- TLP request handling.
- PCI config and MMIO emulation.
- VirtIO feature negotiation.
- Device status transitions.
- Queue configuration writes.
- Queue lifecycle.
- CVQ command handling.
- Config-change MSI-X notification.
- Reset and hotplug.

The app keeps TLP handling latency-sensitive. Expensive work is moved to a workqueue and worker PEs. This is essential because slow endpoint creation or large multiqueue startup can otherwise cause PCIe completion timeouts or host stalls.

8.2. Data Path

The data path is the most important conceptual difference from DevEmu Virtio-blk.

For DevEmu Virtio-net, the app does not receive every packet. RX/TX virtqueues are configured and then owned by the DevEmu Virtio-net offload engine. Once queues are started and the engine is enabled, packet movement is handled by the DevEmu Virtio-net offload path.

The application still owns:

- Virtqueue creation.
- Virtqueue configuration from host-provided ring addresses.
- Virtqueue start/stop/enable/disable.
- Multiqueue control.
- Debug counters and virtqueue stats.
- Representor and integration logic.

The app handles CVQ commands through `doca_devemu_vnet_ctrl_req`, not packet payloads.

8.3. Control Virtqueue Path

The CVQ path is:

1. Host submits a VirtIO-net control command.
2. DevEmu Virtio-net library receives the CVQ command.
3. The registered IO handler is invoked with `req`, `cls`, and `cmd`.
4. The app reads command-specific data using `doca_devemu_vnet_ctrl_req_get_data()`.
5. The app optionally writes command-specific result data using `doca_devemu_vnet_ctrl_req_get_result()`.
6. The app completes with `doca_devemu_vnet_ctrl_req_complete(req, ack, len)`.

For `VIRTIO_NET_CTRL_MQ_VQ_PAIRS_SET`, the app reads the requested queue-pair count and validates it.

If the request decreases active queue pairs, extra queues are disabled/stopped synchronously.

If the request increases active queue pairs, the app completes the CVQ request immediately with `VIRTIO_NET_OK` and starts additional queues asynchronously. This avoids blocking the host in `virtnet_send_command()` while many queues are started.

5.4.20.4.6.9 9. Reset Flow

When the host writes `device_status = 0`, the app treats it as a VirtIO device reset.

The reset path:

1. Queues controller cleanup asynchronously.
2. Keeps the offload engine object alive for normal driver rebind.
3. Disables VQs if the engine is enabled.
4. Disables the engine.
5. Stops VQs.
6. Flushes and unbinds CVQ if needed.
7. Stops and destroys the IO context.
8. Destroys VQs.
9. Stops the engine.
10. Leaves the engine object available unless this is full teardown.

For hot-unplug, the app intentionally does not destroy the engine on the reset edge alone. It waits for the later PCIe power-off / delayed-destroy path so that a reset during driver probe is not misclassified as final removal.

5.4.20.4.6.10 10. Teardown Flow

Full teardown performs stricter cleanup than reset:

1. Shut down the PCI config workqueue.
2. Stop stats and counter collection.
3. Destroy DevEmu Virtio-net counters.
4. Destroy queue debug state lists.
5. Destroy config MSI-X object if present.
6. Disable VQs.
7. Disable the offload engine.
8. Stop VQs.
9. Flush and unbind CVQ.
10. Stop and destroy IO context.
11. Destroy RX/TX/CVQ objects.
12. Stop the offload engine.
13. Destroy the DevEmu Virtio-net offload engine.
14. Stop and destroy TLP devices.
15. Destroy or close representors.
16. Stop and destroy the PCI type.
17. Tear down the DevEmu Virtio-net subsystem.
18. Remove the DOCA device from DevEmu Virtio-net.
19. Destroy progress engines and close the DOCA device.

5.4.20.4.6.11 11. Hotplug

In hotplug mode, endpoints are created and destroyed dynamically. The app accepts runtime commands such as plug and unplug. Hotplug work is performed on the worker path rather than directly on the TLP progress path.

For plug:

1. Create representor.
2. Create and start TLP device.
3. Create DevEmu Virtio-net controller/offload engine.
4. Update slot state.
5. Notify the host.

For unplug:

1. Mark unplug pending.
2. Signal hotplug state to the host.
3. Keep the device alive while the host resets/removes the driver.
4. Destroy the endpoint only after the host reaches the expected power-off/removal sequence.

5.4.20.4.6.12 12. Counters and Debug State

The DevEmu Virtio-net library exposes counters through `doca_devemu_vnet_counters`.

The reference app creates counters after the engine is enabled and periodically collects:

- RX virtqueue counters.
- TX virtqueue counters.
- Common VirtIO queue debug state.

The app also keeps a per-controller debug-state collection state . The queue debug-state data itself is stored in a `doca_devemu_virtio_queue_dbg_state` list. Each populated entry describes one VirtIO queue, including

the queue ID, enabled state, queue size, in-flight request count, hardware available/used indexes, and driver available/used indexes.

The app carefully stops collection before teardown to avoid destroying resources still in use by asynchronous population.

5.4.20.4.6.13 13. Live Update

The DevEmu Virtio-net offload engine supports export/import:

- `doca_devemu_vnet_offload_engine_export()`
- `doca_devemu_vnet_offload_engine_create_from_export()`

The reference VNet PCI application uses these APIs in '--lu-mode active' / '--lu-mode standby'. During handover, the active process exports the offload engine state and the standby process imports it, then starts the restored engine

and replays the required VirtIO queue state.The imported engine preserves runtime state such as MTU, MAC, queue count, and queue configuration.

The standby path skips normal TLP channel setup initially and restores channel/offload state through the live-update sequence.

5.4.20.4.6.14 14. Optional features

Some VirtIO features needs to be enabled explicitly on the offload engine using the optional features API.

(those are usually datapath related with no control functionality)

Optional features are configured in two stages:

1. verify that the device supports configuring the requested feature
2. enable or disable the feature on the offload engine before it is started.

Capability discovery:

Use **doca_devemu_vnet_cap_is_offload_engine_feature_supported()** with the VirtIO feature bit index - this tells if that feature is supported by this device.

(The application should advertise a feature to the host only when the device supports it and the application is prepared to honor the negotiated behavior)

Reading and changing the engine configuration:

Use **doca_devemu_virtio_offload_engine_get_feature_enabled()** to read whether a feature is currently enabled for a specific offload engine.

Use **doca_devemu_virtio_offload_engine_set_feature_enabled()** to enable a feature on a specific offload engine - must be applied while the offload engine before it starts.

5.4.20.4.6.15 15. Reference App Mental Model

A user reading `vnet_pci_dev` should view it as three cooperating subsystems:

`PCI/TLP emulator`: decodes host config/MMIO TLPs and returns completions quickly.

`VirtIO-net controller`: tracks features, device status, queue config, reset, hotplug, and CVQ commands.

`DevEmu Virtio-net offload integration`: creates offload engines and queues, enables packet datapath, exposes representor and counters.

The most important design rule in the reference app is that the main TLP progress path must stay responsive. Virtqueue startup, hotplug, reset cleanup, stats, and multiqueue scaling are deliberately moved away from the main PE to avoid host-visible PCIe timeouts.

5.4.20.4.7 DOCA DevEmu Virtio-blk

> ***Note***
> ***This library is supported at alpha level; backward compatibility is not guaranteed.***

5.4.20.4.7.1 Introduction

The DOCA TLP DevEmu Virtio-blk library, part of the DOCA DevEmu Virtio subsystem, provides low-level software APIs. These APIs serve as the building blocks for developing and manipulating

virtio-blk (virtio block) devices by leveraging the device emulation capabilities of NVIDIA® BlueField® DPUs.

Architecture and Library Interaction

This library is part of a modular subsystem, where each library manages its own logic:

- DOCA DevEmu PCI: Manages the PCIe transport layer, including configuration, discovery, and features like MSI-X and device hot plug/unplug.
- DOCA DevEmu Virtio Common: Manages generic virtio logic, such as handling common virtio registers, the virtio offload engine, and virtqueue lifecycle. Note: for TLP-based virtio-blk devices, the virtio device reset flow is handled by the application.
- DOCA DevEmu Virtio-blk: Manages the specific logic for the virtio block device, including block I/O request parsing, scatter-gather segment handling, and block device configuration.

This layered design allows for seamless integration and independent operation of each component.

Core Functionality

The DOCA DevEmu Virtio-blk library efficiently handles virtio descriptors that carry block I/O requests from the device driver. It translates these into abstract virtio-blk requests (`doca_devemu_vblk_req`) that are then routed to the user application via registered event handlers. Each request carries the block operation type (read, write, get ID, etc.), the target sector (LBA), and a DOCA buffer representing the host memory data region. This process abstracts away the underlying device-specific acceleration details.

The library supports two data path providers:

- DPA (Data Path Accelerator) — Offloads virtqueue processing to BlueField's DPA for hardware-accelerated, low-latency operation. Recommended for production workloads.
- DPU (ARM cores) — Processes virtqueues on the DPU ARM cores for maximum flexibility and debugging.

User Responsibilities

Users of this library are responsible for developing a virtio-blk controller. This controller application must:

1. Manage the underlying DOCA DevEmu Virtio-blk offload engine and IO contexts.
2. Receive the abstracted DOCA Virtio-blk requests via registered event handlers.
3. Process these requests according to the virtio-blk specification (read, write, get device ID, etc.).
4. Perform DMA data transfers between host and DPU memory as needed.
5. Complete each request back to the IO context with the appropriate status code.

> ***Note***
> ***The backend block storage implementation is outside the scope of DOCA. The reference application (`doca_vblk_pci_dev`) demonstrates a RAM-backed block device for illustration.***

5.4.20.4.7.2 Prerequisites

Virtio-blk device emulation is part of the DOCA DevEmu Virtio subsystem. It is, therefore, recommended to read the following guides before proceeding:

- DOCA Device Emulation

- DOCA DevEmu PCI TLP
- DOCA DevEmu Virtio (TODO: link pending)

5.4.20.4.7.3 Environment

DOCA DevEmu Virtio-blk is supported on the BlueField target only. The BlueField must meet the following requirements:

- DOCA version 3.4.0 or greater
- BlueField-3 firmware 32.49.0288 or higher

> ***Info***
> ***Please refer to the DOCA Compatibility Policy.***
> ***Note***
> ***Library must be run with root privileges.***

Perform the following:

1. Configure BlueField to work in DPU mode as described in BlueField Modes of Operation.
2. Enable TLP emulation by running the following on the DPU:

```
[dpu] mlxconfig -y -d /dev/mst/mt41692_pciconf0 set TLP_EMULATION_ENABLE=1
[dpu] mlxconfig -y -d /dev/mst/mt41692_pciconf0 set TLP_EMULATION_NUM_PF=32
```

3. Perform a BlueField system cold reboot for the `mlxconfig` settings to take effect.

Hot-plug

Host Configuration

With a Linux environment on the host OS, additional kernel boot parameters are required to support the hot-plug feature:

- For Intel machines:

intel_iommu=on iommu=pt pci=realloc

- For AMD machines:

iommu=pt pci=realloc

> ***Note***
> ***On AMD machines, hotplug may not work.***

Firmware Configuration

When PCIe switch emulation is enabled, BlueField can support hot-plugged virtio-blk functions. These PCIe functions are shared among all BlueField users and applications and may hold hot-plugged devices of type NVMe, virtio-blk, virtio-fs, or other (e.g., virtio-net).

To enable PCIe switch emulation and configure hot-plugged ports, run:

```
[dpu] mlxconfig -y -d /dev/mst/mt41692_pciconf0 set
PCI_SWITCH_EMULATION_ENABLE=1
[dpu] mlxconfig -y -d /dev/mst/mt41692_pciconf0 set
PCI_SWITCH_EMULATION_NUM_TLP_PORT=1
[dpu] mlxconfig -y -d /dev/mst/mt41692_pciconf0 set
PCI_SWITCH_EMULATION_NUM_PORT=2
```

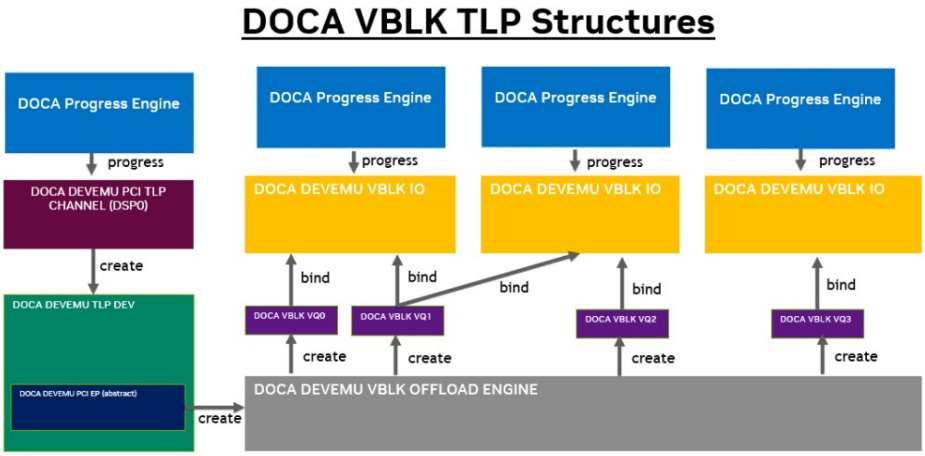
`PCI_SWITCH_EMULATION_NUM_PORT` equals 1 plus the number of hot-plugged PCIe functions.

5.4.20.4.7.4 Architecture

The DOCA DevEmu Virtio-blk library provides the following main software abstractions:

- The **virtio-blk type** — extends the PCIe TLP type; creates a PCI type specifically configured for virtio-blk device emulation
- The **virtio-blk offload engine** — extends the virtio offload engine; manages virtqueue lifecycle and block-specific configuration (e.g., `seg_max`)
- The **virtio-blk IO context** — extends the virtio IO context; represents a progress context responsible for processing virtio descriptors carrying block I/O requests
- The **virtio-blk request** — represents a single block I/O operation from the host driver

The following diagram illustrates the relationships between these abstractions:



Virtio-blk Feature Bits

The following virtio feature bits are relevant to virtio-blk emulation:

Feature Bit	Description
`VIRTIO_BLK_F_SIZE_MAX` (bit 1)	Maximum segment size is reported
`VIRTIO_BLK_F_SEG_MAX` (bit 2)	Maximum segments per request is reported
`VIRTIO_BLK_F_MQ` (bit 12)	Multi-queue support
`VIRTIO_F_INDIRECT_DESC` (bit 28)	Indirect descriptor support
`VIRTIO_F_VERSION_1` (bit 32)	VirtIO 1.0 compliance
`VIRTIO_F_ACCESS_PLATFORM` (bit 33)	Device access restricted to platform IOMMU

For TLP-based virtio-blk devices, feature bits are configured directly by the application in the emulated PCI configuration space. Indirect descriptor support must first be verified via `doca_devemu_vblk_cap_is_indir_descs_supported()` before enabling.

Virtio-blk Configuration Layout

According to the virtio specification, the virtio-blk device configuration structure layout includes the following key fields:

```
struct virtio_blk_config {
    le64 capacity;       /* device capacity in 512-byte sectors */
    le32 size_max;       /* maximum size of any single segment */
    le32 seg_max;        /* maximum number of segments in a request
*/
    /* ... additional fields ... */
    le16 num_queues;     /* number of request queues */
    /* ... */
};
```

For TLP-based virtio-blk devices, these fields are managed directly by the application in the emulated PCI configuration space. The `seg_max`, and `num_queues` values must also be configured on the offload engine (via `doca_devemu_vblk_offload_engine_set_seg_max()`and `doca_devemu_virtio_offload_engine_set_num_queues()` respectively) so the library can allocate internal resources accordingly.

Virtio-blk Type

The virtio-blk type creates a PCI TLP type specifically configured for VirtIO Block device emulation, providing full control over the PCIe configuration space.

To create and configure a virtio-blk type:

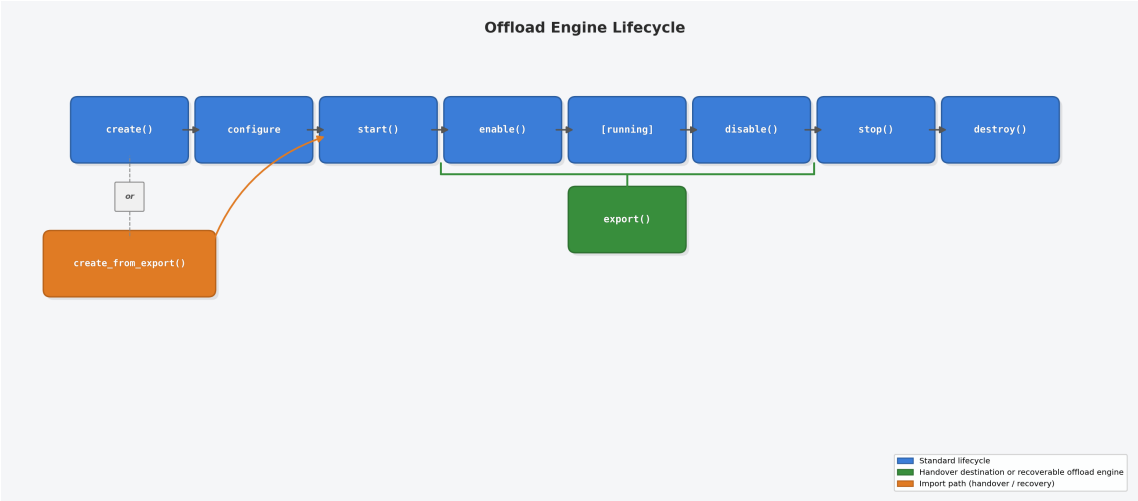
1. `doca_devemu_vblk_is_pci_tlp_type_supported()` — Check if the device supports VirtIO Block TLP type management.
2. `doca_devemu_vblk_pci_tlp_type_create()` — Create a stopped PCI TLP type for VirtIO Block emulation.
3. Configure the type using the following setters:
    - `doca_devemu_pci_type_set_dev()` — Associate a DOCA device with the type.
    - `doca_devemu_pci_tlp_type_set_pci_cap_conf()` — Register PCI capability configurations (e.g., PCI Express, MSI-X).
    - `doca_devemu_pci_type_set_num_msix()` — Set the number of MSI-X vectors.
    - `doca_devemu_pci_type_set_num_db()` — Set the number of doorbells.
4. `doca_devemu_pci_type_start()` — Start the PCI type to finalize configuration.

Virtio-blk Offload Engine

The virtio-blk offload engine (`doca_devemu_vblk_offload_engine`) extends the virtio offload engine with block-specific configuration. It is responsible for managing the state of all offloaded entities on a device, including configuration, creation, destruction, start, and stop of virtio-blk request virtqueues. Each offload engine is associated with a single DOCA DevEmu PCI endpoint.

Offload Engine Lifecycle

The offload engine follows this lifecycle. Configuration (e.g. **`seg_max`**, indirect descriptors) happens between **`create()`** and **`start()`**. In handover and recovery scenarios, **`create_from_export()`** replaces both **`create()`** and the configuration phase — configuration values are inherited from the export descriptor:



Standard Configuration

The standard flow for creating and configuring a virtio-blk offload engine is as follows:

1. `doca_devemu_vblk_offload_engine_create()` — Create an offload engine bound to a PCI endpoint.
2. `doca_devemu_vblk_offload_engine_set_seg_max()` — Set the maximum scatter-gather segments per request.
3. Cast to virtio offload engine
   via `doca_devemu_vblk_offload_engine_as_virtio_offload()` for common configuration:
   - `doca_devemu_virtio_offload_engine_set_indir_descs_enabled()` — Enable/disable indirect descriptor processing.
   - `doca_devemu_virtio_offload_engine_set_num_queues()` — Set the number of virtqueues.
4. `doca_devemu_virtio_offload_engine_start()` — Start the offload engine.
5. `doca_devemu_virtio_offload_engine_enable()` — Enable the offload engine to begin processing requests.

Optional Configurations

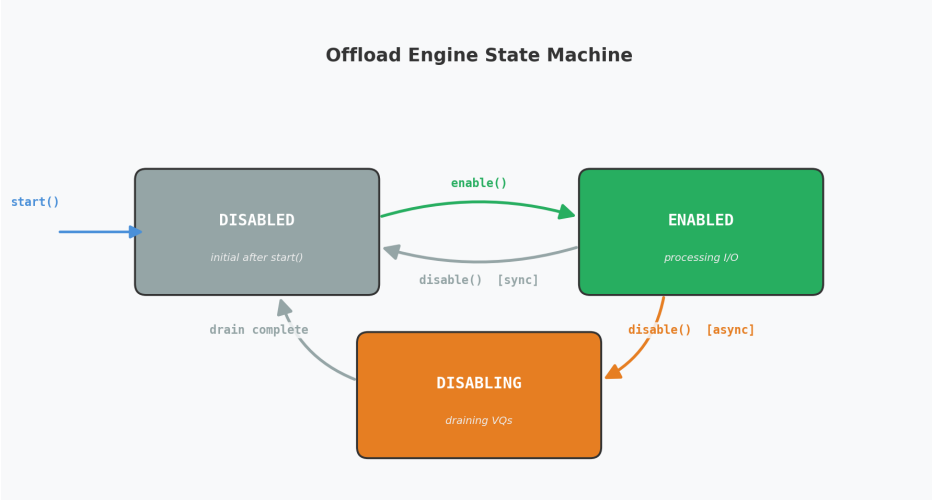
- `doca_devemu_vblk_offload_engine_set_seg_max()` — Set the `seg_max` value for the block device. If not set, the value defaults to the global `seg_max` configured
  via `doca_devemu_vblk_set_seg_max()`.
- `doca_devemu_virtio_offload_engine_set_indir_descs_enabled()` — Enable indirect descriptor support. Only valid
  if `doca_devemu_vblk_cap_is_indir_descs_supported()` returns true.

Export/Import Overlay

Function	When to Call	Effect
`export()`	After `start()`, before any VQ is started.	Produces an opaque export descriptor containing the runtime state of the offload engine. The caller owns the descriptor and must free it with `free()`.
`create_from_export()`	Instead of `create()`.	Creates a new idle offload engine seeded with the state from an export descriptor. The new engine must still be started and enabled.

Offload Engine States

A started offload engine is in one of three states:



State	Description
`DISABLED`	Initial state after `start()`. No virtqueue operations are processed. Virtqueues cannot be individually enabled or disabled.
`ENABLED`	Processing operations from the host driver. Virtqueues can be individually enabled/disabled.
`DISABLING`	Transitioning to `DISABLED`. At least one virtqueue is still draining in-flight operations. Poll with `get_state()` until `DISABLED`.

Enable/Disable Semantics

- `enable()` moves all started-and-disabled virtqueues to the enabled state. If the engine is already enabled, calling `enable()` again enables any newly started virtqueues that are currently disabled.
- `disable()` may return `DOCA_SUCCESS` (synchronous) or `DOCA_ERROR_IN_PROGRESS` (asynchronous, at least one VQ is still draining). In the async case, poll with `get_state()` until the engine reaches `DISABLED`.

Handover Constraint

During a handover, for each source/destination pair of offload engines sharing the same PCI endpoint, at most one engine may be enabled at any time. Both engines may be disabled simultaneously (between the source's `disable()` and the destination's `enable()`), but having both enabled leads to undefined behavior.

State Export/Import (Live Upgrade and Recovery)

The DOCA DevEmu Virtio Block SDK provides a state export/import mechanism that enables two resilience scenarios for an offload engine managing a Virtio Block device:

Scenario	Description
Handover (Live Upgrade)	A running source process exports the offload engine state and transfers it to a destination process. The destination imports the state, and the two processes coordinate so that exactly one engine is enabled at any time. In-flight I/O during handover is not supported.
Takeover (Recovery)	A running process exports the offload engine state and stores the export descriptor in a durable location. If the process crashes, a recovering process imports the descriptor to recreate the offload engine and replay in-flight I/O.

Both scenarios use the same three API calls:

```
doca_devemu_vblk_offload_engine_export()
doca_devemu_vblk_offload_engine_create_from_export()
doca_devemu_vblk_offload_engine_export_release()
```

API Reference

API	Description	Preconditions	Postconditions
`doca_devemu_vblk_offload_engine_export()`	Exports the runtime state of a started offload engine into an opaque descriptor blob.	• The offload engine must be started. • No virtqueue associated with the engine may be started. • Must not be called concurrently.	• `export_desc` points to an opaque blob owned by the caller. • The caller must free the descriptor buffer with `free()` when it is no longer needed.
`doca_devemu_vblk_offload_engine_create_from_export()`	Creates a new offload engine from an export descriptor. This is the import-side counterpart of `export()`. The created engine reflects the same runtime state as the exported engine once it is started and enabled.	• `export_desc` must have been produced by `doca_devemu_vblk_offload_engine_export()`.	• The returned offload engine is idle (not started). It must be started and enabled to begin processing. • Configuration values (e.g. `seg_max`, indirect descriptors) are inherited from the export descriptor.

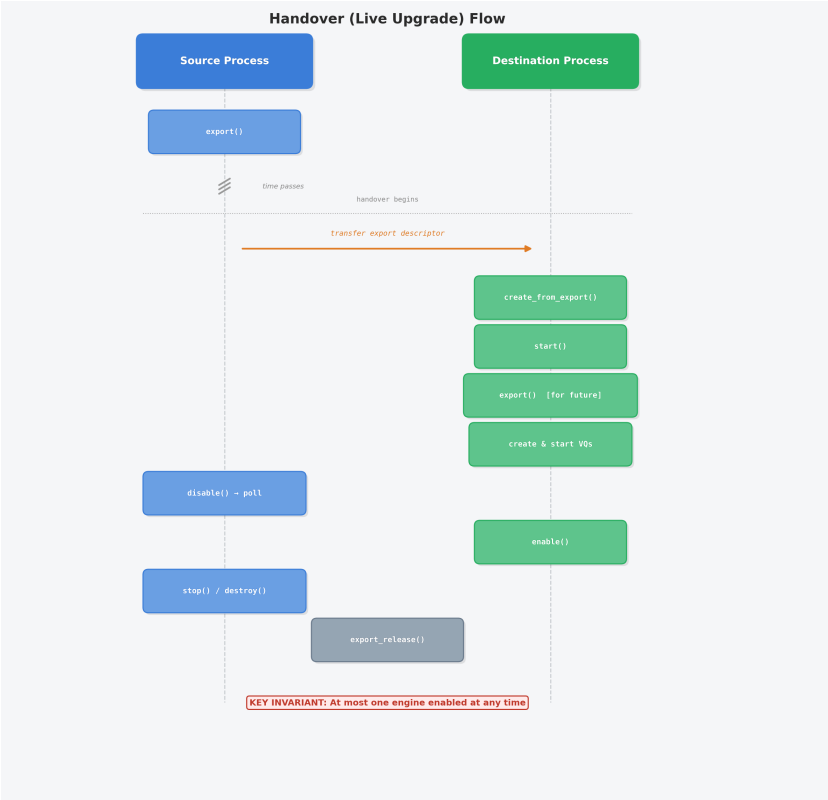
API	Description	Preconditions	Postconditions
`doca_devemu_vblk_offload_engine_export_release()`	Releases engine-side resources associated with the export. Must be called exactly once per export. Does not free the descriptor buffer — the caller is responsible for freeing it with `free()`. Double release is invalid and leads to undefined behavior.	• `export_desc` must have been returned by `export()` and not yet released.	N/A

Handover (Live Upgrade) Flow

In a handover scenario, a source process transfers ownership of an offload engine to a destination process while maintaining host-visible state. The SDK does not prescribe any particular IPC mechanism or process architecture - the coordination protocol is application-defined.

Sequence

1. Source has a valid export descriptor. The `export()` call is typically done during regular bring-up of the application, not necessarily at the time of handover.
2. Source transfers the export descriptor to the destination through an application-defined channel.
3. Destination calls `create_from_export()` with the received descriptor and a PCI endpoint handle. This creates an idle offload engine seeded with the source's state.
4. Destination calls `start()` on the new engine. Configuration is inherited from the descriptor. The engine enters the `DISABLED` state.
5. Destination calls `export()` on its own engine to prepare for future recovery or live update. Virtqueues must not be started before this call.
6. Destination creates, configures, and starts its virtqueues.
7. Source calls `disable()` on its engine. If `DOCA_ERROR_IN_PROGRESS` is returned, poll `get_state()` until `DISABLED`.
8. Once the source is fully disabled, destination calls `enable()` on its engine. The engine and its virtqueues must be enabled to start processing I/Os from the host driver.
9. Source calls `stop()` and `destroy()` to release its engine.
10. `export_release()` is called on the source's descriptor.



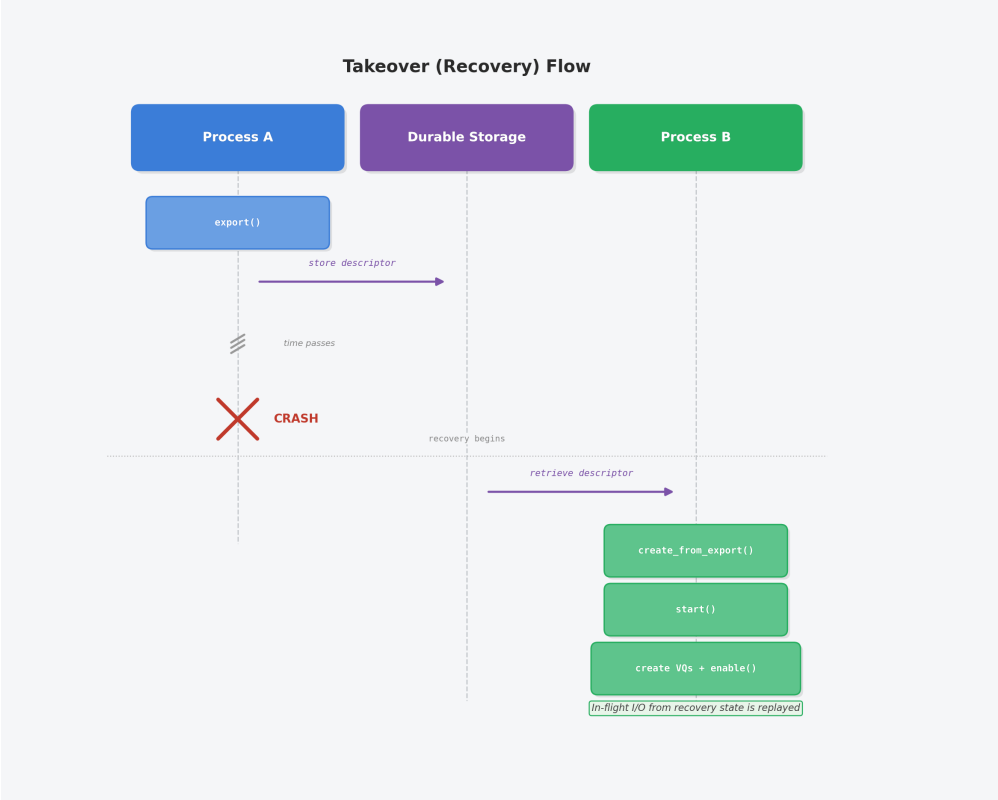
> ***Key invariant: At no point during the handover should both engines be enabled simultaneously.***

Takeover (Recovery) Flow

In a takeover scenario, a process exports its offload engine state proactively so that a recovering process can restore the engine after a crash.

Sequence

1. Process A calls **`export()`** on its started offload engine and stores the export descriptor in a durable location (e.g. shared memory, file, or another process).
2. Process A crashes.
3. Process B (the recovering process) retrieves the stored export descriptor.
4. Process B calls **`create_from_export()`** with the descriptor and a PCI endpoint handle. This creates an idle offload engine seeded with the crashed process's state.
5. Process B calls **`start()`** on the engine.
6. Process B creates, configures, and starts its virtqueues, then calls **`enable()`** on the engine. The engine and its virtqueues must be enabled to start processing I/Os. In-flight I/O from the recovery state is replayed.



> ***Limitation: Process B must configure the VirtIO Block global parameters (e.g. `doca_devemu_vblk_set_*()`) with the same values that were used by Process A before takeover. These values are not captured in the export descriptor.***

Request Virtqueue Management

Each VirtIO Block virtqueue is created and managed through the offload engine:

API	Description
`doca_devemu_vblk_req_vq_create()`	Create a VirtIO Block request virtqueue from a started offload engine. Must be called on the offload engine thread.
`doca_devemu_vblk_req_vq_destroy()`	Destroy a VirtIO Block request virtqueue. Must not be started.
`doca_devemu_vblk_req_vq_as_vq()`	Cast to base virtio virtqueue for common operations (`set_conf`, `start`, `stop`, `enable`, `disable`).

Virtio-blk IO Context

The virtio-blk IO context (`doca_devemu_vblk_io`) extends the virtio IO context. It is responsible for routing incoming block I/O requests from the host driver towards the application, and relaying completions back to the driver. Each IO context is associated with a single offload engine and is progressed by a single DOCA Core progress engine. Usually, users configure a single virtio-blk IO context per BlueField ARM core used by the application.

The IO context can only route requests while in `running` state and when its associated offload engine is enabled.

Virtio-blk IO Configurations

The flow for creating and configuring a virtio-blk IO context is as follows:

1. `doca_devemu_vblk_io_create_from_offload_engine()` — Create a new IO context from a started offload engine. Must be called on the ARM core that will manage the IO context.
2. `doca_devemu_vblk_io_event_vblk_req_register()` — Register the event handler for incoming VirtIO Block requests.
3. Cast to DOCA context via `doca_devemu_vblk_io_as_ctx()` for progress engine connection:
    - `doca_pe_connect_ctx()` — Connect the IO context to a progress engine.
    - `doca_ctx_start()` — Start the IO context.

Mandatory Configurations

- `doca_devemu_vblk_io_event_vblk_req_register()` — Register the event handler for incoming VirtIO Block requests. This is mandatory.

Virtio-blk Request

The virtio-blk request object (`doca_devemu_vblk_req`) serves as an abstraction for handling block I/O requests arriving on virtio-blk request queues. These requests are generated by the device driver through virtio queues and routed to the user via the registered event handler on the associated IO context.

Once the event handler is called, ownership of the virtio-blk request and the associated request user data transfers to the user. The ownership moves back to the associated IO context when the request is completed by calling `doca_devemu_vblk_req_complete()`.

The event handler provides the following parameters directly:

```
void handler(struct doca_devemu_vblk_req *req,
             uint32_t type,        /* VIRTIO_BLK_T_IN,
VIRTIO_BLK_T_OUT, etc. */
             uint64_t sector,      /* starting sector (LBA) */
             void *req_user_data); /* per-request user data buffer */
```

The following APIs operate on a virtio-blk request:

API	Description
`doca_devemu_vblk_req_get_data()`	Get a DOCA buffer representing the `data[]` field of the virtio-blk request. This buffer represents the host memory for the data portion of the request according to the virtio specification.
`doca_devemu_vblk_req_get_data_len()`	Get the total data length (in bytes) of the data buffer linked list.
`doca_devemu_vblk_req_get_data_list_len()`	Get the number of scatter-gather elements in the data buffer linked list.
`doca_devemu_vblk_req_get_vblk_io()`	Get the associated DOCA Virtio Block IO context.
`doca_devemu_vblk_req_get_vq_user_data()`	Get the user data associated with the virtqueue binding. Must be called on the IO context thread.

API	Description
`doca_devemu_vblk_req_complete()`	Complete the request with a status value and the number of bytes written into the device-writable portion of the buffer. The status follows the virtio specification: `VIRTIO_BLK_S_OK` (0), `VIRTIO_BLK_S_IOERR` (1), `VIRTIO_BLK_S_UNSUPP` (2).

5.4.20.4.7.5 Global Configuration and Capability Queries

Before initializing the VirtIO Block subsystem, the application must configure global parameters and may query device capabilities.

Global Configuration

The following APIs configure the VirtIO Block subsystem before `doca_devemu_vblk_init()` is called. For the setter APIs (`set_vblk_req_user_data_size`, `set_max_queue_size`, `set_seg_max`, `set_datapath_on_dpa`), if called multiple times, only the last call before `doca_devemu_vblk_init()` takes effect.

API	Description
`doca_devemu_vblk_add_dev()`	Register a DOCA device for VirtIO Block management. The device must support VirtIO Block type management.
`doca_devemu_vblk_rm_dev()`	Unregister a previously added DOCA device.
`doca_devemu_vblk_set_vblk_req_user_data_size()`	Set the size of the per-request user data buffer allocated for each `doca_devemu_vblk_req`.
`doca_devemu_vblk_set_max_queue_size()`	Set the maximum virtqueue depth. Must conform to `doca_devemu_vblk_cap_get_max_queue_size()`.
`doca_devemu_vblk_set_seg_max()`	Set the global maximum scatter-gather segments per request. Must conform to `doca_devemu_vblk_cap_get_max_seg_max()`.
`doca_devemu_vblk_set_datapath_on_dpa()`	Set to `true` to run the data path on DPA, `false` for DPU ARM cores.
`doca_devemu_vblk_init()`	Initialize the VirtIO Block subsystem. One-time call after all configuration is complete.
`doca_devemu_vblk_teardown()`	Tear down the VirtIO Block subsystem and release all global resources.

Capability Queries

API	Description
`doca_devemu_vblk_cap_get_max_queue_size()`	Query the maximum supported virtqueue depth for the device.
`doca_devemu_vblk_cap_get_max_seg_max()`	Query the maximum supported `seg_max` value for the device.
`doca_devemu_vblk_cap_is_indir_descs_supported()`	Check whether the device supports indirect descriptor processing.

5.4.20.4.7.6 Discovery

The virtio-blk library uses the TLP emulation path, creating its own PCI type and representors dynamically.

To find a suitable device and prepare for virtio-blk emulation, users should perform the following:

1. Open a DOCA device that supports VirtIO Block TLP type management. This can be done by:
    - Iterating `doca_devinfo_create_list()` and checking `doca_devemu_vblk_is_pci_tlp_type_supported()` for each device, or
    - Opening a known device directly by IB device name (e.g., `mlx5_0`) via helper utilities such as `open_doca_device_with_ibdev_name()`.
2. `doca_devemu_vblk_pci_tlp_type_create()` — Create the VirtIO Block PCI TLP type.
3. Configure and start the PCI type (`doca_devemu_pci_type_set_dev()`, `doca_devemu_pci_type_start()`, etc.).
4. `doca_devemu_pci_type_create_rep()` — Create representors for each endpoint.
5. `doca_devemu_pci_tlp_dev_create()` — Create PCI TLP devices from the representors.

At this point, the user can proceed with the initialization flow to create offload engines and IO contexts.

5.4.20.4.7.7 Initialization

This section describes the initialization flow of a DOCA DevEmu Virtio-blk offload engine and one or more IO contexts. During initialization, the user sets up and prepares the environment before starting to receive block I/O requests from the host.

The user should perform the following:

1. Configure global VirtIO Block parameters (`add_dev`, `set_seg_max`, `set_datapath_on_dpa`, etc.) and call `doca_devemu_vblk_init()`.
2. Choose N ARM cores to run the application IO threads on.
3. Create N DOCA Core progress engine (PE) objects.
4. Create the VirtIO Block PCI TLP type via `doca_devemu_vblk_pci_tlp_type_create()`, configure capabilities, and start it.
5. Create a PCI TLP device for the endpoint.
6. Configure global offload enigne parameters.
7. Create and configure the VirtIO Block offload engine:
    - If offload engine doesn't perform state import:
        - `doca_devemu_vblk_offload_engine_create()` with the PCI endpoint.
        - Configure `seg_max`, `num_queues`, and optionally indirect descriptors.
        - `doca_devemu_virtio_offload_engine_start()`.
        - Optionally, call **`doca_vblk_pci_export()`** for future handover/export (see State Export/Import (Live Upgrade and Recovery)).
    - If state import:
        - **`doca_devemu_vblk_offload_engine_create_from_export()`** for future export, passing the export descriptor and its length along with the PCI endpoint.
        - If offload engine is a handover destination, call **`doca_devemu_vblk_offload_engine_export()`**.

- `doca_devemu_virtio_offload_engine_start()`.

8. Create N VirtIO Block IO contexts:
   - `doca_devemu_vblk_io_create_from_offload_engine()` on each ARM core.
   - Register the block request event handler.
   - Connect to the progress engine and start.
9. Enable the offload engine via `doca_devemu_virtio_offload_engine_enable()`.

At this point, the offload engine and IO contexts are fully operational and ready to process block I/O requests from the host.

> ***Note***
> ***During the initialization flow, no block I/O requests are routed to the user until the offload engine is enabled.***

5.4.20.4.7.8 Teardown

This section describes the teardown flow of a DOCA DevEmu Virtio-blk offload engine and its associated IO contexts.

The user should perform the following:

1. For each bound virtqueue, perform the per-VQ teardown sequence:
   a. `doca_devemu_virtio_vq_disable()` — Disable the virtqueue.
   b. If the call returns `DOCA_ERROR_IN_PROGRESS` (virtqueue is draining in-flight requests), optionally call `doca_devemu_virtio_io_flush_vq()` to flush pending events to the application.
   c. Handle and complete any outstanding virtio-blk requests via `doca_devemu_vblk_req_complete()`.
   d. Loop: call `doca_pe_progress()` on the IO context's progress engine until the virtqueue reaches `DISABLED` state (poll via `doca_devemu_virtio_vq_get_state()`).
   e. `doca_devemu_virtio_io_unbind_vq()` — Unbind the virtqueue from the IO context.
2. For each VirtIO Block IO context:
   - Stop the IO context via `doca_ctx_stop()` and wait for `idle` state.
   - Destroy the IO context via `doca_devemu_vblk_io_destroy()`.
3. Disable the offload engine via `doca_devemu_virtio_offload_engine_disable()` after all virtqueues are disabled.
4. Destroy all virtqueues via `doca_devemu_vblk_req_vq_destroy()`.
5. Stop and destroy the offload engine:
   - `doca_devemu_virtio_offload_engine_stop()`
   - `doca_devemu_vblk_offload_engine_destroy()`
6. Stop and destroy the PCI TLP type and devices.
7. Tear down the VirtIO Block subsystem via `doca_devemu_vblk_teardown()`.

5.4.20.4.7.9 Execution Phase

This section describes execution on BlueField ARM cores using DOCA Core progress engines.

Control Path

The DOCA Virtio-blk offload engine manages the control path, including:

- Virtqueue lifecycle — Creation, configuration, start, enable, disable, stop, and destruction of request virtqueues, driven by host driver state transitions (e.g., `DRIVER_OK`, device reset).
- Device reset — When the host driver writes `device_status = 0`, the virtio core library triggers a reset event. The application must tear down all active virtqueues, flush in-flight operations, and reinitialize the device.
- Offload engine state — The engine transitions through `DISABLED → ENABLED → DISABLING → DISABLED` as described in the DOCA DevEmu Virtio (TODO: link pending) offload engine lifecycle.

IO Path

This section describes the flow for a single virtio-blk request sent by the device driver until its completion.

It is assumed that the user properly configured an event handler for incoming virtio-blk requests using `doca_devemu_vblk_io_event_vblk_req_register()`.

It is also assumed that the user is familiar with the virtio-blk specification and has the ability to perform DMA operations to/from the host using DOCA DMA or any other suitable method.

The IO path for a block request:

1. Request arrival — The DOCA library invokes the registered event handler with the request object, type, sector, and user data.
2. Request inspection — The application examines the request type:
    - `VIRTIO_BLK_T_IN` (0) — Read from block device.
    - `VIRTIO_BLK_T_OUT` (1) — Write to block device.
    - `VIRTIO_BLK_T_GET_ID` (8) — Return 20-byte device serial number.
    - Other types (e.g., `VIRTIO_BLK_T_FLUSH`, `VIRTIO_BLK_T_DISCARD`) — The application decides how to handle these based on the feature bits it has exported to the host.
3. Data access — The application calls `doca_devemu_vblk_req_get_data()` to obtain a DOCA buffer representing the host memory data region, and `doca_devemu_vblk_req_get_data_len()` for the total data length.
4. DMA transfer — For read operations, the application populates a local buffer and uses DOCA DMA to transfer data to host memory. Note: For BLK_BT_IN and BLK_T_GET_ID, it needs to call `doca_buf_reset_data_len` for every data buf segment. For write operations, the application uses DOCA DMA to transfer data from host memory to a local buffer.
5. Request completion — The application calls `doca_devemu_vblk_req_complete(req, status, len)` with the appropriate status code and the number of bytes written to the device-writable portion of the buffer.

© Copyright 2026, NVIDIA.

5.4.21 DOCA STA

5.4.21.1 Introduction

The quality status of DOCA libraries is listed here.

The DOCA STA library simplifies the integration and offloading of storage target applications, such as SPDK, onto NVIDIA® BlueField®-3 and newer platforms. A storage target is any application that manages data storage access, whether from hard drives, SSDs, network-attached storage (NAS), or cloud services, and serves requests from local or remote initiators.

To enable network-based access, the NVMe-oF protocol extends the capabilities of NVMe storage across network fabrics like RoCEv1/IB RDMA and NVMeTCP. By allowing NVMe commands to execute over the network, NVMe-oF overcomes the limitations of direct-attached storage.

Applications supporting NVMe-oF are resource-intensive, requiring substantial CPU resources. To reduce CPU usage, improve latency, and conserve power, an offload solution is essential. With BlueField-3, storage target applications can leverage the data processing accelerator (DPA) to offload their operations.

DOCA STA provides a public API to manage both control-plane and data-plane operations, abstracting the complexity of communication between the offloaded target application and the hardware accelerator. This abstraction allows developers to streamline the integration process and focus on application logic.

Unlike traditional software-based storage targets that access NVMe drives over PCIe, DPA offload accelerators utilize PCIe peer-to-peer (P2P) topology to connect to NVMe drives. Special kernel configurations are required to enable this topology.

5.4.21.2 Prerequisites

To enable DPA storage target offload, a dedicated patched P2P kernel is required:

- For target applications running on the host machine, install a patched kernel that supports P2P.
  - Remove private items from linux/vfio_pci_core.h
  - Break up ioctl dispatch functions to one function per ioctl
  - Allow MMIO regions to be exported through dma-buf

    Note that the patches need to be applied manually and a custom kernel needs to be built to make it work.
- For target applications running on a BlueField machine, ensure a P2P-supported BFB is installed when the BlueField is directly attached to NVMe drives. For this you need the latest BFB with Ubuntu 22.04 (and DOCA v2.10.0).

In addition, the library makes use of existing DOCA libraries, therefore, it is recommended to read the following sections before:

- DOCA Comch
- DOCA RDMA
- DOCA DPA
- DOCA Core Device
- DOCA Core Memory Subsystem

5.4.21.3 Changes From Previous Release

N/A

5.4.21.4 Environment

DOCA STA-based applications can run either on the host machine or on the NVIDIA® BlueField-3® networking platform (DPU or SuperNIC).

When running on the BlueField platform, NVMe drives must be directly PCIe-attached, and BlueField-3 or newer is required.

5.4.21.5 Architecture

The DOCA STA library consists of two main components:

- Host side – The host-side component exposes an API for use by the target application. Using this API, the application can:
  - Initialize and configure accelerator parameters.
  - Register callbacks for data processing – Certain NVMe-oF command capsules that cannot be processed by the offload engine are forwarded to the library as "non-offloaded capsules." The application can process these capsules using registered callbacks.
  - Register callbacks for asynchronous events – The offload engine can generate notifications for events such as NVMe drive timeouts. The application can handle these events using registered callbacks.
  - Register tasks for various operations:
    - Datapath tasks – After processing non-offloaded capsules, the application can register tasks for RDMA operations, such as read, write, or send.
    - Control-path tasks – Some control operations, such as queue pair (QP) disconnection or backend destruction, may take time to complete asynchronously. The application can register tasks for these operations.
- Device side – The device-side component consists of low-level accelerator code that implements the target offload logic, such as DPA handler code.

5.4.21.5.1 DOCA STA Contexts

DOCA STA operates with two types of DOCA contexts:

- DOCA STA context
  - Responsible for system-wide initialization and configuration.
  - Manages all target system-wide operations.
  - There should be only one instance of the DOCA STA context in the system (or virtual machine).
- DOCA STA IO context:
  - Responsible for RDMA IO operations.
  - Can exist for each system thread (e.g., per `pthread`).

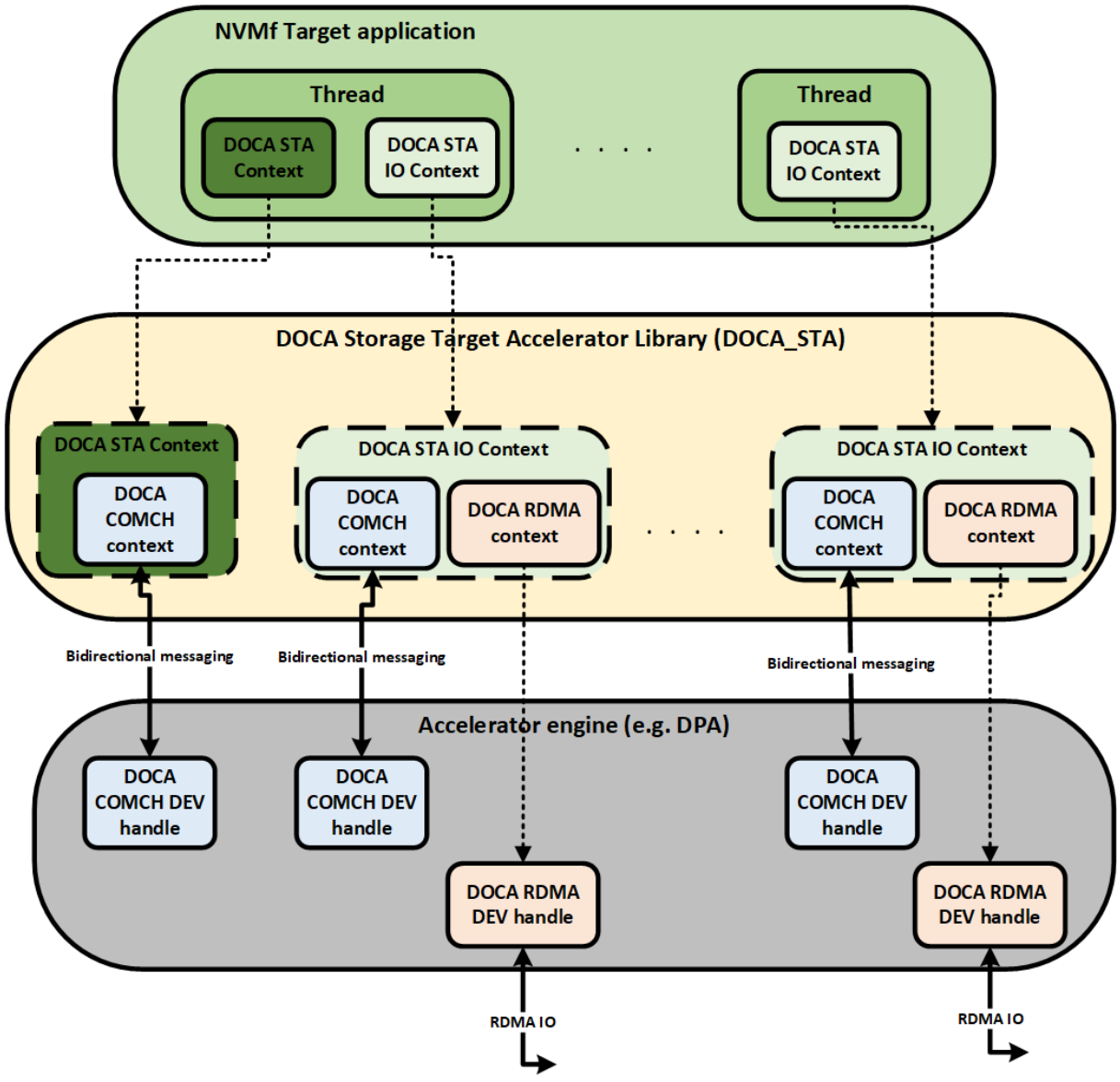
5.4.21.5.2 Key DOCA Libraries

DOCA STA relies on several other DOCA libraries for its functionality. The key libraries are:

- DOCA RDMA – Used for connecting or disconnecting queue pairs (QPs) on the accelerator datapath (e.g., DPA).
- DOCA Comch – Enables communication to and from the offload accelerator engine. For example, it facilitates posting RDMA tasks and receiving completions or handling QP disconnect tasks and their completions.

5.4.21.5.3 High-Level Connectivity

The following diagram illustrates the high-level connectivity of DOCA STA:



5.4.21.5.4 Objects

5.4.21.5.4.1 Device and Device Representor

The DOCA STA library utilizes two types of devices: a control device (PF) and one or more network devices (SF).

- Control device (PF) – This device is used to configure the DPA, such as loading the DPA-related application image.
- Network device (SF) – This device handles incoming and outgoing NVMe-oF traffic.

Applications using the DOCA STA library must select both a control device and network devices with the appropriate representors. For more information, refer to the sections on DOCA Core Device Discovery and DOCA Core Device Representor Discovery.

5.4.21.6 Configuration Phase

To begin using the DOCA STA library, users must complete a configuration phase as outlined in the DOCA Core Context Configuration Phase.

This section details the steps required to configure and initialize the DOCA STA context.

5.4.21.6.1 Configurations

The DOCA STA context must be configured to align with the application's use case.

The configuration process involves the following steps:

1. Create the DOCA STA object (main context).
2. Add network device(s).
3. Connect the DOCA STA context to the DOCA progress engine (PE).
4. Start the DOCA STA context.
5. Create the DOCA STA IO object(s) (IO context/IO threads).
6. Connect the DOCA STA IO context(s) to the DOCA PE.
7. Start the DOCA STA IO context(s).

> ⚠ The IO context(s) can only be created after the main context has been started.

The following order must be followed during the start phase:

1. Create the main context.
2. Start the main context.
3. Create IO context(s).
4. Start IO context(s).

The following order must be followed during the stop phase:

1. Stop IO context(s).
2. Stop the main context.

5.4.21.6.1.1 Create Main Context

The library relies heavily on the DPA, so the control device (PF) with STA functionality must be used to create the `doca-sta` object.

Use the `doca_sta_cap_is_supported` API to verify if the PF device supports STA functionality before proceeding.

5.4.21.6.1.2 Add Network Device

The library utilizes network device(s) (SF) to manage incoming and outgoing NVMe-oF traffic.

Before starting the `doca-sta` object, network devices (resources) must be added to it. Use the `doca_sta_add_dev` API to add network devices (SF) as needed.

5.4.21.6.1.3 Progress Engine

The Progress Engine (PE) is responsible for progressing tasks and events. Each PE is associated with one or more DOCA contexts, but a DOCA context can only be associated with a single PE.

It is the responsibility of the application using the library to create a dedicated PE and connect the `doca-sta` context to it.

5.4.21.6.1.4 Start Main Context

Once the `doca-sta` context is created and configured, it must be started to enable its functionality.

5.4.21.6.1.5 Create IO Context

The IO context represents the IO thread, which is responsible for initiating RDMA connectivity flows and handling "non-offload" commands.

Examples of such flows include:

- Connecting QPs
- Disconnecting QPs
- Handling non-offload commands

Non-offload commands refer to commands that the `doca-sta` library does not handle internally. These commands must be forwarded to the application layer for processing.

The IO context should be created in a similar manner to the main context.

The application using the library is responsible for creating a dedicated PE for each IO context and connecting the `doca-sta-io` context to this PE.

5.4.21.6.1.6 Start IO Context

Once the **doca-sta-io** context is created and configured, it must be started to enable IO-related operations.

5.4.21.7 Execution Phase

The DOCA STA library enables offloading of NVMe-oF traffic to the DPA, providing APIs to configure an NVMe-oF target application in compliance with the NVMe specification.

Before the application can begin receiving traffic from the initiator, the following steps must be completed:

1. Add subsystems.
2. Add namespaces to the subsystems.
3. Bind NVMe PCI disks to the namespaces.

> ⚠ The application is responsible for implementing the RDMA connection management service.

For additional details, refer to the **doca_sta_subsystem.h** and **doca_sta_be.h** header files.

5.4.21.7.1 RDMA Connect

Upon receiving an RDMA connection request, the application should use one of the following APIs:

- **doca_sta_io_qp_alloc** + **doca_sta_io_qp_accept**
- **doca_sta_io_qp_connect**

If the operation completes successfully, an STA QP (Queue Pair) is created, represented by an STA QP handle.

> ⚠ All communication between the application layer and QP API (and vice versa) must occur within the same IO thread (IO context) from which the QP was created.

5.4.21.7.1.1 Fabric Connect

After a connection is established, the initiator will immediately send a "Fabric Connect" (FC) command capsule.

The application is responsible for handling the FC command as part of the non-offload flow. This includes:

1. Retrieving the data/payload for the FC command (if not received as inline data).
2. Verifying the parameters inside the FC command (e.g., **host NQN** , **subsystem NQN**).
3. Completing FC processing by sending a response to the initiator.

> ⚠ All actions initiated by the application layer are asynchronous and are part of the non-offload flow. Communication between the DPA and host (DPU) is handled in this context.

For further details, refer to the **doca_sta_io_non_offload.h** header file.

5.4.21.7.2 Tasks

The DOCA STA library provides support for asynchronous tasks to handle various asynchronous flows, such as:

- Disconnecting a QP
- Removing (detaching) a namespace
- Destroying a back-end queue

The asynchronous flow is initiated by the application through the allocation of a task using the appropriate API. Once the task is allocated, the application is responsible for submitting it to trigger the desired action.

After submission, the library manages the execution and monitors the completion of the asynchronous flow. When the operation is completed, the library notifies the application by invoking a callback function.

5.4.21.7.2.1 Task Input

Common input as described in DOCA Core Task.

5.4.21.7.2.2 Task Output

Common output as described in DOCA Core Task.

5.4.21.7.2.3 Task Limitations

- The operation is not atomic
- Other limitations are described in the DOCA Core Task

5.4.21.7.2.4 Non-offload Flow Task

The application layer is responsible for processing NVMe-oF capsules within the non-offload flow.

Initially, the NVMe-oF capsule is delivered from the DPA to the host and then delegated to the application layer for processing. During this process, the application may need to perform RDMA operations to handle the capsule effectively.

> ⚠ The QP is managed by the DOCA STA library and must only be accessed through the library's API.

The following non-offload tasks can be used by the application layer to initiate RDMA operations for processing non-offload capsules:

- RDMA-READ
- RDMA-WRITE with RDMA-SEND

- RDMA-SEND

For additional details, refer to the `doca_sta_io_non_offload.h` file.

Non-offload RDMA-READ

The RDMA-READ task initiates an RDMA-READ operation. Upon completion of the operation, the library will invoke the appropriate callback function to notify the application layer of its completion.

Task Configuration

Description	API to Set the Configuration
Enable the task	`doca_sta_io_task_non_offload_set_rdma_read_conf`

Task Allocation

Description	API to Set the Configuration
Allocation the task	`doca_sta_io_task_non_offload_rdma_read_alloc_init`

Task Completion Success

After the task is completed successfully:

- The RDMA-READ operation is completed successfully

Task Completion Failure

If the task fails midway:

- The RDMA-READ operation fails

Non-offload RDMA-WRITE + RDMA-SEND

This task is responsible for initiating an RDMA-WRITE operation followed by an RDMA-SEND operation.

Upon successful completion of the operation, the library will invoke the corresponding callback function to notify the application layer about its completion.

Task Configuration

Description	API to Set the Configuration
Enable the task	`doca_sta_io_task_non_offload_set_rdma_write_send_conf`

Task Allocation

Description	API to Set the Configuration
Allocation the task	`doca_sta_io_task_non_offload_rdma_write_send_alloc_init`

Task Completion Success

After the task is completed successfully:

- The RDMA-WRITE and RDMA-SEND operation completed successfully

Task Completion Failure

If the task fails midway:

- The RDMA-WRITE and RDMA-SEND operation fails

Non-offload RDMA-SEND

This task is responsible for initiating an RDMA-SEND operation.

Upon completion, the library will invoke the appropriate callback function to notify the application layer of its successful completion.

Task Configuration

Description	API to Set the Configuration
Enable the task	`doca_sta_io_task_non_offload_set_rdma_write_send_conf`

Task Allocation

Description	API to Set the Configuration
Allocation the task	`doca_sta_io_task_non_offload_rdma_send_alloc_init`

Task Completion Success

After the task is completed successfully:

- The RDMA-SEND operation is completed successfully

Task Completion Failure

If the task fails midway:

- The RDMA-SEND operation fails

5.4.21.7.2.5 QP Disconnect Task

The QP disconnect task is responsible for initiating the asynchronous flow required to disconnect a QP. The process includes the following steps:

1. Move the QP to the error state.
2. Send a notification message from the host (DPU) to the corresponding DPA, indicating the QP disconnect request.
3. The DPA verifies that there are no pending inbound or outbound IO operations.
4. Once verification is complete, the DPA sends a response back to the host (DPU).
5. The library notifies the application about the completion of the disconnect task through a callback function.

Task Configuration

Description	API to Set the Configuration
Enable the task	`doca_sta_io_task_disconnect_set_conf`

Task Allocation

Description	API to Set the Configuration
Allocation the task	`doca_sta_io_task_disconnect_alloc_init`

Task Completion Success

After the task is completed successfully:

- The QP is disconnected and might be destroyed

Task Completion Failure

If the task fails midway:

- The QP was not disconnected

5.4.21.7.2.6 Remove Namespace Task

The remove (detach) namespace task initiates the asynchronous process for detaching a namespace. The flow includes the following steps:

1. Send a notification message from the host (DPU) to the corresponding DPA about the detach namespace request.
2. The DPA marks the namespace as no longer valid.
3. The DPA sends a response back to the host (DPU) confirming the detachment.
4. The library notifies the application about the task's completion through a callback function.

Task Configuration

Description	API to Set the Configuration
Enable the task	`doca_sta_subsystem_task_rm_ns_set_conf`

Task Allocation

Description	API to Set the Configuration
Allocation the task	`doca_sta_subsystem_task_rm_ns_alloc_init`

Task Completion Success

After the task is completed successfully:

- The namespace is removed
- Any I/O sent to the removed namespace is completed with an error

Task Completion Failure

If the task fails midway:

- The namespace is not removed

5.4.21.7.2.7 Destroy Back-end Queue Task

The destroy back-end (BE) queue task initiates the asynchronous process for destroying a back-end queue. The flow includes the following steps:

1. Send a notification message from the host (DPU) to the appropriate DPA to request the destruction of the back-end queue.
2. The DPA marks the back-end queue as destroyed.
3. Any outstanding commands in the queue are completed with an error status.
4. The DPA sends a response back to the host (DPU) confirming the destruction.
5. The library notifies the application about the task's completion through a callback function.

Task Configuration

Description	API to Set the Configuration
Enable the task	`doca_sta_be_task_destroy_queue_set_conf`

Task Allocation

Description	API to Set the Configuration
Allocation the task	`doca_sta_be_destroy_queue_task_alloc_init`

Task Completion Success

After the task is completed successfully:

- The BE queue is removed

Task Completion Failure

If the task fails midway:

- The BE queue is not removed

5.4.21.8 State Machine

The DOCA STA library follows the context state machine as described in DOCA Core Context State Machine.

The following section describes how to move states and what is allowed in each state.

5.4.21.8.1 Idle

In this state, it is expected that the application:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to the section
- Starting the context

It is possible to reach this state as follows:

Previous State	Transition Action
None	Create the context
Running	Call stop after making sure all tasks have been freed
Stopping	Call progress until all tasks are completed and freed

5.4.21.8.2 Starting

This state cannot be reached.

5.4.21.8.3 Running

In this state, it is expected that the application:

- Allocates and submits tasks
- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocating a previously configured task
- Submitting a task
- Calling stop

It is possible to reach this state as follows:

Previous State	Transition Action
Idle	Call start after the configuration

5.4.21.8.4 Stopping

In this state, it is expected that the application:

- Calls progress to complete all inflight tasks (tasks completed with failure)
- Frees any completed tasks

Allowed operations:

- Call progress

It is possible to reach this state as follows:

Previous State	Transition Action
Running	Call progress and a fatal error occurs
Running	Call stop without freeing all tasks

5.4.22 DOCA Management

This guide provides instructions on how to use the DOCA Management API for developing applications for device management and configuration.

5.4.22.1 Introduction

> ⚠ The quality status of DOCA libraries is listed here.

DOCA Management provides an API to manage NVIDIA® BlueField® DPU and host devices and their associated functions (PFs, VFs, and SFs).

Using DOCA Management, administrators and applications can dynamically configure and query device settings, enable or disable capabilities, and manage device behavior at runtime through a standardized API that communicates directly with device firmware via the Linux kernel's **fwctl** framework.

This document is intended for software developers and system administrators wishing to manage and configure DPU and host devices programmatically, control device capabilities, and integrate device management into their applications.

5.4.22.2 Prerequisites

DOCA Management library uses the Linux kernel's **fwctl** framework and as such requires the following kernel drivers to be available and loaded:

- **fwctl.ko** – fwctl device firmware access framework
- **mlx5_fwctl.ko** – mlx5 ConnectX fwctl driver

The library also uses DOCA Core Device for device and device representor discovery. It is recommended to read the following section before proceeding:

- DOCA Core Device

5.4.22.3 Library Changes From Previous Releases

5.4.22.3.1 Changes in 3.3.0

5.4.22.3.1.1 Added

- **doca_mgmt_icm_quota.h**

5.4.22.3.2 Changes in 3.4.0

5.4.22.3.2.1 Added

- **doca_mgmt_diagnostics_data.h**
- **doca_mgmt.h** :

- `doca_mgmt_dev_rep_ctx_create_by_pci_addr()`

5.4.22.3.3 Changes in 3.5.0

5.4.22.3.3.1 Added

- `doca_mgmt_raw_common.h`
- `doca_mgmt_raw_register.h`
- `doca_mgmt_raw_hca_cap.h`
- `doca_mgmt.h` (management target discovery and target-string context creation APIs):
  - `doca_mgmt_target_info_dev_create_list()` / `doca_mgmt_target_info_dev_destroy_list()`
  - `doca_mgmt_target_info_dev_rep_create_list()` / `doca_mgmt_target_info_dev_rep_destroy_list()`
  - `doca_mgmt_target_info_dev_get_*()` and `doca_mgmt_target_info_dev_rep_get_*()` accessors
  - `doca_mgmt_dev_ctx_create_from_target_string()`
  - `doca_mgmt_dev_rep_ctx_create_from_target_string()`

5.4.22.4 Environment

DOCA Management-based applications can run on the host machine and on the NVIDIA® BlueField® DPU.

Management operations are performed either through a *management device context* or through a *management device representor context*. The choice of context depends on the operation: some operations are performed on the device itself, while others (such as configuring the capabilities of a child VF/SF) can only be performed through the function's representor. Representor-based operations must therefore be executed where the representor resides (for example, on the BlueField® DPU or on the host PF that owns the representor, depending on the deployment).

Since the library communicates with the device firmware through the Linux kernel's `fwctl` framework, the invoking process must have sufficient permissions to access the corresponding `fwctl` character device.

Certain operations impose additional requirements on the target function. For example, setting device capabilities requires the represented device to be uninitialized (i.e., unbound from the `mlx5_core` driver) at the time the capabilities are set.

5.4.22.5 Architecture

The library operates through *management device contexts* and *management device representor contexts*. These management contexts are the main objects of the library. They hold a management interface for the device and are passed to other APIs of the library to specify the device to operate on.

To operate on a device or representor, the application must first identify it. This can be done either through the library's own management target discovery, which enumerates the manageable devices and representors present on the system (see Management Contexts and Targets), or through DOCA Core device discovery.

The library provides two sets of APIs:

- Typed, feature-specific APIs – each targets a specific feature and exposes a dedicated, strongly typed interface for it (for example, `doca_mgmt_icm_quota`, `doca_mgmt_cc_global_status`, etc.). These APIs are built around dedicated management handles (see Dedicated Management Handles) that encapsulate the feature's configuration attributes and provide setters/getters together with query/modify operations that apply or read the attributes to or from the device. This is the recommended way to configure a feature, if a dedicated API for it exists.
- Generic APIs (the `doca_mgmt_raw_*` family) – provide low-level access to the device for advanced users. They allow querying and modifying device registers and HCA capabilities by name, as well as sending arbitrary firmware commands directly to the device. These APIs are intended for cases where a dedicated typed API is not available, and require detailed knowledge of the underlying device interface (see Raw APIs).

5.4.22.5.1 Objects

5.4.22.5.1.1 Management Contexts and Targets

The DOCA Management device context (`doca_mgmt_dev_ctx`) represents a management interface to a device and is used to perform device-level management operations. Similarly, the DOCA Management device representor context (`doca_mgmt_dev_rep_ctx`) represents a management interface to a device representor and is used to perform management operations for the device represented by the representor.

The management device context and management device representor context function as handles that are passed to other DOCA Management APIs to specify the target device. Some DOCA Management APIs accept a management device context, while others accept a management device representor context.

A *management target* is a device or device representor that can be managed by the library. The library provides a discovery mechanism that enumerates the management targets present on the system, so that an application can identify and select the device or representor to operate on.

The discovery APIs return opaque target information objects – `doca_mgmt_target_info_dev` for devices and `doca_mgmt_target_info_dev_rep` for device representors. Each object exposes accessors for the target's attributes (such as its PCI function type, interface name, and, for representors, the controller, PF, VF/SF indices, and VHCA ID) and, in particular, a *target string* that uniquely identifies the target.

A management context is obtained by passing a target string, discovered through the discovery APIs, to the corresponding target-string context creation API (see Device Context Creation). The target string formats are:

Target	Target String Format	Example
Device	`pci/<BDF>`	`pci/0000:08:00.0`
Uplink representor	`pci/<BDF>`	`pci/0000:08:00.0`
PF representor	`pci/<BDF>,c<controller>pf<pf>vf-1`	`pci/0000:08:00.0,c0pf0vf-1`

Target	Target String Format	Example
VF representor	`pci/<BDF>,c<controller>pf<pf>vf<vf>`	`pci/0000:08:00.0,c0pf0vf2`
SF representor	`pci/<BDF>,c<controller>pf<pf>sf<sf>`	`pci/0000:08:00.0,c0pf0sf88`

Alternatively, a management context can be created directly from a DOCA Core device or device representor, without going through management target discovery (see DOCA Core Device).

5.4.22.5.1.2 Dedicated Management Handles

The library provides dedicated handles for specific management operations. Each handle is an opaque structure that encapsulates configuration attributes and provides APIs to both configure the handle (via setters/getters) and apply/query these attributes to or from the device.

The following table lists the currently supported handles:

Handle	Description	Context	Notes
`doca_mgmt_cc_global_status`	Holds congestion control global status configuration parameters	Device	Configuration applies to PFs only. Enables or disables congestion control operation for the PF and its child functions based on priority (0–7) and protocol (reaction point/notification point).
`doca_mgmt_device_caps_general`	Holds general device capabilities configuration parameters	Device representor	Configuration is done via the representor. The represented device must be uninitialized (unbound from `mlx5_core`) when setting capabilities.
`doca_mgmt_diagnostics_data`	Holds diagnostics data configuration parameters	Device	Configuration is performed on a PF, not a representor. On supported hardware, multi-domain diagnostic data is governed by `DIAG_DATA_OWNERSHIP`. The device advertises its support and domain limits through NIC capabilities registers.
`doca_mgmt_icm_quota`	Holds ICM quota configuration parameters	Device or device representor	Interconnect Context Memory is a portion of host memory allocated exclusively for the HCA to maintain and manage its internal control objects. The amount of ICM required depends directly on the device's capabilities and the scale of resources it must support. On supported devices, the ICM Quota feature allows administrators to limit the amount of ICM a specific function is allowed to consume, query current ICM usage information, and configure these limits on both physical devices and representors.

5.4.22.5.1.3 Raw API Objects

Raw API allows configuring device registers and HCA capabilities. Unlike the typed feature-specific APIs, the generic `doca_mgmt_raw_*` APIs are not built around dedicated handles. Instead, they reference registers and capability types by name and their contents are expressed as a list of fields.

The library provides APIs to enumerate the supported registers and HCA capability types, and to describe the fields of a given register or HCA capability type. The describe APIs return *field descriptors*, each represented by a **doca_mgmt_raw_field_desc** structure, which holds:

- **name** – the name of the field.
- **bit_size** – the size of the field, in bits.
- **access** – the access type of the field, one of the **doca_mgmt_raw_field_access** enum values:
  - **DOCA_MGMT_RAW_FIELD_ACCESS_RO** – read-only.
  - **DOCA_MGMT_RAW_FIELD_ACCESS_WO** – write-only.
  - **DOCA_MGMT_RAW_FIELD_ACCESS_RW** – read-write.
  - **DOCA_MGMT_RAW_FIELD_ACCESS_INDEX** – an index field, provided as input to select the specific entity to access.
  - **DOCA_MGMT_RAW_FIELD_ACCESS_OP** – an operation field, provided as input to select the operation to perform.
  - **DOCA_MGMT_RAW_FIELD_ACCESS_UNKNOWN** – the access type is not known.

The library also provides APIs to query and modify the fields of a register or HCA capability type by name. A field value is represented by a **doca_mgmt_raw_field** structure, which holds:

- **name** – the name of the field.
- **value** – the 32-bit value of the field.

For HCA capabilities, a query or modify additionally specifies a capability mode (applicable only for query) and a target function. The capability mode is given by the **doca_mgmt_raw_hca_cap_mode** enum, which selects which set of values is accessed:

- **DOCA_MGMT_RAW_HCA_CAP_MODE_MAX** – the maximum capability values supported by the device.
- **DOCA_MGMT_RAW_HCA_CAP_MODE_CUR** – the current, effective capability values.

The target function is given by the **doca_mgmt_raw_hca_cap_target** structure, which identifies the function whose capabilities are accessed, according to its type:

- **DOCA_MGMT_RAW_HCA_CAP_TARGET_TYPE_SELF** – the function executing the operation.
- **DOCA_MGMT_RAW_HCA_CAP_TARGET_TYPE_OTHER_FUNCTION_ID** – another function, identified by its function ID.
- **DOCA_MGMT_RAW_HCA_CAP_TARGET_TYPE_OTHER_VHCA_ID** – another function, identified by its VHCA ID.

There is a known limitation where the access type of HCA capability fields is currently reported as **DOCA_MGMT_RAW_FIELD_ACCESS_UNKNOWN**. When modifying HCA capabilities, the user should determine whether a given field is writable by other means.

Finally, the raw command API allows sending an arbitrary firmware command directly to the device. The command is described by an input and output payload and the appropriate command scope. The scope of the command is given by the **doca_mgmt_cmd_scope** enum which maps exactly to the fwctl command scopes:

- **DOCA_MGMT_CMD_SCOPE_CONFIGURATION**

- `DOCA_MGMT_CMD_SCOPE_DEBUG_READ_ONLY`
- `DOCA_MGMT_CMD_SCOPE_DEBUG_WRITE`
- `DOCA_MGMT_CMD_SCOPE_DEBUG_WRITE_FULL`

See Raw APIs for the expected usage flow.

5.4.22.6 Expected Usage Flow

The following section describes the expected usage flow of the DOCA Management library. It first covers the creation of the management contexts, which is common to all operations, and then presents representative examples: ICM quota configuration as an example of a typed, feature-specific API, and the raw APIs as an example of the generic APIs. The remaining typed feature-specific APIs follow the same pattern as the ICM quota example; refer to the Dedicated Management Handles table and the DOCA Management Samples for details on each of them.

5.4.22.6.1 Device Context Creation

To start using the library, a DOCA Management device context (`doca_mgmt_dev_ctx`) must be created. The context can be created in one of two ways: from a management target string, or from a DOCA device.

5.4.22.6.1.1 Creating a Device Context From a Target String

1. Obtain a device target string:
    a. Create the list of device targets by calling `doca_mgmt_target_info_dev_create_list()` .
    b. Select the desired device by inspecting the attributes of the returned entries using the accessors, and retrieve its target string using `doca_mgmt_target_info_dev_get_target_string()` .
    c. Free the list using `doca_mgmt_target_info_dev_destroy_list()` when it is no longer needed. The target string is owned by its list entry, so copy it if it must outlive the list.
2. Create the `doca_mgmt_dev_ctx` by calling `doca_mgmt_dev_ctx_create_from_target_string()` with the target string.
3. The created `doca_mgmt_dev_ctx` can now be used to configure the device or to create a DOCA Management device representor context for one of its representors (as explained in Device Representor Context Creation). When the application finishes using it, destroy it using `doca_mgmt_dev_ctx_destroy()` .

5.4.22.6.1.2 Creating a Device Context From a DOCA Device

1. Open the `doca_dev` that should be configured.
    For information about opening a DOCA device, see DOCA Core Device Discovery.
2. Create the corresponding `doca_mgmt_dev_ctx` by calling `doca_mgmt_dev_ctx_create()` with the opened `doca_dev` .
3. The created `doca_mgmt_dev_ctx` can now be used to configure the device or to create a DOCA Management device representor context for one of its representors (as explained in Device Representor Context Creation).

4. When the application finishes using the `doca_mgmt_dev_ctx`, destroy it using `doca_mgmt_dev_ctx_destroy()`. The `doca_dev` must remain valid until the `doca_mgmt_dev_ctx` is destroyed.

5.4.22.6.2 Device Representor Context Creation

Some configurations (such as configuring device capabilities) can be done only via the device representor. For this, a DOCA Management device representor context (`doca_mgmt_dev_rep_ctx`) must be created. As with the device context, it can be created in one of two ways: from a management target string, or from a DOCA device representor.

5.4.22.6.2.1 Creating a Device Representor Context From a Target String

1. Obtain a representor target string:
    a. Create the list of device targets by calling `doca_mgmt_target_info_dev_create_list()` and select the representor's parent device, as described for a device target string above.
    b. Create the list of that device's representor targets by calling `doca_mgmt_target_info_dev_rep_create_list()`, passing the `doca_mgmt_target_info_dev` of the desired parent device.
    c. Select the desired representor by inspecting the attributes of the returned entries using the accessors, and retrieve its target string using `doca_mgmt_target_info_dev_rep_get_target_string()`.
    d. Free the lists using `doca_mgmt_target_info_dev_rep_destroy_list()` and `doca_mgmt_target_info_dev_destroy_list()` when they are no longer needed. The target string is owned by its list entry, so copy it if it must outlive the list.
2. Create the `doca_mgmt_dev_rep_ctx` by calling `doca_mgmt_dev_rep_ctx_create_from_target_string()` with the representor target string. The library resolves the target string internally and creates the underlying management device context on the application's behalf.
3. Use the created `doca_mgmt_dev_rep_ctx` to configure the represented device, and destroy it using `doca_mgmt_dev_rep_ctx_destroy()` when finished. In this case the representor management context owns the underlying management device context and destroys it automatically.

5.4.22.6.2.2 Creating a Device Representor Context From a DOCA Device Representor

1. Create a `doca_mgmt_dev_ctx` for the parent device of the representor (as explained in Device Context Creation). For example, if `VF0` of `PF0` should be configured, create a `doca_mgmt_dev_ctx` for `PF0`.
2. Open the `doca_dev_rep` that should be configured. For information about opening a DOCA device representor, see DOCA Core Device Representor Discovery.
3. Create the corresponding `doca_mgmt_dev_rep_ctx` by calling `doca_mgmt_dev_rep_ctx_create()` with the created `doca_mgmt_dev_ctx` and the opened `doca_dev_rep`.

4. The created `doca_mgmt_dev_rep_ctx` can now be used to configure the device represented by the representor.
5. When the application finishes using the `doca_mgmt_dev_rep_ctx`, destroy it using `doca_mgmt_dev_rep_ctx_destroy()`. Both the `doca_dev_rep` and `doca_mgmt_dev_ctx` must remain valid until the `doca_mgmt_dev_rep_ctx` is destroyed.

5.4.22.6.3 ICM Quota Configuration

ICM Quota configuration is supported on both devices and representors.

5.4.22.6.3.1 Querying ICM Quota Capabilities

Not all devices support ICM quota configuration. To check for support and discover capabilities:

1. Create a `doca_mgmt_dev_ctx` for the target device.
2. Check for support by calling `doca_mgmt_cap_icm_quota_is_supported()`.
3. If supported, retrieve the maximum quota limit using `doca_mgmt_cap_icm_quota_get_max_limit()`.

   > ⚠ This returned value represents the maximum valid input for `doca_mgmt_icm_quota_set_limit()`.

   > ⚠ If a device supports ICM Quota, its associated representors also support it and share the same limit.

5.4.22.6.3.2 Getting ICM Quota Configuration

To retrieve the current configuration:

1. Create a `doca_mgmt_dev_ctx` or a `doca_mgmt_dev_rep_ctx` for the device or device representor that should be configured (as explained in Device Context Creation and Device Representor Context Creation).
2. Initialize a `doca_mgmt_icm_quota` handle for the device or the device representor by calling `doca_mgmt_icm_quota_create_for_dev()` or `doca_mgmt_icm_quota_create_for_dev_rep()` with the created `doca_mgmt_dev_ctx` or `doca_mgmt_dev_rep_ctx`, respectively. The created handle will be associated with the given device or device representor and the following query will be applied to this device or device representor.
3. Query the ICM quota configuration by calling `doca_mgmt_icm_quota_query()`.
4. Use the following getters to read the results:
   - `doca_mgmt_icm_quota_get_limit()`
   - `doca_mgmt_icm_quota_get_current_allocation()`
   - `doca_mgmt_icm_quota_get_max_reached()`
5. Destroy the handle using `doca_mgmt_icm_quota_destroy()`.

5.4.22.6.3.3 Setting ICM Quota Configuration

1. Create a `doca_mgmt_dev_ctx` or a `doca_mgmt_dev_rep_ctx` for the device or device representor that should be configured (as explained in Device Context Creation and Device Representor Context Creation).
2. Initialize a `doca_mgmt_icm_quota` handle for the device or the device representor by calling `doca_mgmt_icm_quota_create_for_dev()` or `doca_mgmt_icm_quota_create_for_dev_rep()` with the created `doca_mgmt_dev_ctx` or `doca_mgmt_dev_rep_ctx`, respectively. The created handle will be associated with the given device or device representor and the following configuration will be applied to this device or device representor.
3. Set the relevant attributes to the handle:
    a. Set the quota limit using `doca_mgmt_icm_quota_set_limit()`.
    b. Reset the "max reached" counter using `doca_mgmt_icm_quota_set_reset_max_reached()`.
4. Commit the configuration by calling `doca_mgmt_icm_quota_modify()`.
5. Destroy the handle using `doca_mgmt_icm_quota_destroy()`.

5.4.22.6.4 Raw APIs

For advanced use cases, the library provides the generic `doca_mgmt_raw_*` APIs, which allow accessing the device directly rather than through a dedicated, feature-specific abstraction. These APIs are organized into three groups: raw register access, raw HCA capability access, and raw firmware commands.

All raw APIs operate on a management device context (`doca_mgmt_dev_ctx`), created as explained in Device Context Creation.

> ⚠ The raw APIs are intended for advanced users with detailed knowledge of the device firmware interface. It is recommended to use the higher-level dedicated APIs (see Dedicated Management Handles) when a dedicated API for the required operation is available. The detailed description of each register and HCA capability type and their fields is outside the scope of this API and may be obtained separately.

5.4.22.6.4.1 Raw Register

The raw register API allows querying and modifying device registers by name. A register is expressed as a set of named fields.

1. Create a `doca_mgmt_dev_ctx` for the target device.
2. Optionally, retrieve the list of supported registers by calling `doca_mgmt_raw_register_list()`. The returned list must be freed using `doca_mgmt_raw_str_list_free()`.
3. Optionally, retrieve the field descriptors of a specific register by calling `doca_mgmt_raw_register_describe()` with the register name. The returned list must be freed using `doca_mgmt_raw_field_desc_list_free()`.
4. Query a register:

- To query specific fields, call **doca_mgmt_raw_register_query()** with the register name, optional input fields, and the output fields to read.
- To query all fields, call **doca_mgmt_raw_register_query_all()** with the register name and optional input fields. The returned fields must be freed using **doca_mgmt_raw_field_list_free()**.

5. To modify a register, call **doca_mgmt_raw_register_modify()** with the register name and the input fields to set.

5.4.22.6.4.2 Raw HCA Capability

The raw HCA capability API allows querying and modifying the HCA capabilities of a function by capability type name, following the same field-based model as the raw register API.

1. Create a **doca_mgmt_dev_ctx** for the target device.
2. Optionally, retrieve the list of supported HCA capability types by calling **doca_mgmt_raw_hca_cap_list()**. The returned list must be freed using **doca_mgmt_raw_str_list_free()**.
3. Optionally, retrieve the field descriptors of a specific HCA capability type by calling **doca_mgmt_raw_hca_cap_describe()** with the capability type name. The returned list must be freed using **doca_mgmt_raw_field_desc_list_free()**.
4. Query HCA capabilities:
   - To query specific fields, call **doca_mgmt_raw_hca_cap_query()** with the capability type name, the capability mode (max or current), the target function (the function that executes the operation, or another function identified by its function ID or VHCA ID), and the output fields to read.
   - To query all fields, call **doca_mgmt_raw_hca_cap_query_all()** with the capability type name, the capability mode (max or current), the target function (the function that executes the operation, or another function identified by its function ID or VHCA ID). The returned fields must be freed using **doca_mgmt_raw_field_list_free()**.
5. To modify HCA capabilities, call **doca_mgmt_raw_hca_cap_modify()** with the target function and the input fields to set. The target function must be a function other than the one executing the operation (identified by its function ID or VHCA ID).

> ⚠ HCA capabilities can be set only on a function that has not been initialized yet. For example, the function must not be bound to the **mlx5_core** driver when setting its HCA capabilities.

> ⚠ There is a known limitation where the access type of HCA capability fields is currently reported as **DOCA_MGMT_RAW_FIELD_ACCESS_UNKNOWN**. When modifying HCA capabilities, the user should determine whether a given field is writable by other means.

5.4.22.6.4.3 Raw Command

The raw command API allows sending an arbitrary firmware command directly to the device. The following steps detail how to send a raw command to a device:

1. Create a `doca_mgmt_dev_ctx` for the device that should receive the command.
2. Prepare the input payload buffer according to the specific command requirements.
3. Allocate an output payload buffer to receive the command response.
4. Call `doca_mgmt_raw_cmd()` with the following parameters:
    - The created `doca_mgmt_dev_ctx`
    - A command identifier string (for logging)
    - The command scope
    - Pointer to the input payload buffer and its size
    - Pointer to the output payload buffer and its size
5. Check the return value to verify that the command executed successfully.
6. Parse the output payload buffer according to the specific command response format.

The `command scope` parameter specifies the scope of the command being performed and must match the `fwctl` scope assigned to it by the `mlx5_fwctl` driver.

The mapping between DOCA and `fwctl` command scopes is as follows:

DOCA Management Command Scope	Corresponding fwctl Scope
`DOCA_MGMT_CMD_SCOPE_CONFIGURATION`	`FWCTL_RPC_CONFIGURATION`
`DOCA_MGMT_CMD_SCOPE_DEBUG_READ_ONLY`	`FWCTL_RPC_DEBUG_READ_ONLY`
`DOCA_MGMT_CMD_SCOPE_DEBUG_WRITE`	`FWCTL_RPC_DEBUG_WRITE`
`DOCA_MGMT_CMD_SCOPE_DEBUG_WRITE_FULL`	`FWCTL_RPC_DEBUG_WRITE_FULL`

5.4.22.7 DOCA Management Samples

This section describes DOCA Management samples based on the DOCA Management library. Each sample is a standalone command-line tool. The samples demonstrate how to use the DOCA Management API for the following device management tasks:

- Configure device data direct capability.
- Configure device congestion control global status.
- Configure and query ICM quota.
- Configure and query diagnostics data.
- Access device registers, HCA capabilities, and firmware commands directly through the raw APIs.

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

5.4.22.7.1 Running the Samples

Refer to the following documents:

- *DOCA Installation Guide for Linux* for details on how to install BlueField-related software.

- *NVIDIA BlueField Platform Software Troubleshooting Guide* for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

To build a given sample, run the following command. If you downloaded the sample from GitHub, update the path in the first line to reflect the location of the sample file:

```
cd /opt/mellanox/doca/samples/doca_mgmt/<sample_name>
meson /tmp/build
ninja -C /tmp/build
```

The binary **doca_<sample_name>** is created under **/tmp/build/**.

Sample usage (e.g., **doca_mgmt_data_direct**):

```
Usage: doca_mgmt_data_direct [Program Commands] [DOCA Flags]

Program Commands:
  get                              Get data direct capability
  list-targets                     List the available management targets
  set                              Set data direct capability

DOCA Flags:
  -h, --help                       Print a help synopsis
  -v, --version                    Print program version information
  -l, --log-level                  Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                  Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                Parse command line flags from an input json file
```

For additional information per sample, use the **-h** option:

```
/tmp/build/doca_<sample_name> -h
/tmp/build/doca_<sample_name> get -h
/tmp/build/doca_<sample_name> list-targets -h
/tmp/build/doca_<sample_name> set -h
```

Most samples follow a common command pattern:

- **list-targets** — enumerate the management targets available on the system (supported by all samples).
- **caps** — check whether the capability is supported by the target (where applicable).
- **get** — read the current configuration.
- **set** — modify the configuration.

The exact set of commands and parameters varies per sample (for example, **mgmt_raw** groups its operations under the **register** and **hca-cap** sub-commands). See the per-sample sections below for each sample's specific commands, parameters, and examples.

5.4.22.7.2 Listing Management Targets

All the samples share a common **list-targets** command that enumerates the management targets available on the system and prints their target strings. A target string printed by **list-targets** can be passed directly to a sample's **-d** / **--device** parameter (device target) or **-r** / **--rep** parameter (representor target).

```
$ doca_mgmt_<sample> list-targets [-v/--verbose]
```

By default only the target strings are printed. Pass **-v** / **--verbose** to additionally print each target's attributes. In samples that manage representors, each device's representor targets are printed nested beneath it.

The device and representor target string formats are described in Management Contexts and Targets.

5.4.22.7.2.1 Examples

- List the device targets (target strings only):

```
$ doca_mgmt_diagnostics_data list-targets
pci/0000:08:00.0
pci/0000:08:00.1
```

- List the device targets and their representors, with all attributes:

```
$ doca_mgmt_data_direct list-targets --verbose
pci/0000:08:00.0
    pci_func_type       PF
    native_port_num     1
    iface_name          eth2
    ibdev_name          mlx5_0
    pci/0000:08:00.0,c0pf0vf0
        pci_func_type       VF
        controller_num      0
        pf_index            0
        vf_index            0
        vhca_id             2
        iface_name          eth4
```

5.4.22.7.3 Samples

The samples are also available on GitHub.

5.4.22.7.3.1 Management Data Direct

The NVIDIA® ConnectX®-8 device may expose a side DMA engine as an additional PCIe PF called the Data Direct device. This additional device allows access to data buffers through multiple PCIe data path interfaces. By default, the *data direct* capability is disabled for VFs and SFs and must be explicitly enabled to allow data direct usage for a specific VF or SF.

This sample demonstrates how to set and get the data direct capability for VFs and SFs using the DOCA Management library.

The sample supports three commands:

- **list-targets** – Lists the available management targets (see Listing Management Targets).
- **get** – Displays the current data direct capability state.
- **set** – Enables or disables the data direct capability.

For **get** and **set** commands, the device (VF or SF) to operate on must be specified.

Specifying the Device

- For VFs:

a. Specify the VF representor using the **-r** or **--rep** parameter with a VF representor target string. For example:

```
pci/0000:08:00.0,c0pf0vf0
```

b. Specify the VF PCIe address (e.g., **0000:08:00.2**) using the **-v** or **--vf-pci-addr** parameter.

- For SFs, specify the SF representor using the **-r** or **--rep** parameter with a SF representor target string. For example:

```
pci/0000:08:00.0,c0pf0sf88
```

Set Command

The **-e** or **--enabled** parameter must be specified with **true** or **false** values to indicate whether the data direct capability should be enabled or disabled, respectively.

> To get or set data direct, the VF or SF must have a representor, i.e., their parent PF must be in switchdev mode.

> If the device port type is InfiniBand, which does not support representors, it is still possible to configure data direct for a VF using the **-v** / **--vf-pci-addr** parameter.

> To set data direct, the VF or SF must be uninitialized (not bound to the **mlx5_core** driver).

> Data direct remains enabled for a VF even after it is destroyed and re-created, so it must be explicitly disabled when it is no longer needed for the VF.

> For command-line syntax details of device and representor identifiers, refer to the DOCA Arg Parser documentation.

Example Executions

- Get *data direct* state for a VF with PCIe address **0000:08:00.2** :

```
$ /tmp/build/doca_mgmt_data_direct get --vf-pci-addr 0000:08:00.2
```

- Enable data direct for a VF (uninitialized):

```
$ /tmp/build/doca_mgmt_data_direct set --vf-pci-addr 0000:08:00.2 --enabled true
```

- Disable data direct for a VF (uninitialized):

```
$ /tmp/build/doca_mgmt_data_direct set --vf-pci-addr 0000:08:00.2 --enabled false
```

- Enable data direct using VF representor:

```
$ /tmp/build/doca_mgmt_data_direct set --rep pci/0000:08:00.0,c0pf0vf0 --enabled true
```

- Enable data direct for SF representor:

```
$ /tmp/build/doca_mgmt_data_direct set --rep pci/0000:08:00.0,c0pf0sf88 --enabled true
```

- List the available targets:

```
$ /tmp/build/doca_mgmt_data_direct list-targets
```

Sample Logic

- Get Data Direct:
  a. Creating a DOCA management device representor context for the target VF/SF: either from the representor target string or from the VF PCI address.
  b. Creating a device caps general handle.
  c. Executing get operation through the DOCA management device caps general API.
  d. Retrieving the data direct capability for the specified representor interface.
  e. Cleaning up all DOCA management and device structures.
- Set Data Direct:
  a. Creating a DOCA management device representor context for the target VF/SF: either from the representor target string or from the VF PCI address.
  b. Creating a device caps general handle and setting data direct attribute.
  c. Executing set operation through the DOCA management device caps general API.
  d. Cleaning up all DOCA management and device structures.

Reference:

- **/opt/mellanox/doca/samples/doca_mgmt/mgmt_data_direct/mgmt_data_direct_sample.c**
- **/opt/mellanox/doca/samples/doca_mgmt/mgmt_data_direct/mgmt_data_direct_main.c**
- **/opt/mellanox/doca/samples/doca_mgmt/mgmt_data_direct/meson.build**

5.4.22.7.3.2 Management Congestion Control Global Status

This sample demonstrates how to set and get congestion control global status for different priorities and protocols (**NP** / **RP**) on a DOCA device using the DOCA Management library.

The sample supports three commands:

- **list-targets** – Lists the available management targets (see Listing Management Targets).
- **get** – Shows the current congestion control global status.
- **set** – Enables or disables the congestion control global status.

For **get** and **set** , the device to operate on must be specified using the **-d** / **--device** parameter with a device target string. The following parameters are also required:

- **-p** / **--priority** – The congestion control priority (0–7).
- **-t** / **--protocol** – The congestion control protocol, **rp** or **np** .

- **-e** / **--enabled** – For **set**, specified with **true** or **false** to enable or disable the congestion control, respectively.

Example Executions

- List the available targets:

```
$ /tmp/build/doca_mgmt_cc_global_status list-targets
```

- Get the congestion control global status for a device, priority 3, protocol **np**:

```
$ /tmp/build/doca_mgmt_cc_global_status get --device pci/0000:08:00.0 --priority 3 --protocol np
```

- Enable the congestion control global status for a device, priority 0, protocol **rp**:

```
$ /tmp/build/doca_mgmt_cc_global_status set --device pci/0000:08:00.0 --priority 0 --protocol rp --enabled
true
```

The sample logic includes two main operations:

- Get Congestion Control Global Status:
  a. Creating a DOCA management device context from the target string that was specified in the command line arguments.
  b. Creating a congestion control global status handle and setting priority and protocol attributes.
  c. Executing get operation through the DOCA management congestion control global status API.
  d. Retrieving the congestion control global status enabled attribute for the specified priority and protocol.
  e. Cleaning up all DOCA management and device structures.
- Set Congestion Control Global Status:
  a. Creating a DOCA management device context from the target string that was specified in the command line arguments
  b. Creating a congestion control global status handle and setting priority, protocol and enabled attributes.
  c. Executing set operation through the DOCA management congestion control global status API.
  d. Cleaning up all DOCA management and device structures.

Reference:

- **/opt/mellanox/doca/samples/doca_mgmt/mgmt_cc_global_status/mgmt_cc_global_status_sample.c**
- **/opt/mellanox/doca/samples/doca_mgmt/mgmt_cc_global_status/mgmt_cc_global_status_main.c**
- **/opt/mellanox/doca/samples/doca_mgmt/mgmt_cc_global_status/meson.build**

5.4.22.7.3.3 Management ICM Quota

This sample allows getting, setting, and querying capabilities for ICM quota on a given device or device representor (PF/VF/SF), and demonstrates how to do it using DOCA management library API.

The sample supports four commands:

- **list-targets** – Lists the available management targets (see Listing Management Targets).
- **caps** – Shows the ICM quota capabilities for the device.
- **get** – Shows the current ICM quota configuration (limit, current allocation, max reached allocation).
- **set** – Sets the ICM quota limit or resets the max reached allocation counter.

For **caps**, **get**, and **set**, either a device or a device representor must be specified. The **-d** / **--device** and **-r** / **--rep** parameters are mutually exclusive.

Specifying the Device/Device Representor

- Specify a device using the **-d** or **--device** parameter with a device target string. Example for a device with PCIe address **0000:08:00.0**:

```
pci/0000:08:00.0
```

- Specify a device representor using the **-r** or **--rep** parameter with a representor target string:
  a. Example for a VF:

```
pci/0000:08:00.0,c0pf0vf0
```

  b. Example for a SF:

```
pci/0000:08:00.0,c0pf0sf88
```

Caps Command

Shows whether the device supports ICM quota and, if so, also shows the maximum ICM quota limit that can be set by the **set** command. The capabilities reported for a device also apply to its representors.

Get Command

The following optional parameters can be specified to retrieve specific attributes:

- **--limit** shows the ICM quota limit that is configured for the device.
- **--cur-alloc** shows the currently allocated ICM quota of the device.
- **--max-reached** shows the maximum reached ICM quota that the device has reached so far.

If no parameters are specified, all attributes are retrieved.

Set Command

At least one of the following parameters must be specified:

- **-L** or **--limit** sets the ICM quota limit for the device (e.g., 4096, 4K, 1M, 1G, 1T, unlimited). The value must be aligned to 4K. The value must be less than or equal to the maximum ICM quota limit reported by **caps** command and less than or equal to 16TB-8KB. Value of 'unlimited' indicates no limit.
- **--reset-max-reached** resets the maximum reached ICM quota counter.

Example Executions

- Get ICM quota capabilities for a device:

```
$ /tmp/build/doca_mgmt_icm_quota caps --device pci/0000:08:00.0
```

- Get all ICM quota attributes for a device:

```
$ /tmp/build/doca_mgmt_icm_quota get --device pci/0000:08:00.0
```

- Get only the ICM quota limit for a device:

```
$ /tmp/build/doca_mgmt_icm_quota get --device pci/0000:08:00.0 --limit
```

- Set ICM quota limit to 1GB for a VF:

```
$ /tmp/build/doca_mgmt_icm_quota set --rep pci/0000:08:00.0,c0pf0vf0 --limit 1G
```

- Set ICM quota limit to unlimited for a device:

```
$ /tmp/build/doca_mgmt_icm_quota set --device pci/0000:08:00.0 --limit unlimited
```

- Reset the ICM quota max reached counter for a SF:

```
$ /tmp/build/doca_mgmt_icm_quota set --rep pci/0000:08:00.0,c0pf0sf88 --reset-max-reached
```

- List the available targets:

```
$ /tmp/build/doca_mgmt_icm_quota list-targets
```

Sample Logic

- Get ICM Quota:
  a. Creating a DOCA management device context from the device target string, or a DOCA management device representor context from the representor target string.
  b. Creating an ICM quota handle.
  c. Executing get operation through the DOCA management ICM quota API.
  d. Retrieving and displaying the requested attributes (limit, current allocation, max reached).
  e. Cleaning up all DOCA management and device structures.
- Set ICM Quota:
  a. Creating a DOCA management device context from the device target string, or a DOCA management device representor context from the representor target string.

b. Creating an ICM quota handle and setting the requested attributes (limit, reset max reached).
c. Executing set operation through the DOCA management ICM quota API.
d. Cleaning up all DOCA management and device structures.

- Get ICM Quota Capabilities:
  a. Creating a DOCA management device context from the device target string.
  b. Checking if ICM quota is supported on the device.
  c. Retrieving and displaying the maximum ICM quota limit supported by the device.
  d. Cleaning up all DOCA management and device structures.

Reference:

- **/opt/mellanox/doca/samples/doca_mgmt/mgmt_icm_quota/mgmt_icm_quota_sample.c**
- **/opt/mellanox/doca/samples/doca_mgmt/mgmt_icm_quota/mgmt_icm_quota_main.c**
- **/opt/mellanox/doca/samples/doca_mgmt/mgmt_icm_quota/meson.build**

5.4.22.7.3.4 Management Diagnostics Data

This sample allows you to get, set, and query capabilities for diagnostics data on a given device (PF), and demonstrates how to execute this using the DOCA Management API.

The sample supports four commands (a device must be specified for **caps**, **get**, and **set**):

- **list-targets** – Lists the available management targets (see Listing Management Targets).
- **caps** – Shows the diagnostics data capabilities for the device.
- **get** – Shows the current diagnostics data configuration (**multi_domain**).
- **set** – Sets the diagnostics data **multi_domain** configuration.

Specifying the Device

Specify a device using the **-d** or **--device** parameter with a device target string. Example for a device with PCIe address **0000:08:00.0**:

```
pci/0000:08:00.0
```

For **set**, the **--multi-domain** parameter must also be specified with **true** or **false** to enable or disable multi-domain mode, respectively.

> Changing **multi_domain** may be rejected by the device if any of the relevant diagnostics data domains of the device has ownership; in that case an appropriate error is returned.

Example Executions

- Get diagnostics data capabilities for a device:

```
$ /tmp/build/doca_mgmt_diagnostics_data caps --device pci/0000:08:00.0
```

- Get diagnostics data **multi_domain** setting for a device:

```
$ /tmp/build/doca_mgmt_diagnostics_data get --device pci/0000:08:00.0
```

- Enable multi-domain for a device:

```
$ /tmp/build/doca_mgmt_diagnostics_data set --device pci/0000:08:00.0 --multi-domain true
```

- Disable multi-domain for a device:

```
$ /tmp/build/doca_mgmt_diagnostics_data set --device pci/0000:08:00.0 --multi-domain false
```

- List the available targets:

```
$ /tmp/build/doca_mgmt_diagnostics_data list-targets
```

Sample Logic

- Get diagnostics data **multi_domain**:
  a. Creating a DOCA management device context from the target string.
  b. Creating a diagnostics data handle.
  c. Executing get operation through the DOCA management diagnostics data API.
  d. Retrieving and displaying the **multi_domain** attribute.
  e. Cleaning up all DOCA management and device structures.
- Set diagnostics data **multi_domain**:
  a. Creating a DOCA management device context from the target string.
  b. Creating a diagnostics data handle and setting the **multi_domain** attribute.
  c. Executing set operation through the DOCA management diagnostics data API.
  d. Cleaning up all DOCA management and device structures.
- Get diagnostics data capabilities:
  a. Creating a DOCA management device context from the target string.
  b. Checking if diagnostics data is supported on the device.
  c. Cleaning up all DOCA management and device structures.

Reference:

- **/opt/mellanox/doca/samples/doca_mgmt/mgmt_diagnostics_data/mgmt_diagnostics_data_sample.c**
- **/opt/mellanox/doca/samples/doca_mgmt/mgmt_diagnostics_data/mgmt_diagnostics_data_main.c**
- **/opt/mellanox/doca/samples/doca_mgmt/mgmt_diagnostics_data/meson.build**

5.4.22.7.3.5 Management Raw

This sample demonstrates how to use the DOCA Management raw APIs, which provide a low-level interface for device configuration intended for advanced users.

The sample illustrates how to use the raw register API to list the supported registers of a device, describe the fields of a register, query a register, and modify a register. It also illustrates how to use the raw HCA capability API to list the supported HCA capability types of a device, describe the fields of an HCA capability type, query HCA capabilities, and modify HCA capabilities.

⚠ A detailed explanation of each register and its fields, and of each HCA capability type and its fields, is outside the scope of this API and may be obtained separately.

In addition to the **list-targets** command (see Listing Management Targets), the sample provides two commands, **register** and **hca-cap**, each with four sub-commands. For all commands, the device must be specified using the **-d** / **--device** parameter with a device target string (e.g., **pci/0000:08:00.0**).

Raw Register

The **register** command has four sub-commands:

- **list** – List the available registers for the device.
- **describe** – List the available fields for a given register.
- **get** – Query a register. Returns all fields if output fields are not specified, or only the requested fields.
- **set** – Modify a register with the given input fields.

For **describe**, **get**, and **set**, the register name must be specified using **-r** / **--register-name**. For **get**, optional input fields (**-i** / **--input-fields**, in the format **field_name1=value1,field_name2=value2,...**) and output fields (**-o** / **--output-fields**, in the format **field_name1,field_name2,...**) can be specified. For **set**, input fields (**-i** / **--input-fields**) are required.

Examples

- List the available registers of a device:

```
$ doca_mgmt_raw register list --device pci/0000:08:00.0
```

- Describe the fields of the **VHCA_ICM_CTRL** register:

```
$ doca_mgmt_raw register describe --device pci/0000:08:00.0 --register-name VHCA_ICM_CTRL
```

- Get specific fields of the **VHCA_ICM_CTRL** register with input parameters:

```
$ doca_mgmt_raw register get --device pci/0000:08:00.0 --register-name VHCA_ICM_CTRL --input-fields
vhca_id=2,vhca_id_valid=1 --output-fields cur_alloc_icm
```

- Set fields of the **VHCA_ICM_CTRL** register:

```
$ doca_mgmt_raw register set --device pci/0000:08:00.0 --register-name VHCA_ICM_CTRL --input-fields
max_alloc_icm_th_mask=1,max_alloc_icm_th=0xffffffff
```

Raw HCA Cap

The **hca-cap** command has four sub-commands:

- **list** – List the available HCA capability types for the device.
- **describe** – List the available fields for a given HCA capability type.

- **get** – Query HCA capabilities. Returns all fields if output fields are not specified, or only the requested fields.
- **set** – Modify HCA capabilities with the given input fields.

For **describe**, **get**, and **set**, the HCA capability type must be specified using **-c** / **--cap-type**. For **get**, the capability mode **-m** / **--cap-mode** (**max** or **current**) is required, and the target function may optionally be selected using **-f** / **--function-id** or **-v** / **--vhca-id** (mutually exclusive; if neither is set, the target is the device specified by **-d** / **--device**). For **set**, the target function (**-f** / **--function-id** or **-v** / **--vhca-id**) and input fields (**-i** / **--input-fields**) are required. The VHCA ID of a function can be retrieved using the **list-targets** command (see Listing Management Targets).

HCA capabilities can be set only on a function that has not been initialized yet. For example, the function must not be bound to the **mlx5_core** driver when setting its HCA capabilities.

Examples

- List the available HCA capability types of a device:

```
$ doca_mgmt_raw hca-cap list --device pci/0000:08:00.0
```

- Get all fields of the **cmd_hca_cap** HCA capability type in max capability mode:

```
$ doca_mgmt_raw hca-cap get --device pci/0000:08:00.0 --cap-type cmd_hca_cap --cap-mode max
```

- Get specific fields of the **cmd_hca_cap** HCA capability type of another function specified by its VHCA ID:

```
$ doca_mgmt_raw hca-cap get --device pci/0000:08:00.0 --cap-type cmd_hca_cap --cap-mode max --vhca-id 2 --
output-fields vhca_id,log_max_qp
```

- Set fields of the **cmd_hca_cap** HCA capability type of another function specified by its VHCA ID:

```
$ doca_mgmt_raw hca-cap set --device pci/0000:08:00.0 --cap-type cmd_hca_cap --vhca-id 2 --input-fields
selective_repeat=0x0
```

Sample Logic

- List Registers Operation:
  a. Creating a DOCA management device context from the target string.
  b. Retrieving the list of supported register names via the DOCA management raw register API.
  c. Displaying the register names and cleaning up.
- Describe Register Operation:
  a. Creating a DOCA management device context from the target string.
  b. Retrieving the list of field descriptors for the specified register via the DOCA management raw register API.
  c. Displaying the fields and cleaning up.

- Get Register Operation:
    a. Creating a DOCA management device context from the target string.
    b. Parsing optional input fields and optional output fields from the command line.
    c. If output fields were specified, querying only those fields; otherwise querying all fields via the DOCA management raw register API.
    d. Displaying the results and cleaning up.
- Set Register Operation:
    a. Creating a DOCA management device context from the target string.
    b. Parsing input fields from the command line.
    c. Modifying the register with the given fields via the DOCA management raw register API.
    d. Cleaning up.
- List HCA Capability Types Operation:
    a. Creating a DOCA management device context from the target string.
    b. Retrieving the list of supported HCA capability types via the DOCA management raw HCA cap API.
    c. Displaying the HCA capability types and cleaning up.
- Describe HCA Capability Operation:
    a. Creating a DOCA management device context from the target string.
    b. Retrieving the list of field descriptors for the specified HCA capability type via the DOCA management raw HCA cap API.
    c. Displaying the fields and cleaning up.
- Get HCA Capability Operation:
    a. Creating a DOCA management device context from the target string.
    b. Parsing optional output fields from the command line.
    c. If output fields were specified, querying only those fields; otherwise querying all fields via the DOCA management raw HCA cap API.
    d. Displaying the results and cleaning up.
- Set HCA Capability Operation:
    a. Creating a DOCA management device context from the target string.
    b. Parsing input fields from the command line.
    c. Modifying the HCA capabilities with the given fields via the DOCA management raw HCA cap API.
    d. Cleaning up.

Reference:

- **/opt/mellanox/doca/samples/doca_mgmt/mgmt_raw/mgmt_raw_sample.c**
- **/opt/mellanox/doca/samples/doca_mgmt/mgmt_raw/mgmt_raw_main.c**
- **/opt/mellanox/doca/samples/doca_mgmt/mgmt_raw/meson.build**

5.5 DOCA Utils

This section includes modules that may be used by application developers to speed up their development process.

This section contains the following pages:

- DOCA Arg Parser

5.5.1 DOCA Arg Parser

This guide provides an overview and configuration instructions for DOCA Arg Parser API.

5.5.1.1 Introduction

The DOCA Arg Parser module simplifies the creation of command-line interfaces for user-facing applications, as well as provides basic support for JSON-based configuration files. It allows developers to define and handle structured program arguments while automatically generating help and usage messages.

When a user provides invalid input, the Arg Parser outputs clear error messages along with the corresponding usage instructions, helping ensure a smooth user experience.

- The Arg Parser validates a wide range of input errors, including unsupported arguments and incorrect types.
- Upon encountering an error, the module:
    a. Prints a descriptive error message.
    b. Displays the full usage guide for the program.
    c. Exits the application with an appropriate error code.

5.5.1.2 API

For the library API reference, refer to ARGP API documentation in DOCA API References.

> ⚠ The pkg-config (***.pc** file) for the Arg Parser library is **doca-argp**.

The following sections provide additional details about the library API.

5.5.1.2.1 doca_argp_param

The data structure contains the program flag details needed to process DOCA ARGP. These details are used to generate usage information for the flag, identify if the user passed the flag in the command line and notify the program about the flag's value.

```
struct doca_argp_param;
```

5.5.1.2.2 doca_argp_param_create

Creates a DOCA ARGP parameter instance. The user is required to update the param attributes by calling the respective setter functions and registering the param by calling **doca_argp_register_param()**.

```
doca_error_t doca_argp_param_create(struct doca_argp_param **param);
```

- **param [out]** – DOCA ARGP param structure with unset attributes

5.5.1.2.3 doca_argp_register_param

Calling this function registers the program flags in the Arg Parser database. The registration includes flag details. Those details are used to parse the input arguments and generate usage print.

> ⚠ The user must register all program flags before calling **doca_argp_start()**.

```
doca_error_t doca_argp_register_param(struct doca_argp_param *input_param);
```

- **input_param [in]** – program flag details

> ⚠ Once registered, the ownership of the parameter passes to the ARGP module, and it should no longer be accessed/used by the user.

5.5.1.2.4 doca_argp_cmd

The data structure contains the program command details needed to process DOCA ARGP. These details are used to generate usage information for the command, identify if the user passed the command in the command line, and notify the program about the selected command.

```
struct doca_argp_cmd;
```

5.5.1.2.5 doca_argp_cmd_create

Creates a DOCA ARGP command instance. The user is required to update the command attributes by calling the respective setter functions and registering the command by calling **doca_argp_register_cmd()**.

```
doca_error_t doca_argp_cmd_create(struct doca_argp_cmd **cmd);
```

- **cmd [out]** – DOCA ARGP command structure with unset attributes

5.5.1.2.6 doca_argp_cmd_register_param

Calling this function registers the program flags in the Arg Parser database as associated with the respective command. The registration includes flag details which are used to parse the input arguments and generate usage print.

```
doca_error_t doca_argp_cmd_register_param(struct doca_argp_cmd *cmd, struct doca_argp_param *input_param);
```

- **cmd [in]** – program cmd
- **input_param [in]** – program flag details

⚠ Once registered, the ownership of the parameter passes to the ARGP module, and it should no longer be accessed/used by the user.

5.5.1.2.7 doca_argp_cmd_register_cmd

Calling this function registers the program cmd as a sub-command to the respective associated command. The registration includes cmd details which are used to parse the input arguments and generate usage print.

```
doca_error_t doca_argp_cmd_register_cmd(struct doca_argp_cmd *cmd, struct doca_argp_cmd *input_cmd);
```

- **cmd [in]** – program cmd
- **input_cmd [in]** – program cmd details (sub-command)

⚠ Once registered, the ownership of the sub-command passes to the ARGP module, and it should no longer be accessed/used by the user.

5.5.1.2.8 doca_argp_register_cmd

Calling this function registers the program command in the Arg Parser database. The registration includes command details which are used to parse the input arguments and generate usage print.

⛔ The user must register all program commands before calling **doca_argp_start()**.

```
doca_error_t doca_argp_register_cmd(struct doca_argp_cmd *input_cmd);
```

- **input_cmd [in]** – program command details

⚠ Once registered, the ownership of the command passes to the ARGP module, and it should no longer be accessed/used by the user.

5.5.1.2.9 doca_argp_set_dpdk_program

Marks the programs as a DPDK program. Once ARGP is finished with the parsing, DPDK (EAL) flags are forwarded to the program by calling the given callback function.

```
void doca_argp_set_dpdk_program(dpdk_callback callback);
```

- **callback [in]** - callback function to handle DPDK (EAL) flags.

⚠ Due to a current limitation, programs that support several commands cannot be marked as "DPDK Programs".

5.5.1.2.10 doca_argp_init

This function initializes the ARGP module and must be the first function called when using this module. Initialization includes the basic information about the program and about the user's context as will later be used during the parsing of the command-line.

```
doca_error_t doca_argp_init(const char *program_name, void *program_config);
```

- **program_name [in]** – program name as will be used in the help printout
- **program_config [in]** – pointer to the user's configuration struct (if there is one)

5.5.1.2.11 doca_argp_start

The **doca_argp_start()** function initiates the parsing and classification of command-line arguments. It maps user-provided arguments to their corresponding user-defined parameters and, if present, transfers DPDK EAL flags back to the program by invoking the registered callback.

```
doca_error_t doca_argp_start(int argc, char **argv);
```

- **argc [in]** – the number of input arguments
- **argv [in]** – the array of input strings representing the command-line arguments

5.5.1.2.12 doca_argp_destroy

Cleans up all the resources used by the module. Should be invoked upon program termination following a successful invocation of **doca_argp_init()**.

```
doca_error_t doca_argp_destroy(void);
```

5.5.1.3 JSON Syntax

DOCA Arg Parser has built-in support for basic JSON configuration files, which is activated when using the **j** / **json** CLI flags.

Important nodes:

- The key names in the JSON file are the long flag names as were registered on DOCA Arg Parser's API.
- A flag can be passed to both CLI and JSON file, in which case the CLI value will override that found in the provided JSON file.

5.5.1.4 DOCA General Flags

The following table lists the supported DOCA general flags:

Short Flag	Long Flag	Description
-h	**--help**	Prints a help synopsis and exits

Short Flag	Long Flag	Description
`-v`	`--version`	Prints program version information and exits
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

5.5.1.5 DOCA Program Flags

The flags for each program can be found in the document dedicated to that program, including instructions on how to run it using the command-line interface.

5.5.1.6 DOCA Program Commands

While simple programs might only use a single command with some program flags, DOCA ARGP supports the registration of a command tree so as to provide support for more advanced programs. Using this mechanism, program flags could be associated directly with their respective commands, allowing the user to access them only when needed.

When existing, the commands for each program can be found in the document dedicated to that program, including instructions on how to run it.

5.5.1.6.1 Detailed Example of Program Commands

Assuming there is a dummy program named `doca_test` with the following commands and parameters:

- Command – `info` – Show extended information about a given category
  - Command – `devices` – Show information about all devices, across all PCIe addresses
    - Parameter – `--pci-address` – Limit the command to the provided PCIe address (optional)
- Command – `ping` – Ping a given remote network address
  - Parameter – `--network-address` – Target network address for the ping (mandatory)

    - Parameter – **--payload-length** – Length of the payload for the ping command (optional; default is 100 bytes)
- Parameter – **--output** – Output file to save the log messages into (optional; inherited)

Optional invocations:

- Show information about all devices:

```
doca_test info devices
```

- Show information about all devices under a specific PCIe address and store output to **/tmp/log.txt**:

```
doca_test info devices --pci-address 00:03:00.0 --output /tmp/log.txt
```

- Ping a remote machine at **8.8.8.8** and store output to **/tmp/log.txt**:

```
doca_test ping --network-address 8.8.8.8 --output /tmp/log.txt
```

> The **--output** parameter is inherited. That is, it is registered once and serves all commands at the relevant command level, including all their sub-commands (**info**, **info devices**, **ping**).

5.5.1.7 Syntax – Device and Device Representor

In addition to the primitive built-in types supported by DOCA Argp, the module also provides support for the following advanced argument types:

- DOCA Device – **DOCA_ARGP_TYPE_DEVICE**
- DOCA Device Representor – **DOCA_ARGP_TYPE_DEVICE_REP**

Support for these advanced types includes the following logic:

1. Parsing the string-based identifier provided via the command line.
2. Locating the requested device or representor based on the parsed identifier.
3. Passing the resolved device object to the user's callback using the appropriate context type:
    - Device: **struct doca_argp_device_ctx**
    - Device Representor: **struct doca_argp_device_rep_ctx**

The accepted syntax for these types is composed of several identifier categories, each of which is detailed in the sections below.

> To view a list of currently available devices and representors, it is recommended to use the DOCA Caps utility:
>
> - **--list-devs** – Lists all available DOCA devices
> - **--list-rep-devs** – Lists all available DOCA representors

The output from this tool provides the necessary information to construct valid identifiers for all supported categories described in the following subsections.

5.5.1.7.1 JSON Syntax

The JSON syntax for the devices and representors is as follows.:

```
"device-flag": {
    "type": "pci",
    "identifier": "08:00.0",
}
...
"device-rep-flag": {
    "type": "pci",
    "identifier": "08:00.0,pf0vf1",
    "devargs": "dv_flow_en=2",
}
```

The example above uses "device-flag" as the long flag name for the device type parameter, and "device-rep-flag" as the long name for the device representor type parameter.

⚠ The "devargs" subkey is supported by both device and device representor types and is optional.

5.5.1.7.2 PCIe Identifiers

5.5.1.7.2.1 Devices

- Syntax:

```
pci/<PCI device identifier>[,<device args>]
```

- Examples:
  - `pci/08:00.0`
  - `pci/0000:03:00.1,dv_flow_en=2`

This identifier category is used to identify devices based on their PCIe device identifiers, following the standard Bus-Device-Function (BDF) notation.

5.5.1.7.2.2 Representors

- Syntax:

```
pci/<PCI device identifier>,<representor identifiers>[,<device args>]
```

- Examples:
  - `pci/08:00.0,pf0vf1` - The representor of the second VF (indices start from 0) under PF0, and associated to device with PCI address of `08:00.0`
  - `pci/0000:03:00.0,c1pf[0-1]vf[2-3,6]sf2,dv_flow_en=2` - Multiple representors under the respective PF0 and PF1, associated with the host (controller) with index 1:
    - VF2
    - VF3,
    - VF6

- SF2

> ⚠ If PF1 is to be used as a representor under PF0 as the eswitch manager, the above representors string is **not** enough, and an **additional** argument should be passed so to indicate PF1 as a representor. For example: `pci/08:00.1`

> ⚠ If only a single representor is expected by the program, it is the developer's responsibility to ensure that the user did not pass multiple representors.

- `pci/03:00.0,pf0vf-1` - The host representor of PF0
- `pci/03:00.0` - The uplink representor of the device with the given PCI address.

Aside from the exceptions for the host and uplink representors, the syntax follows the scheme `pci/BDF,c#pf#vf#sf#`, where:

- `c#` – *(Optional)* Specifies the host/controller index
- `pf#` – *(Mandatory)* Specifies the index or indices of the relevant PFs
- `vf#` – *(Optional)* Specifies the index or indices of the relevant VFs
- `sf#` – *(Optional)* Specifies the index or indices of the relevant SFs

Each identifier may be expressed as:

- A single numeric value (e.g., `3`)
- A set of values or ranges (e.g., `[2,4-8]`), allowing for compact specification of multiple instances.

5.5.1.7.3 InfiniBand (IB) Identifiers

5.5.1.7.3.1 Devices

- Syntax:

```
ib/<IB-device-identifier>[,<device args>]
```

- Examples:
  - `ib/mlx5_3`
  - `ib/mlx5_4,dv_flow_en=2`

This identifier category is used to identify devices based on their InfiniBand device name.

5.5.1.7.3.2 Representors

- Syntax:

```
ib/<IB-device-identifier>:<port identifier>[,<device args>]
```

- Examples:

- **ib/mlx5_0:2**
- **ib/mlx5_1:3,dv_flow_en=2**

Since representors are identified by their port association within the scope of a given InfiniBand (IB) device, this identifier format includes both the IB device name and the associated port number.

5.5.1.7.4 Network Interface Names

- Syntax:

```
ifname/<interface-name>[,<device args>]
```

- Examples:
  - **ifname/eth2**
  - **ifname/eth3,dv_flow_en=2**

This identifier category supports all device and representor types and identifies them based on their network interface name.

5.5.1.7.5 Network Interface Indices

- Syntax:

```
ifindex/<interface-index>[,<device args>]
```

- Examples:
  - **ifindex/3**
  - **ifindex/4,dv_flow_en=2**

This identifier category supports all device and representor types and identifies them based on their network interface index.

5.5.1.7.6 Auxiliary Device Indices

- Syntax:

```
aux/<aux-device-index>[,<device args>]
```

- Examples:
  - **aux/3**
  - **aux/4,dv_flow_en=2**

This identifier category is primarily intended for identifying SFs, which are defined as auxiliary devices in PCIe terminology.

> The auxiliary device syntax applies only to actual devices; it is not applicable to representor devices, as representors do not have an associated auxiliary index.

5.6 DOCA Drivers

This section describes underlying drivers included in DOCA and includes the following pages:

- DOCA UCX
- MLX Drivers and Technologies

5.6.1 DOCA UCX

This guide provides instructions for developing applications on top of the UCX library.

5.6.1.1 Introduction

Unified Communication X (UCX) is an optimized point-to-point communication framework.

UCX exposes a set of abstract communication primitives that utilize the best available hardware resources and offloads, such as active messages, tagged send/receive, remote memory read/write, atomic operations, and various synchronization routines. The supported hardware types include RDMA (InfiniBand and RoCE), TCP, GPUs, and shared memory.

UCX facilitates rapid development by providing a high-level API, masking the low-level details, while maintaining high-performance and scalability.

UCX implements best practices for transfer of messages of all sizes, based on the accumulated experience gained from applications running on the world's largest datacenters and supercomputers.

5.6.1.2 Prerequisites

UCX runtime libraries are installed as part of the DOCA installation.

UCX is used the same way from the host and the DPU side.

Any active network device available on the system might be used by UCX, including network devices that might be unreachable to the remote peer.

If one of the destinations is not reachable via a certain network device (e.g., a BlueField cannot reach another BlueField via **tmfifo_net0**), UCX communication may fail.

To resolve this, use the UCX environment variable **UCX_NET_DEVICES** to specify which devices UCX can use. For example:

```
export UCX_NET_DEVICES=enp3s0f0s0,enp3s0f1s0
```

Or:

```
env UCX_NET_DEVICES=enp3s0f0s0,enp3s0f1s0 <UCX-program>
```

Using the command **show_gids** on the BlueField one can obtain the mlx device name and the port of an SF. Then that can be used to limit the UCX network interfaces and allow IB. For example:

```
dpu> show_gids
DEV     PORT    INDEX   GID                                     IPv4            VER  DEV
---     ----    -----   ---                                     ------------    ---  ---
mlx5_2  1       0       fe80:0000:0000:0000:0052:72ff:fe63:1651                 v2   enp3s0f0s0
mlx5_3  1       0       fe80:0000:0000:0000:0032:6bff:fe13:f13a                 v2   enp3s0f1s0
dpu> env UCX_NET_DEVICES=mlx5_2:1,mlx5_3:1 <UCX-program>
```

When RDMACM is not available, it is also required to list the Ethernet devices in **UCX_NET_DEVICES** configuration, so they could be used for TCP-based connection establishment. For example:

```
dpu> env UCX_NET_DEVICES=enp3s0f0s0,enp3s0f1s0,mlx5_2:1,mlx5_3:1 <UCX-program>
```

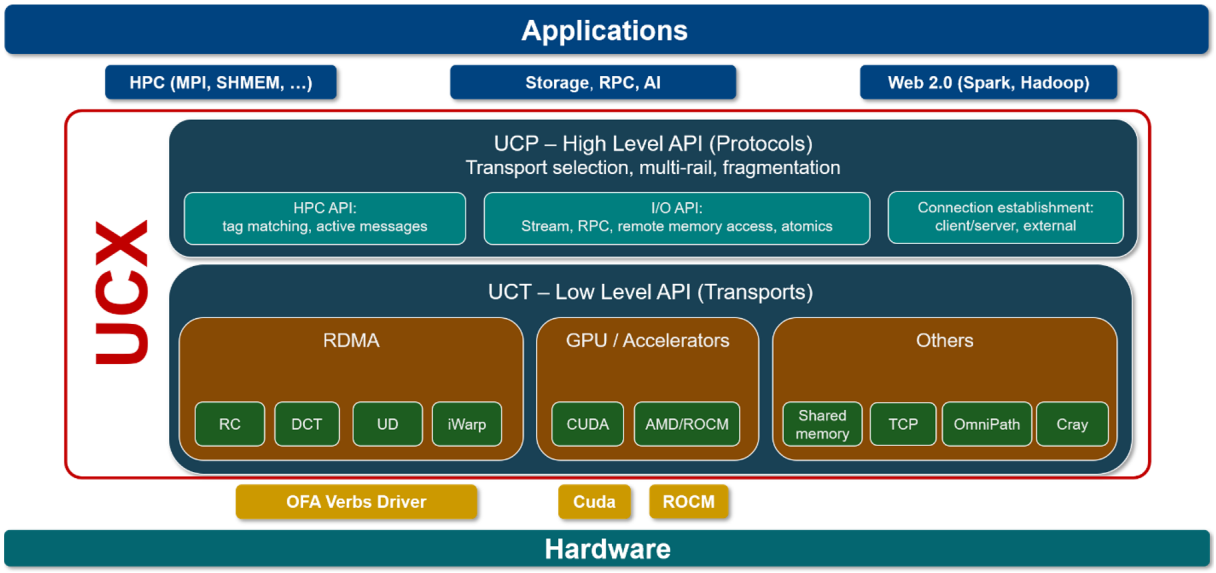
5.6.1.3 Architecture

The following image describes the software layers of UCX middleware.

On the upper layer, various applications that utilize high-speed communications are built on top of the UCX high-level API (UCP).

UCP layer implements the business logic to utilize, combine, and manipulate different transports to achieve the best possible performance for different use cases. This logic decides which transports must be used for each message, which types of basic hardware communication primitives to use, how to fragment messages, etc.

UCT, the transport API, is a hardware abstraction layer that brings different types of communication devices to a common denominator. There are multiple communication primitives defined by UCT API, but each transport service may implement only some of them—preferably the ones that are natively supported by the underlying hardware. UCT users (e.g., UCP) are expected to handle the missing communication primitives defined by UCT API but not implemented by a transport service.



5.6.1.3.1 UCP Objects

This section describes the high-level communication objects that are used by most applications written on top of UCX.

5.6.1.3.1.1 UCP Context (ucp_context_h)

The context is the top-level object and it defines the scope of all other UCX objects. It is possible to create multiple contexts in the same process to have a complete separation of hardware and memory resources.

5.6.1.3.1.2 UCP Worker (ucp_worker_h)

The worker represents a communication state and its associated network resources. It is responsible for sending and processing incoming messages and handling all network-related events. All point-to-point connections are created in the scope of a particular worker.

A worker object can be defined to support usage from multiple threads. However, due to lock contention, the performance is better when a given worker is used most of the time from one thread.

The worker progresses communications either by active polling, waiting for asynchronous events, or a combination of both.

5.6.1.3.1.3 UCP Endpoint (ucp_ep_h)

The endpoint represents a connection from a local worker to a remote worker. That remote worker may be created in any place that is reachable by one of the communication networks supported by UCT layer. That could be, for example, on a different host in the fabric, the same host, on the DPU, or even in the same process.

5.6.1.3.1.4 UCP Listener (ucp_listener_h)

The listener binds to a network port number on the underlying operating system, and dispatches incoming connection requests. The incoming connection request can be used to create a matching endpoint on the server (passive) side or rejected and released.

5.6.1.3.1.5 UCP Request (ucp_request_h)

The request object is created by one of the non-blocking communications primitives in a case where the operation could not be completed immediately in-place. The application is expected to check the request for completion, either by testing it directly, or by associating a custom callback with the request.

5.6.1.4 API

This section describes the main UCX APIs for high-speed communications. For the full reference, refer to UCX API specification.

UCX exposes two kinds of API: the high-level UCP API and the low-level UCT (transport) API. For most applications, it is recommended to use only the UCP API, since it relieves much of the burden of handling each transport's capabilities, limitations, and performance traits.

Many of the APIs accept a structure pointer with a **field_mask** as an argument. This method is used to provide backward ABI/API compatibility: If new function arguments are introduced, they are added as new fields in the struct, so the function signature does not change. In addition, **field_mask** specifies which struct fields are valid from the caller's (user application) perspective. UCX only accesses the fields enabled by this bitmask and uses default values for the remaining struct fields.

Some APIs require passing user-defined callbacks as a method to get notifications about specific events. Unless otherwise specified, such callbacks are called from the context of the **ucp_worker_progress()** call (detailed below), and are expected to complete quickly or defer some of their tasks to another thread (to avoid timeouts and starvation of processing from other network events).

The pkg-config (***.pc** file) for the UCX library is named **ucx** .

The following sections provide additional details about the library API.

5.6.1.4.1 ucs_status_t

An **enum** type that holds all UCX error codes.

5.6.1.4.2 ucp_init

```
ucs_status_t ucp_init(const ucp_params_t *params, const ucp_config_t *config, ucp_context_h *context_p)
```

- **params [in]** – points to a structure with optional parameters. All fields are optional except `features`, which must be set.
- **config [in]** – optional, can be NULL for default behavior. Configuration can be obtained by calling `ucp_config_read()`.

  The supported configuration options can change between UCX versions. The full list can be obtained by running the `ucx_info` CLI tool:
  ucx_info -c -f
- **context_p [out]** – a pointer to a location in memory for the created UCP context

The function returns an error code as defined by **ucs_status_t** .

This function creates a new UCP top-level context and returns it by value in the **context_p** argument.

5.6.1.4.3 ucp_cleanup

```
void ucp_cleanup(ucp_context_h context_p)
```

- **context_p [in]** – a UCP context instance

This function destroys a previously created context. Prior to calling this function, any other resources created on this context (e.g., workers or endpoints) must be destroyed.

5.6.1.4.4 ucp_worker_create

```
ucs_status_t ucp_worker_create(ucp_context_h context, const ucp_worker_params_t *params, ucp_worker_h *worker_p)
```

- **context [in]** – an existing UCP context
- **params [in]** – points to a structure with configuration parameters. All fields are optional. Commonly, only the field **thread_mode** is used. Possible **thread_mode** values are as follows:
  - **UCS_THREAD_MODE_SINGLE** – only one specific thread (typically, the one that created the worker) is used to access the worker and its associated endpoints.
  - **UCS_THREAD_MODE_SERIALIZED** – multiple threads can access the worker and its associated endpoints, but only one at a time. This implies an exclusion mechanism (e.g., locking) implemented in the application. Sometimes, more expensive bus flushing instructions are needed with serialized mode, compared to single thread mode.
  - **UCS_THREAD_MODE_MULTI** – multiple threads can access the worker at any given time. UCX takes care of the locking internally. As of version 1.12, it is implemented as a global lock on the worker.
- **worker_p [out]** – a pointer to a location in memory for the created worker

The function returns an error code as defined by **ucs_status_t**.

This function creates a new UCP worker on a previously created context and returns it by value in the **worker_p** argument.

When **ucp_worker_create()** succeeds, the caller is still expected to check the actual thread mode the worker was created with by calling **ucp_worker_query()** API, and take the necessary actions (for example, report an error or fallback) if the returned thread mode is not as expected to be.

5.6.1.4.5 ucp_worker_destroy

```
void ucp_worker_destroy(ucp_worker_h worker)
```

- **context_p [in]** – an UCP worker instance

This function destroys a previously created worker. Prior to calling this function, all associated endpoints and listeners must be destroyed.

Destroying the worker may cause communication errors on any remote peer that has an open endpoint to this worker. These errors are handled according to that endpoint's error handling configuration (detailed in section ucp_ep_create).

5.6.1.4.6 ucp_listener_create

```
ucs_status_t ucp_listener_create(ucp_worker_h worker, const ucp_listener_params_t *params, ucp_listener_h
*listener_p)
```

- **worker [in]** – an existing UCP worker
- **params [in]** – points to a structure with configuration parameters. The fields **sockaddr** and **conn_handler** are mandatory, but the rest of the fields are optional.
  - **sockaddr** – specifies IPv4/IPv6 address to listen for connections. The semantics are similar to the built-in bind() function. **INADDR_ANY/INADDR6_ANY** can be used to listen on all network interfaces. If the port number is set to 0, a random unused port is selected. The actual port number can be obtained by calling the **ucp_listener_query()** API.
  - **conn_handler** – a callback for handling incoming connection requests along with an associated user-defined argument. The callback type is defined as:

    ```
    void (*ucp_listener_conn_callback_t) (ucp_conn_request_h conn_request, void *arg)
    ```

    Whenever a remote endpoint is created through this listener, this callback is called on the listener side with a new **conn_request** object representing the incoming connection, and the user-defined argument **arg** that is passed to **ucp_listener_create()**.
    The callback is expected to process this connection request by either creating an endpoint for it (pass **conn_request** as a parameter to **ucp_ep_create**, including on a different worker), or rejecting and destroying it (call **ucp_listener_reject**). This does not have to happen immediately. The callback may put the connection request on an internal application queue and process it later.
- **listener_p [out]** – a pointer to a location in memory for the created listener

The function returns an error code as defined by **ucs_status_t**.

This function creates a new listener object to accept incoming connections on a specific network port, and returns it by value in the **listener_p** argument.

5.6.1.4.7 ucp_listener_destroy

```
void ucp_listener_destroy(ucp_listener_h listener_p)
```

- `listener_p [in]` – a listener instance

This function destroys a previously created listener. Prior to calling this function, any connection requests that were reported by **conn_handler** are expected to be processed. Pending connection requests that have not been reported to the application yet, or new connection requests that arrive after this function is called, are rejected.

5.6.1.4.8 ucp_ep_create

```
ucs_status_t ucp_ep_create(ucp_worker_h worker, const ucp_ep_params_t *params, ucp_ep_h *ep_p)
```

- **worker [in]** – an existing UCP worker
- **params [in]** – Points to a structure with configuration parameters. A creation mode field must be set. Other fields are optional. Commonly used fields are described in the following subsections.
- **ep_p [in]** – a pointer to a location in memory for the created endpoint

The function returns an error code as defined by **ucs_status_t**.

This function creates a new connection to a remote peer and returns it by value in the **ep_p** parameter. The new endpoint can be used for communication immediately after it is created, though some operations may be queued internally and sent after the underlying connection is established.

5.6.1.4.8.1 Create Modes (ucp_ep_params_t)

There are three ways the endpoint can be created:

- Client connects to a remote listener
  In this case, the `sockaddr` field specifies the remote IPv4/IPv6 address and port number. The **flags** field must be enabled and must include the **UCP_EP_PARAMS_FLAGS_CLIENT_SERVER** flag. Optionally, from UCX version 1.13 on, the **local_sockaddr** field may be used to specify a local source device address to bind to.
- Server creates an endpoint due to an incoming connection request
  In this case, the **conn_request** field must be set to this connection request. Such endpoint can optionally be created on a different worker, not the same one this connection request was accepted on.
- Create an endpoint to a specific worker address
  In this case, the field **address** must be set to point to a remote worker's address. That address (and its length) must be obtained on the remote side by calling **ucp_worker_query(**) and sent using an application-defined method (e.g., TCP socket, or other existing communication mechanism). The internal structure of the address is opaque and may change in different versions.

5.6.1.4.8.2 User-Defined Error Handling (ucp_ep_params_t)

By default, unexpected errors on the connection (e.g., network disconnection or aborted remote process) generate a fatal failure. To enable graceful error handing, several parameters must be set during endpoint creation:

- The `err_mode` field must be set to **UCP_ERR_HANDLING_MODE_PEER**. This guarantees that send requests are always completed (successfully or error). Otherwise, network errors are considered fatal and abort the application without giving it a chance to perform cleanup or fallback flows.
- **The err_handler.cb** field must be set to a user-defined callback which is called if a connection error occurs. The error handler is defined as follows:

```
void (*ucp_err_handler_cb_t)(void *arg, ucp_ep_h ep, ucs_status_t status)
```

  The callback parameters are the user-defined argument (passed in `user_data`), the endpoint handle on which the error happened, and the error code.
  After this callback, no more communications should be done on the endpoint. The application is expected to close the endpoint.
- The **user_data** field must be set to a user-defined argument passed to the **err_handler** callback

5.6.1.4.9 ucs_status_ptr_t

```
typedef void* ucs_status_ptr_t;
```

This function is commonly used as a return value for non-blocking operations.

The return value of **ucs_status_ptr_t** combines a status code and a request pointer which may be one of the following:

- A NULL pointer indicating that the operation has completed successfully in-place. The user-provided callback, if there is one, is not called.
- An error status, that can be detected by the **UCS_PTR_IS_ERR(status)** macro and extracted by **UCS_PTR_STATUS(status)**.
- Otherwise, the status is a request pointer which can also be detected by the **UCS_PTR_IS_PTR(status)** macro. This means that the communication operation has started (or was queued) but not yet completed. The completion is reported by calling the user-provided callback (in **ucp_request_param_t**) or through an explicit check on the request status by calling **ucp_request_check_status()**.

5.6.1.4.10 ucp_ep_close_nbx

```
ucs_status_ptr_t ucp_ep_close_nbx(ucp_ep_h ep, const ucp_request_param_t *param)
```

- **ep [in]** – an existing UCP endpoint
- **param [in]** – points to a structure that defines how the closing operation is performed. The **flags** field of the **param** structure specifies which method to use to close the endpoint:
  - **UCP_EP_CLOSE_MODE_FORCE** – close the endpoint immediately without attempting to flush outstanding operation. Some requests already completed on the transport level may complete successfully, others may be completed with an error status. In

the latter case, it is not known whether they have reached the destination process or completed there.
Closing an endpoint this way is equivalent to calling **close()** on a TCP socket and can generate a connection error on the remote side. Therefore, to use this mode, both the local and remote endpoints must be created with the **err_mode** parameter set to **UCP_ERR_HANDLING_MODE_PEER**.

- **UCP_EP_CLOSE_MODE_FLUSH** – synchronize with the remote peer and flush outstanding operations. Some operations may be canceled and complete with the status **UCS_ERR_CANCELED**. However, it is guaranteed that they did not complete on the remote peer as well.

The function returns a status pointer to check the operation's status. **NULL** means success.

This function starts the process of closing a previously created endpoint. The function is non-blocking, and the returned value is a status pointer used to indicate when the endpoint is fully destroyed. For more information, refer to section Communications.

5.6.1.4.11 ucp_request_param_t

```
struct ucp_request_param_t {
    uint32_t op_attr_mask;
    uint32_t flags;
    union ucp_request_param_t cb;
    void *user_data;
    ucp_datatype_t datatype;
    /* Some other fields that are rarely used */
    …
}
```

- **op_attr_mask [in]** – mask of enabled fields and several control flags.
- **flags [in]** – operation-specific flags. Each API method defines its own set of flags for this field.
- **cb [in]** – callback for when the operation is completed.
- **user_data [in]** – user-defined argument passed to the completion callback.
- **datatype [in]** – may be used to specify a custom data layout for the data buffer (not **user_data**) that is provided to the communication API. If this parameter is not set, the data buffer is treated as a contiguous byte buffer.

The fields of **ucp_request_param_t** specify several common attributes and flags that are used to control how the communications request is allocated and completed. This is aimed to optimize different use-cases.

5.6.1.4.12 ucp_worker_progress

```
unsigned ucp_worker_progress(ucp_worker_h worker)
```

- **worker [in]** – an existing UCP worker

The function returns a non-zero value if any communication has been progressed. Otherwise, it returns zero.

This function progresses outstanding communications on the worker. This includes polling hardware and shared memory queues, calling callbacks, pushing pending operations to the network devices, advancing the state of complex protocols, progressing connection establishment process, and more.

Though some transports, such as RDMA, offload do much of the heavy lifting, the initiation and completion of communication operations still must be performed explicitly by the process. UCX does not spawn additional progress threads. Instead, it is expected that the upper-layer application spawns its own progress thread, as needed, to call **ucp_worker_progress()**.

This function cannot be used from inside a callback.

5.6.1.4.13 ucp_am_send_nbx

```
ucs_status_ptr_t ucp_am_send_nbx(ucp_ep_h ep, unsigned id, const void *header,
                                 size_t header_length, const void *buffer,
                                 size_t count, const ucp_request_param_t *param)
```

- **ep [in]** – connection to send the active message on. Previously returned from **ucp_ep_create()**.
- **id [in]** – active message identifier. This is an arbitrary 16-bit integer value defined by the application and used to select the active message callback to call on the receiver side. This allows handling different types of messages by different callback functions.
- **header [in]** – pointer to a user-defined header for an active message
- **header_length [in]** – length of the header to send. Usually, the header is small and, in any case, it should be no larger than the **max_am_header** worker attribute, as returned from **ucp_worker_query()**. The header size could vary depending on the available transports and is usually expected to be at least 256 bytes.
- **buffer [in]** – pointer to the active message payload
- **count [in]** – number of elements in the payload buffer. By default, each element is a single byte, so this is the byte-length of the buffer. Other data layouts, such as IO vector (IOV) list, could be specified by **param->datatype**.
- **param [in]** – additional parameters controlling request completion semantics. The relevant field is only **flags** and it can be set to a combination of the following flags:
  - **UCP_AM_SEND_FLAG_REPLY** – force passing **reply_ep** to the callback on the receiver side. This can increase the internal header size and add some overhead.
  - **UCP_AM_SEND_FLAG_EAGER** – force using eager protocol (details below).
  - **UCP_AM_SEND_FLAG_RNDV** – force using rendezvous protocol (details below).

  The active message can be sent either by the eager or rendezvous protocol. Eager protocol means the data buffer is available on the receiver immediately during the callback, while the rendezvous protocol requires fetching the data using an additional call to **ucp_am_recv_data_nbx()**, allowing it to be placed directly to an application-selected buffer. By default, smaller messages are sent via eager protocol, and larger messages use rendezvous protocol. This can be overridden using **UCP_AM_SEND_FLAG_EAGER** or **UCP_AM_SEND_FLAG_RNDV**.

`UCP_AM_SEND_FLAG_EAGER` and `UCP_AM_SEND_FLAG_RNDV` are mutually exclusive.

The function returns a status pointer to check the operation's status. **`NULL`** means success.

This function initiates sending of an active message from the initiator side. As a result, a designated callback (registered by **`ucp_worker_set_am_recv_handler`**) is called on the receiver side to handle this message. The function is non-blocking, so if the send operation is not completed immediately, a request handle is retuned.

5.6.1.4.14 ucp_worker_set_am_recv_handler

```
ucs_status_t ucp_worker_set_am_recv_handler(ucp_worker_h worker, const ucp_am_handler_param_t *param)
```

- **`worker [in]`** – an existing UCP worker.
- **`param [in]`** – set callback configurations. See more below.

The function returns a non-zero value if any communication has been progressed. Otherwise, it returns zero.

This function registers a callback for processing active messages on the given worker.

The following are the mandatory fields to set in **`param`**:

- **`id`** – active message identifier to bind with the registered callback. Callback is invoked when receiving incoming messages with the same ID.
- **`arg`** – a user-defined argument to pass to the active message callback.
- **`cb`** – a user-defined callback to invoke when an active message arrives. The callback is defined as:

```
ucs_status_t (*ucp_am_recv_callback_t)(void *arg, const void *header,
                                       size_t header_length, void *data,
                                       size_t length,
                                       const ucp_am_recv_param_t *param)
```

The following are the parameters passed from UCX to the callback:

- **`arg`** – the same user-defined argument passed to **`ucp_worker_set_am_recv_handler`**.
- **`header`** – points to the active message header as defined by the sender side while sending the active message. The header should be consumed by the callback since it is not valid after the callback returns.
- **`header_length`** – valid size of the buffer pointer by **`header`**.
- **`data`** – pointer to the data or an opaque handle that can be used to fetch the data according to the **`UCP_AM_RECV_ATTR_FLAG_RNDV`** flag in the field **`param->recv_attr`**. When flag is on, this is an opaque handle.
- **`length`** – length of the active message data (even if the data argument is an opaque handle and not the actual data).
- **`param`** – pointer to additional parameters of the incoming message. The relevant fields are:
  - **`recv_attr`** – flags providing more information about the incoming message.

- **reply_ep** – if **UCP_AM_RECV_ATTR_FIELD_REPLY_EP** is set in **recv_attr**, then this field holds a handle to an endpoint that can be used to send replies to the active message sender.

The callback is expected to return **UCS_OK** if the message data has been consumed or if **UCP_AM_RECV_ATTR_FLAG_RNDV** is set in **recv_attr**. Otherwise, the if **UCP_AM_RECV_ATTR_FLAG_DATA** is set in **recv_attr**, the callback is allowed to keep the data for later processing (by adding it to an internal application queue, for example). In this case, the callback should return **UCS_INPROGRESS** as indication that the data should persist.

When a message arrives with **UCP_AM_RECV_ATTR_FLAG_RNDV** flag, the function ucp_am_recv_data_nbx must be used to fetch the data from the sender.

5.6.1.4.15 ucp_am_recv_data_nbx

```
ucs_status_ptr_t ucp_am_recv_data_nbx(ucp_worker_h worker, void *data_desc,
                                      void *buffer, size_t count,
                                  const ucp_request_param_t *param)
```

- **worker [in]** – UCP worker object to use for initiating the receive operation.

  > ⚠ The connection handle (endpoint) is not needed.

- **data_desc [in]** – handle for the data to receive. Obtained from the **data** argument for the active message callback.
- **buffer [in]** – receive buffer for the incoming data.
- **count [in]** – number of elements in the payload buffer. By default, each element is a single byte, so this is the byte-length of the buffer. Other data layouts, such as the IOV list, may be specified by **param->datatype**.
- **param [in]** – additional parameters that control request allocation and completion reporting. No specific flags are needed for this function.

The function returns a status pointer to check the operation's status. **NULL** means success.

This function is used for rendezvous active messages. The function initiates the process of fetching data from the sender side into an application-defined receive buffer. It is expected to be used when an active message callback is called with the **UCP_AM_RECV_ATTR_FLAG_RNDV** flag set in **params->recv_attr** field.

5.6.1.5 UCX Best Practices

5.6.1.5.1 Initialization

An application using UCX will usually create one global context (**ucp_context_h**) then create one or more workers (**ucp_worker_h**). Each worker consumes some memory for send/receive buffers, so it is not recommended to create too many workers. The rule of thumb is that the number of workers should be roughly tied to the number of CPU cores/threads.

The mapping of workers to threads is defined by the application's use case, for example:

- A single-threaded application does not need more than one worker
- A simple implementation of a multi-threaded application can create one or more workers in multi-threaded mode. These workers can be used by any thread.
- A multi-threaded application with a strong affinity between the thread and CPU core can create a dedicated worker per thread. These workers can be created in a single-threaded mode.
- Applications with many threads can implement a pool of workers and use one randomly or assign some to threads temporarily.

If there are multiple workers, each of them needs to create its own set of endpoints, since every endpoint connects a specific pair of workers.

To initiate communications, the application should create endpoints (**ucp_ep_h**) connected to the remote peers. There are two main methods to create an endpoint: Either by connecting directly to a remote worker's address, or by creating a listener object (**ucp_listener_h**) and connecting to remote IP address and port. These methods are described in more detail in the ucp_ep_create() section.

5.6.1.5.2 Communications

After initializing the UCP context, worker, and endpoints, the application can start using the endpoint for communications. Usually, endpoints are associated with application-level object that represents a connection.

Most communication operations follow a similar pattern: A non-blocking function (with **_nbx** suffix) receives a pointer to the **ucp_request_param_t** structure and returns **ucs_status_ptr_t** . Using a struct pointer allows extending the operations and while maintaining backward compatibility.

There are several types of communication methods supported by UCP intended for different kinds of applications. The recommended method for most applications is active messages which mean that the initiator can send arbitrary data to the responder, and the responder invokes a callback that can access this data.

5.6.2 MLX Drivers and Technologies

It is recommended to enable the "above 4G decoding" BIOS setting for features that require a large amount of PCIe resources (e.g., SR-IOV with numerous VFs, PCIe Emulated Switch, Large BAR Requests).

The chapter contains the following sections:

- InfiniBand Network
- Ethernet Network
- RDMA
- DPU
- Storage Protocols

- Virtualization
- Resiliency
- Docker Containers
- HPC-X
- Fast Driver Unload

5.6.2.1 InfiniBand Network

The chapter contains the following sections:

- InfiniBand Interface
- NVIDIA SM
- InfiniBand QoS
- IP Over InfiniBand
- Advanced Transport
- Optimized Memory Access
- NVIDIA PeerDirect
- CPU Overhead Distribution
- Out-of-order Data Placement
- IB Router
- MAD Congestion Control

5.6.2.1.1 InfiniBand Interface

5.6.2.1.1.1 Port Type Management

For information on port type management of ConnectX-4 and above adapter cards, please refer to Port Type Management Configuration section.

5.6.2.1.1.2 RDMA Counters

- RDMA counters are available only through sysfs located under:
  - **# /sys/class/infiniband/<device>/ports/*/hw_counters/**
  - **# /sys/class/infiniband/<device>/ports/*/counters**

For mlx5 port and RDMA counters, refer to the Understanding mlx5 Linux Counters Community post.

5.6.2.1.2 NVIDIA SM

5.6.2.1.2.1 OpenSM Application

OpenSM is an InfiniBand compliant Subnet Manager and Subnet Administrator that runs on top of the NVIDIA OFED stack. OpenSM performs the InfiniBand specification's required tasks for initializing InfiniBand hardware. One SM must be running for each InfiniBand subnet.

OpenSM defaults were designed to meet the common case usage on clusters with up to a few hundred nodes. Thus, in this default mode, OpenSM will scan the IB fabric, initialize it, and sweep occasionally for changes.

OpenSM attaches to a specific IB port on the local machine and configures only the fabric connected to it. (If the local machine has other IB ports, OpenSM will ignore the fabrics connected to those other ports). If no port is specified, opensm will select the first "best" available port. opensm can also present the available ports and prompt for a port number to attach to.

By default, the OpenSM run is logged to **/var/log/opensm.log**. All errors reported in this log file should be treated as indicators of IB fabric health issues.

When a fatal and non-recoverable error occurs, OpenSM will exit.

opensm.log should include the message "SUBNET UP" if OpenSM was able to set up the subnet correctly.

Syntax:

```
opensm [OPTIONS]
```

For the complete list of OpenSM options:

```
opensm --help / -h / -?
```

Environment Variables

The following environment variables control OpenSM behavior:

- **OSM_TMP_DIR** – controls the directory in which the temporary files generated by OpenSM are created. These files are: **opensm-subnet.lst**, **opensm.fdbs**, and **opensm.mcfdbs**. By default, this directory is **/var/log**.
- **OSM_CACHE_DIR** – opensm stores certain data to the disk such that subsequent runs are consistent. The default directory used is **/var/cache/opensm**. The following file is included in it:
    - **guid2lid** – stores the LID range assigned to each GUID

Signaling

When OpenSM receives a HUP signal, it starts a new heavy sweep as if a trap has been received or a topology change has been found.

Also, SIGUSR1 can be used to trigger a reopen of **/var/log/opensm.log** for logrotate purposes.

Running OpenSM as Daemon

OpenSM can also run as daemon. To run OpenSM in this mode, enter:

```
host1# service opensmd start
```

5.6.2.1.2.2 osmtest

osmtest is a test program for validating the InfiniBand Subnet Manager and Subnet Administrator. **osmtest** provides a test suite for opensm. It can create an inventory file of all available nodes, ports, and **PathRecords**, including all their fields. It can also verify the existing inventory with all the object fields and matches it to a pre-saved one.

osmtest has the following test flows:

- Multicast Compliancy test
- Event Forwarding test
- Service Record registration test
- RMPP stress test
- Small SA Queries stress test

For further information, please refer to the tool's man page.

5.6.2.1.2.3 Partitions

OpenSM enables the configuration of partitions (PKeys) in an InfiniBand fabric. By default, OpenSM searches for the partition's configuration file under the name **/etc/opensm/partitions.conf**. To change this filename, you can use opensm with the **--Pconfig** or **-P** flags.

The default partition is created by OpenSM unconditionally, even when a partition configuration file does not exist or cannot be accessed.

The default partition has a P_Key value of **0x7fff**. The port out of which runs OpenSM is assigned full membership in the default partition. All other end-ports are assigned partial membership.

- Adding a new partition to the **partition.conf** file, does not require SM restart, but signalling SM process via a HUP signal (e.g., **pkill -HUP opensm**).
- The default partition cannot be removed.

Adjustments to the Port GUIDs, including additions, removals, or membership alterations (denoted as **<PortGUID>=[full|limited|both]** in the **Partition Definition**) can be applied with a HUP signal to the Subnet Manager process (e.g., **pkill -HUP opensm**).

Performing changes in the **ipoib_bc_flags** (ipoib/sl/scope/rate/mtu) and **mgroup** flags of an existing partition requires a restart of the Subnet Manager to take effect.

File Format

Line content followed after **#** character is comment and ignored by parser.

- General file format:

```
<Partition Definition>:\[<newline>\]<Partition Properties>
```

- **<Partition Definition>** :

```
[PartitionName][=PKey][,indx0][,ipoib_bc_flags][,defmember=full|limited]
```

Where:

PartitionName	String, will be used with logging. When omitted empty string will be used.
PKey	P_Key value for this partition. Only low 15 bits will be used. When omitted will be auto-generated.
indx0	Indicates that this pkey should be inserted in block 0 index 0.
ipoib_bc_flags	Used to indicate/specify IPoIB capability of this partition.
defmember=full|limited|both	Specifies default membership for port GUID list. Default is limited.

ipoib_bc_flags are:

ipoib	Indicates that this partition may be used for IPoIB, as a result the IPoIB broadcast group will be created with the flags given, if any.
rate=<val>	Specifies rate for this IPoIB MC group (default is 3 (10GB/s))
mtu=<val>	Specifies MTU for this IPoIB MC group (default is 4 (2048))
sl=<val>	Specifies SL for this IPoIB MC group (default is 0)
scope=<val>	Specifies scope for this IPoIB MC group (default is 2 (link local))

- **<Partition Properties>** :

```
\[<Port list>|<MCast Group>\]* | <Port list>
```

  - **<Port List>** :

```
<Port Specifier>[,<Port Specifier>]
```

    - **<Port Specifier>** :

```
<PortGUID>[=[full|limited|both]]
```

Where:

PortGUID	GUID of partition member EndPort. Hexadecimal numbers should start from 0x, decimal numbers are accepted too.
full, limited	Indicates full and/or limited membership for this both port. When omitted (or unrecognized) limited membership is assumed. Both indicate full and limited membership for this port.

- **<MCast Group>** :

```
mgid=gid[,mgroup_flag]*<newline>
```

Where:

mgid=gid	gid specified is verified to be a Multicast address IP groups are verified to match the rate and mtu of the broadcast group. The P_Key bits of the mgid for IP groups are verified to either match the P_Key specified in by "Partition Definition" or if they are 0x0000 the P_Key will be copied into those bits.	
mgroup_flag	**rate=<val>**	Specifies rate for this MC group (default is 3 (10GBps))
	mtu=<val>	Specifies MTU for this MC group (default is 4 (2048))
	sl=<val>	Specifies SL for this MC group (default is 0)
	scope=<val>	Specifies scope for this MC group (default is 2 (link local)). Multiple scope settings are permitted for a partition. ⚠ This overwrites the scope nibble of the specified mgid. Furthermore specifying multiple scope settings will result in multiple MC groups being created.
	qkey=<val>	Specifies the Q_Key for this MC group (default: 0x0b1b for IP groups, 0 for other groups)
	tclass=<val>	Specifies tclass for this MC group (default is 0)
	FlowLabel=<val>	Specifies FlowLabel for this MC group (default is 0)

Note that values for rate, MTU, and scope should be specified as defined in the IBTA specification (e.g., **mtu=4** for 2048). To use 4K MTU, edit that entry to **mtu=5** (5 indicates 4K MTU to that specific partition).

PortGUIDs list:

```
PortGUID    GUID of partition member EndPort. Hexadecimal numbers should start from 0x, decimal numbers are
accepted too.
full or limited indicates full or limited membership for this port. When omitted (or unrecognized) limited
membership is assumed.
```

There are some useful keywords for **PortGUID** definition:

- **ALL_CAS** means all Channel Adapter end ports in this subnet
- **ALL_VCAS** means all virtual end ports in the subnet
- **ALL_SWITCHES** means all Switch end ports in this subnet
- **ALL_ROUTERS** means all Router end ports in this subnet
- **SELF** means subnet manager's port. An empty list means that there are no ports in this partition

Notes:

- White space is permitted between delimiters (**=**, **,**, **:**, **;**).
- **PartitionName** does not need to be unique, PKey does need to be unique. If PKey is repeated then those partition configurations will be merged and the first **PartitionName** will be used (see the next note).
- It is possible to split partition configuration in more than one definition, but then PKey should be explicitly specified (otherwise different PKey values will be generated for those definitions).

Examples:

```
Default=0x7fff : ALL, SELF=full ;
Default=0x7fff : ALL, ALL_SWITCHES=full, SELF=full ;

NewPartition , ipoib : 0x123456=full, 0x3456789034=limi, 0x2134af2306 ;

YetAnotherOne = 0x300 : SELF=full ;
YetAnotherOne = 0x300 : ALL=limited ;

ShareIO = 0x80 , defmember=full : 0x123451, 0x123452;
# 0x123453, 0x123454 will be limited
ShareIO = 0x80 : 0x123453, 0x123454, 0x123455=full;
# 0x123456, 0x123457 will be limited
ShareIO = 0x80 : defmember=limited : 0x123456, 0x123457, 0x123458=full;
ShareIO = 0x80 , defmember=full : 0x123459, 0x12345a;
ShareIO = 0x80 , defmember=full : 0x12345b, 0x12345c=limited, 0x12345d;

# multicast groups added to default
Default=0x7fff,ipoib:
mgid=ff12:401b::0707,sl=1 # random IPv4 group
mgid=ff12:601b::16 # MLDv2-capable routers
mgid=ff12:401b::16 # IGMP
mgid=ff12:601b::2 # All routers
mgid=ff12::1,sl=1,Q_Key=0xDEADBEEF,rate=3,mtu=2 # random group
ALL=full;
```

The following rule is equivalent to how OpenSM used to run prior to the partition manager:

```
Default=0x7fff,ipoib:ALL=full;
```

5.6.2.1.2.4 Effect of Topology Changes

If a link is added or removed, OpenSM may not recalculate the routes that do not have to change. A route has to change if the port is no longer UP or no longer the MinHop. When routing changes are performed, the same algorithm for balancing the routes is invoked.

In the case of using the file-based routing, any topology changes are currently ignored. The **file** routing engine just loads the LFTs from the file specified, with no reaction to real topology. Obviously, this will not be able to recheck LIDs (by GUID) for disconnected nodes, and LFTs for non-existent switches will be skipped. Multicast is not affected by **file** routing engine (this uses min hop tables).

5.6.2.1.2.5 Routing Algorithms

OpenSM offers the following routing engines:

1. Min Hop Algorithm
   Based on the minimum hops to each node where the path length is optimized.
2. UPDN Algorithm
   Based on the minimum hops to each node, but it is constrained to ranking rules. This algorithm should be chosen if the subnet is not a pure Fat Tree, and a deadlock may occur due to a loop in the subnet.
3. Fat-tree Routing Algorithm
   This algorithm optimizes routing for a congestion-free "shift" communication pattern. It should be chosen if a subnet is a symmetrical Fat Tree of various types, not just a K-ary-N-Tree: non-constant K, not fully staffed, and for any CBB ratio. Similar to UPDN, Fat Tree routing is constrained to ranking rules.
4. DOR Routing Algorithm
   Based on the Min Hop algorithm, but avoids port equalization except for redundant links between the same two switches. This provides deadlock free routes for hypercubes when the fabric is cabled as a hypercube and for meshes when cabled as a mesh.
5. Torus-2QoS Routing Algorithm
   Based on the DOR Unicast routing algorithm specialized for 2D/3D torus topologies. Torus-2QoS provides deadlock-free routing while supporting two quality of service (QoS) levels. Additionally, it can route around multiple failed fabric links or a single failed fabric switch without introducing deadlocks, and without changing path SL values granted before the failure.
6. Routing Chains
   Allows routing configuration of different parts of a single InfiniBand subnet by different routing engines. In the current release, minhop/updn/ftree/dor/torus-2QoS/pqft can be combined.

Please note that LASH Routing Algorithm is not supported.

MINHOP/UPDN/DOR routing algorithms are comprised of two stages:

1. MinHop matrix calculation. How many hops are required to get from each port to each LID. The algorithm to fill these tables is different if you run standard (min hop) or Up/Down. For standard routing, a "relaxation" algorithm is used to propagate min hop from every destination LID through neighbor switches. For Up/Down routing, a BFS from every target is used. The BFS tracks link direction (up or down) and avoid steps that will perform up after a down step was used.
2. Once MinHop matrices exist, each switch is visited and for each target LID a decision is made as to what port should be used to get to that LID. This step is common to standard and Up/Down routing. Each port has a counter counting the number of target LIDs going through it. When there are multiple alternative ports with same MinHop to a LID, the one with less previously assigned ports is selected.
   If LMC > 0, more checks are added. Within each group of LIDs assigned to same target port:
   a. Use only ports which have same MinHop

b. First prefer the ones that go to different systemImageGuid (then the previous LID of the same LMC group)
c. If none, prefer those which go through another NodeGuid
d. Fall back to the number of paths method (if all go to same node).

Min Hop Algorithm

The Min Hop algorithm is invoked by default if no routing algorithm is specified. It can also be invoked by specifying '-R minhop'.
The Min Hop algorithm is divided into two stages: computation of min-hop tables on every switch and LFT output port assignment. Link subscription is also equalized with the ability to override based on port GUID. The latter is supplied by:

```
-i <equalize-ignore-guids-file>
-ignore-guids <equalize-ignore-guids-file>
```

This option provides the means to define a set of ports (by GUIDs) that will be ignored by the link load equalization algorithm.

LMC awareness routes based on a (remote) system or on a switch basis.

UPDN Algorithm

The UPDN algorithm is designed to prevent deadlocks from occurring in loops of the subnet. A loop-deadlock is a situation in which it is no longer possible to send data between any two hosts connected through the loop. As such, the UPDN routing algorithm should be sent if the subnet is not a pure Fat Tree, and one of its loops may experience a deadlock (due, for example, to high pressure).
The UPDN algorithm is based on the following main stages:

1. Auto-detect root nodes – based on the CA hop length from any switch in the subnet, a statistical histogram is built for each switch (hop num vs the number of occurrences). If the histogram reflects a specific column (higher than others) for a certain node, then it is marked as a root node. Since the algorithm is statistical, it may not find any root nodes. The list of the root nodes found by this auto-detect stage is used by the ranking process stage.

   The user can override the node list manually.

   If this stage cannot find any root nodes, and the user did not specify a GUID list file, OpenSM defaults back to the Min Hop routing algorithm.

2. Ranking process – All root switch nodes (found in stage 1) are assigned a rank of 0. Using the BFS algorithm, the rest of the switch nodes in the subnet are ranked incrementally. This ranking aids in the process of enforcing rules that ensure loop-free paths.
3. Min Hop Table setting – after ranking is done, a BFS algorithm is run from each (CA or switch) node in the subnet. During the BFS process, the FDB table of each switch node traversed by BFS is updated, in reference to the starting node, based on the ranking rules and GUID values.

At the end of the process, the updated FDB tables ensure loop-free paths through the subnet.

UPDN Algorithm Usage

Activation through OpenSM:

- Use `-R updn` option (instead of old `-u`) to activate the UPDN algorithm.
- Use `-a <root_guid_file>` for adding an UPDN GUID file that contains the root nodes for ranking. If the `-a` option is not used, OpenSM uses its auto-detect root nodes algorithm.

Notes on the GUID list file:

- A valid GUID file specifies one GUID in each line. Lines with an invalid format will be discarded
- The user should specify the root switch GUIDs

Fat-tree Routing Algorithm

The fat-tree algorithm optimizes routing for "shift" communication pattern. It should be chosen if a subnet is a symmetrical or almost symmetrical fat-tree of various types. It supports not just K-ary-N-Trees, by handling for non-constant K, cases where not all leafs (CAs) are present, any Constant Bisectional Ratio (CBB) ratio. As in UPDN, fat-tree also prevents credit-loop-dead- locks.

If the root GUID file is not provided (`a` or `-root_guid_file` options), the topology has to be pure fat-tree that complies with the following rules:

- Tree rank should be between two and eight (inclusively)
- Switches of the same rank should have the same number of UP-going port groups, unless they are root switches, in which case the shouldn't have UP-going ports at all.
  Note: Ports that are connected to the same remote switch are referenced as “port group”.
- Switches of the same rank should have the same number of DOWN-going port groups, unless they are leaf switches.
- Switches of the same rank should have the same number of ports in each UP-going port group.
- Switches of the same rank should have the same number of ports in each DOWN-going port group.
- All the CAs have to be at the same tree level (rank).

If the root GUID file is provided, the topology does not have to be pure fat-tree, and it should only comply with the following rules:

- Tree rank should be between two and eight (inclusively)
- All the Compute Nodes have to be at the same tree level (rank). Note that non-compute node CAs are allowed here to be at different tree ranks.
  Note: List of compute nodes (CNs) can be specified using `-u` or `--cn_guid_file` OpenSM options.

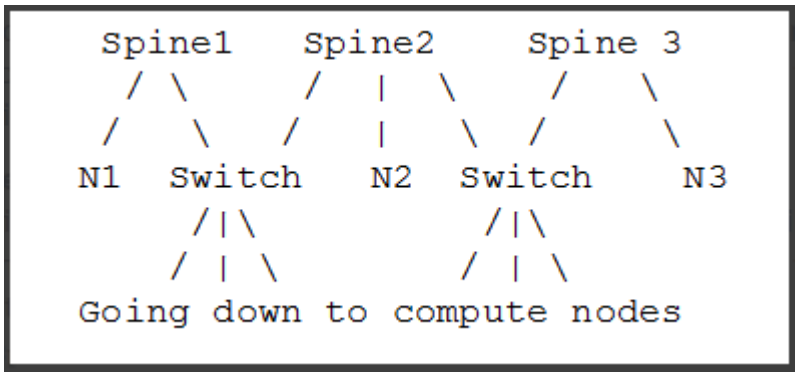
Topologies that do not comply cause a fallback to min-hop routing. Note that this can also occur on link failures which cause the topology to no longer be a "pure" fat-tree.

Note that although fat-tree algorithm supports trees with non-integer CBB ratio, the routing will not be as balanced as in case of integer CBB ratio. In addition to this, although the algorithm allows leaf switches to have any number of CAs, the closer the tree is to be fully populated, the more effective the "shift" communication pattern will be. In general, even if the root list is provided, the closer the topology to a pure and symmetrical fat-tree, the more optimal the routing will be.

The algorithm also dumps the compute node ordering file (**opensm-ftree-ca-order.dump**) in the same directory where the OpenSM log resides. This ordering file provides the CN order that may be used to create efficient communication pattern, that will match the routing tables.

Routing between non-CN Nodes

The use of the **io_guid_file** option allows non-CN nodes to be located on different levels in the fat tree. In such case, it is not guaranteed that the Fat Tree algorithm will route between two non-CN nodes. In the scheme below, N1, N2, and N3 are non-CN nodes. Although all the CN have routes to and from them, there will not necessarily be a route between N1, N2, and N3. Such routes would require using at least one of the switches the wrong way around.



To solve this problem, a list of non-CN nodes can be specified by \'-G\' or \'--io_guid_file\' option. These nodes will be allowed to use switches the wrong way around a specific number of times (specified by \'-H\' or \'--max_reverse_hops\'. With the proper max_reverse_hops and io_guid_file values, you can ensure full connectivity in the Fat Tree. In the scheme above, with a **max_reverse_hop** of 1, routes will be instantiated between N1<->N2 and N2<->N3. With a max_reverse_hops value of 2, N1,N2 and N3 will all have routes between them.

> ⚠ Using max_reverse_hops creates routes that use the switch in a counter-stream way. This option should never be used to connect nodes with high bandwidth traffic between them! It should only be used to allow connectivity for HA purposes or similar. Also having routes the other way around can cause credit loops.

Activation through OpenSM

Use **-R ftree** option to activate the fat-tree algorithm.

> ⚠ LMC > 0 is not supported by fat-tree routing. If this is specified, the default routing algorithm is invoked instead.

DOR Routing Algorithm

The Dimension Order Routing algorithm is based on the Min Hop algorithm and so uses shortest paths. Instead of spreading traffic out across different paths with the same shortest distance, it chooses among the available shortest paths based on an ordering of dimensions. Each port must be consistently cabled to represent a hypercube dimension or a mesh dimension. Paths are grown

from a destination back to a source using the lowest dimension (port) of available paths at each step. This provides the ordering necessary to avoid deadlock. When there are multiple links between any two switches, they still represent only one dimension and traffic is balanced across them unless port equalization is turned off. In the case of hypercubes, the same port must be used throughout the fabric to represent the hypercube dimension and match on both ends of the cable. In the case of meshes, the dimension should consistently use the same pair of ports, one port on one end of the cable, and the other port on the other end, continuing along the mesh dimension.

Use `-R dor` option to activate the DOR algorithm.

Torus-2QoS Routing Algorithm

Torus-2QoS is a routing algorithm designed for large-scale 2D/3D torus fabrics. The torus-2QoS routing engine can provide the following functionality on a 2D/3D torus:

- Free of credit loops routing
- Two levels of QoS, assuming switches support 8 data VLs
- Ability to route around a single failed switch, and/or multiple failed links, without:
    - introducing credit loops
    - changing path SL values
- Very short run times, with good scaling properties as fabric size increases

Unicast Routing

Torus-2 QoS is a DOR-based algorithm that avoids deadlocks that would otherwise occur in a torus using the concept of a dateline for each torus dimension. It encodes into a path SL which datelines the path crosses as follows:

```
sl = 0;
for (d = 0; d < torus_dimensions; d++)
/* path_crosses_dateline(d) returns 0 or 1 */
sl |= path_crosses_dateline(d) << d;
```

For a 3D torus, that leaves one SL bit free, which torus-2 QoS uses to implement two QoS levels. Torus-2 QoS also makes use of the output port dependence of switch SL2VL maps to encode into one VL bit the information encoded in three SL bits. It computes in which torus coordinate direction each inter-switch link "points", and writes SL2VL maps for such ports as follows:

```
for (sl = 0; sl < 16; sl ++)
/* cdir(port) reports which torus coordinate direction a switch port
* "points" in, and returns 0, 1, or 2 */
sl2vl(iport,oport,sl) = 0x1 & (sl >> cdir(oport));
```

Thus, on a pristine 3D torus, i.e., in the absence of failed fabric switches, torus-2 QoS consumes 8 SL values (SL bits 0-2) and 2 VL values (VL bit 0) per QoS level to provide deadlock-free routing on a 3D torus. Torus-2 QoS routes around link failure by "taking the long way around" any 1D ring interrupted by a link failure. For example, consider the 2D 6x5 torus below, where switches are denoted by [+a-zA-Z]:

For a pristine fabric the path from S to D would be S-n-T-r-D. In the event that either link S-n or n-T has failed, torus-2QoS would use the path S-m-p-o-T-r-D.

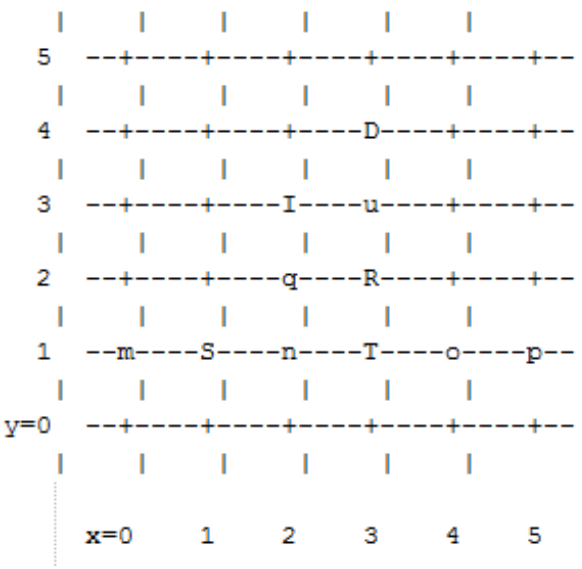
Note that it can do this without changing the path SL value; once the 1D ring m-S-n-T-o-p-m has been broken by failure, path segments using it cannot contribute to deadlock, and the x-direction dateline (between, say, x=5 and x=0) can be ignored for path segments on that ring. One result of this is that torus-2QoS can route around many simultaneous link failures, as long as no 1D ring is broken into disjoint segments. For example, if links n-T and T-o have both failed, that ring has been broken into two disjoint segments, T and o-p-m-S-n. Torus-2QoS checks for such issues, reports if they are found, and refuses to route such fabrics.

Note that in the case where there are multiple parallel links between a pair of switches, torus-2QoS will allocate routes across such links in a round-robin fashion, based on ports at the path destination switch that are active and not used for inter-switch links. Should a link that is one of several such parallel links fail, routes are redistributed across the remaining links. When the last of such a set of parallel links fails, traffic is rerouted as described above.

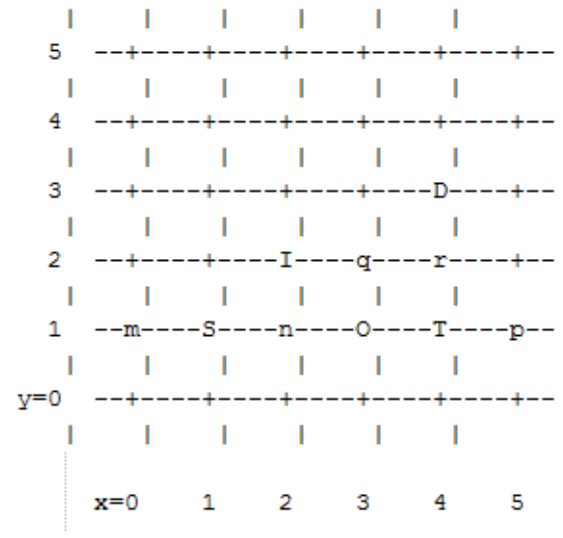
Handling a failed switch under DOR requires introducing into a path at least one turn that would be otherwise "illegal", i.e. not allowed by DOR rules. Torus-2QoS will introduce such a turn as close as possible to the failed switch in order to route around it. n the above example, suppose switch T has failed, and consider the path from S to D. Torus-2QoS will produce the path S-n-I-r-D, rather than the S-n-T-r-D path for a pristine torus, by introducing an early turn at n. Normal DOR rules will cause traffic arriving at switch I to be forwarded to switch r; for traffic arriving from I due to the "early" turn at n, this will generate an "illegal" turn at I.

Torus-2QoS will also use the input port dependence of SL2VL maps to set VL bit 1 (which would be otherwise unused) for y-x, z-x, and z-y turns, i.e., those turns that are illegal under DOR. This causes the first hop after any such turn to use a separate set of VL values, and prevents deadlock in the presence of a single failed switch. For any given path, only the hops after a turn that is illegal under DOR can contribute to a credit loop that leads to deadlock. So in the example above with failed switch T, the location of the illegal turn at I in the path from S to D requires that any credit loop caused by that turn must encircle the failed switch at T. Thus the second and later hops after the illegal turn at I (i.e., hop r-D) cannot contribute to a credit loop because they cannot be used to construct a loop encircling T. The hop I-r uses a separate VL, so it cannot contribute to a credit loop encircling T. Extending this argument shows that in addition to being capable of routing around a single switch failure without introducing deadlock, torus-2QoS can also route

around multiple failed switches on the condition they are adjacent in the last dimension routed by DOR. For example, consider the following case on a 6x6 2D torus:



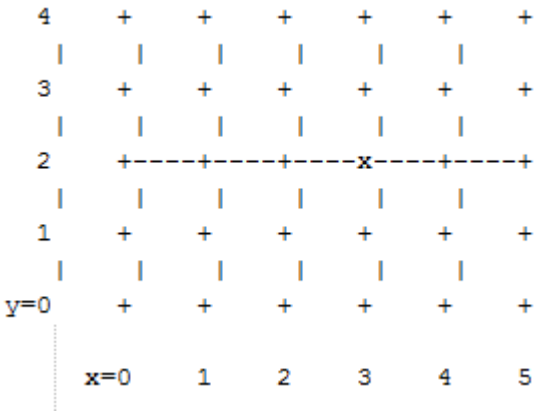
Suppose switches T and R have failed, and consider the path from S to D. Torus-2QoS will generate the path S-n-q-I-u-D, with an illegal turn at switch I, and with hop I-u using a VL with bit 1 set. As a further example, consider a case that torus-2QoS cannot route without deadlock: two failed switches adjacent in a dimension that is not the last dimension routed by DOR; here the failed switches are O and T:



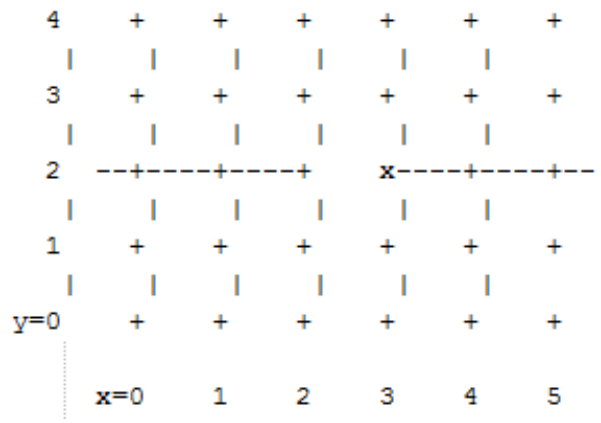
In a pristine fabric, torus-2QoS would generate the path from S to D as S-n-O-T-r-D. With failed switches O and T, torus-2QoS will generate the path S-n-I-q-r-D, with an illegal turn at switch I, and with hop I-q using a VL with bit 1 set. In contrast to the earlier examples, the second hop after the illegal turn, q-r, can be used to construct a credit loop encircling the failed switches.

Multicast Routing

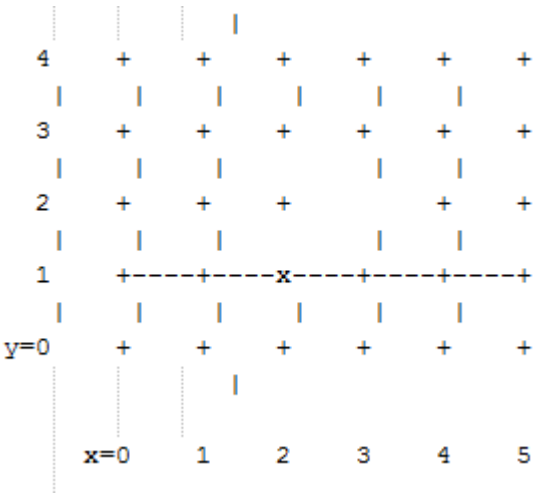
Since torus-2QoS uses all four available SL bits, and the three data VL bits that are typically available in current switches, there is no way to use SL/VL values to separate multicast traffic from unicast traffic. Thus, torus-2QoS must generate multicast routing such that credit loops cannot arise from a combination of multicast and unicast path segments. It turns out that it is possible to construct spanning trees for multicast routing that have that property. For the 2D 6x5 torus example above, here is the full-fabric spanning tree that torus-2QoS will construct, where "x" is the root switch and each "+" is a non-root switch:



For multicast traffic routed from root to tip, every turn in the above spanning tree is a legal DOR turn. For traffic routed from tip to root, and some traffic routed through the root, turns are not legal DOR turns. However, to construct a credit loop, the union of multicast routing on this spanning tree with DOR unicast routing can only provide 3 of the 4 turns needed for the loop. In addition, if none of the above spanning tree branches crosses a dateline used for unicast credit loop avoidance on a torus, and if multicast traffic is confined to SL 0 or SL 8 (recall that torus-2QoS uses SL bit 3 to differentiate QoS level), then multicast traffic also cannot contribute to the "ring" credit loops that are otherwise possible in a torus. Torus-2QoS uses these ideas to create a master spanning tree. Every multicast group spanning tree will be constructed as a subset of the master tree, with the same root as the master tree. Such multicast group spanning trees will in general not be optimal for groups which are a subset of the full fabric. However, this compromise must be made to enable support for two QoS levels on a torus while preventing credit loops. In the presence of link or switch failures that result in a fabric for which torus-2QoS can generate credit-loop-free unicast routes, it is also possible to generate a master spanning tree for multicast that retains the required properties. For example, consider that same 2D 6x5 torus, with the link from (2,2) to (3,2) failed. Torus-2QoS will generate the following master spanning tree:



Two things are notable about this master spanning tree. First, assuming the x dateline was between x=5 and x=0, this spanning tree has a branch that crosses the dateline. However, just as for unicast, crossing a dateline on a 1D ring (here, the ring for y=2) that is broken by a failure cannot contribute to a torus credit loop. Second, this spanning tree is no longer optimal even for multicast groups that encompass the entire fabric. That, unfortunately, is a compromise that must be made to retain the other desirable properties of torus-2QoS routing. In the event that a single switch fails, torus-2QoS will generate a master spanning tree that has no "extra" turns by appropriately selecting a root switch. In the 2D 6x5 torus example, assume now that the switch at (3,2) (i.e., the root for a pristine fabric), fails. Torus-2QoS will generate the following master spanning tree for that case:



Assuming the dateline was between y=4 and y=0, this spanning tree has a branch that crosses a dateline. However, this cannot contribute to credit loops as it occurs on a 1D ring (the ring for x=3) that is broken by failure, as in the above example.

Torus Topology Discovery

The algorithm used by torus-2QoS to construct the torus topology from the undirected graph representing the fabric requires that the radix of each dimension be configured via torus-2QoS.conf. It also requires that the torus topology be "seeded"; for a 3D torus this requires

configuring four switches that define the three coordinate directions of the torus. Given this starting information, the algorithm is to examine the cube formed by the eight switch locations bounded by the corners (x,y,z) and (x+1,y+1,z+1). Based on switches already placed into the torus topology at some of these locations, the algorithm examines 4-loops of inter-switch links to find the one that is consistent with a face of the cube of switch locations and adds its switches to the discovered topology in the correct locations.

Because the algorithm is based on examining the topology of 4-loops of links, a torus with one or more radix-4 dimensions requires extra initial seed configuration. See torus-2QoS.conf(5) for details. Torus-2QoS will detect and report when it has an insufficient configuration for a torus with radix-4 dimensions.

In the event the torus is significantly degraded, i.e., there are many missing switches or links, it may happen that torus-2QoS is unable to place into the torus some switches and/or links that were discovered in the fabric, and will generate a warning in that case. A similar condition occurs if torus-2QoS is misconfigured, i.e., the radix of a torus dimension as configured does not match the radix of that torus dimension as wired, and many switches/links in the fabric will not be placed into the torus.

Quality Of Service Configuration

OpenSM will not program switches and channel adapters with SL2VL maps or VL arbitration configuration unless it is invoked with **-Q**. Since torus-2QoS depends on such functionality for correct operation, always invoke OpenSM with **-Q** when torus-2QoS is in the list of routing engines. Any quality of service configuration method supported by OpenSM will work with torus-2QoS, subject to the following limitations and considerations. For all routing engines supported by OpenSM except torus-2QoS, there is a one-to-one correspondence between QoS level and SL. Torus-2QoS can only support two quality of service levels, so only the high-order bit of any SL value used for unicast QoS configuration will be honored by torus-2QoS. For multicast QoS configuration, only SL values 0 and 8 should be used with torus-2QoS.

Since SL to VL map configuration must be under the complete control of torus-2QoS, any configuration via **qos_sl2vl**, **qos_swe_sl2vl**, etc., must and will be ignored, and a warning will be generated. Torus-2QoS uses VL values 0-3 to implement one of its supported QoS levels, and VL values 4-7 to implement the other. Hard-to-diagnose application issues may arise if traffic is not delivered fairly across each of these two VL ranges. Torus-2QoS will detect and warn if VL arbitration is configured unfairly across VLs in the range 0-3, and also in the range 4-7. Note that the default OpenSM VL arbitration configuration does not meet this constraint, so all torus-2QoS users should configure VL arbitration via **qos_vlarb_high**, **qos_vlarb_low**, etc.

Operational Considerations

Any routing algorithm for a torus IB fabric must employ path SL values to avoid credit loops. As a result, all applications run over such fabrics must perform a path record query to obtain the correct path SL for connection setup. Applications that use rdma_cm for connection setup will automatically meet this requirement.

If a change in fabric topology causes changes in path SL values required to route without credit loops, in general, all applications would need to repath to avoid message deadlock. Since torus-2QoS has the ability to reroute after a single switch failure without changing path SL values, re-pathing by running applications is not required when the fabric is routed with torus-2QoS.

Torus-2QoS can provide unchanging path SL values in the presence of subnet manager failover provided that all OpenSM instances have the same idea of dateline location. See torus-

2QoS.conf(5) for details. Torus-2QoS will detect configurations of failed switches and links that prevent routing that is free of credit loops and will log warnings and refuse to route. If **no_fallback** was configured in the list of OpenSM routing engines, then no other routing engine will attempt to route the fabric. In that case, all paths that do not transit the failed components will continue to work, and the subset of paths that are still operational will continue to remain free of credit loops. OpenSM will continue to attempt to route the fabric after every sweep interval and after any change (such as a link up) in the fabric topology. When the fabric components are repaired, full functionality will be restored. In the event OpenSM was configured to allow some other engine to route the fabric if torus-2QoS fails, then credit loops and message deadlock are likely if torus-2QoS had previously routed the fabric successfully. Even if the other engine is capable of routing a torus without credit loops, applications that built connections with path SL values granted under torus-2QoS will likely experience message deadlock under routing generated by a different engine, unless they repath. To verify that a torus fabric is routed free of credit loops, use **ibdmchk** to analyze data collected via **ibdiagnt -vlr**.

Torus-2QoS Configuration File Syntax

The file **torus-2QoS.conf** contains configuration information that is specific to the OpenSM routing engine torus-2QoS. Blank lines and lines where the first non-whitespace character is **#** are ignored. A token is any contiguous group of non-whitespace characters. Any tokens on a line following the recognized configuration tokens described below are ignored.

```
[torus|mesh] x_radix[m|M|t|T] y_radix[m|M|t|T] z_radix[m|M|t|T]
```

Either torus or mesh must be the first keyword in the configuration and sets the topology that torus-2QoS will try to construct. A 2D topology can be configured by specifying one of **x_radix**, **y_radix**, or **z_radix** as 1. An individual dimension can be configured as mesh (open) or torus (looped) by suffixing its radix specification with one of m, M, t, or T. Thus, **mesh 3T 4 5** and **torus 3 4M 5M** both specify the same topology.

Note that although torus-2QoS can route mesh fabrics, its ability to route around failed components is severely compromised on such fabrics. A failed fabric components very likely to cause a disjoint ring; see UNICAST ROUTING in torus-2QoS(8).

```
xp_link sw0_GUID sw1_GUID
yp_link sw0_GUID sw1_GUID
zp_link sw0_GUID sw1_GUID
xm_link sw0_GUID sw1_GUID
ym_link sw0_GUID sw1_GUID
zm_link sw0_GUID sw1_GUID
```

These keywords are used to seed the torus/mesh topology. For example, **xp_link 0x2000 0x2001** specifies that a link from the switch with node GUID 0x2000 to the switch with node GUID 0x2001 would point in the positive x direction, while **xm_link 0x2000 0x2001** specifies that a link from the switch with node GUID 0x2000 to the switch with node GUID 0x2001 would point in the negative x direction. All the link keywords for a given seed must specify the same "from" switch.

In general, it is not necessary to configure both the positive and negative directions for a given coordinate; either is sufficient. However, the algorithm used for topology discovery needs extra information for torus dimensions of radix four (see TOPOLOGY DISCOVERY in torus-2QoS(8)). For such cases, both the positive and negative coordinate directions must be specified.

Based on the topology specified via the torus/mesh keyword, torus-2QoS will detect and log when it has insufficient seed configuration.

```
GUIDx_dateline position
y_dateline position
z_dateline position
```

In order for torus-2QoS to provide the guarantee that path SL values do not change under any conditions for which it can still route the fabric, its idea of dateline position must not change relative to physical switch locations. The dateline keywords provide the means to configure such behavior.

The dateline for a torus dimension is always between the switch with coordinate 0 and the switch with coordinate radix-1 for that dimension. By default, the common switch in a torus seed is taken as the origin of the coordinate system used to describe switch location. The position parameter for a dateline keyword moves the origin (and hence the dateline) the specified amount relative to the common switch in a torus seed.

```
next_seed
```

If any of the switches used to specify a seed were to fail torus-2QoS would be unable to complete topology discovery successfully. The **next_seed** keyword specifies that the following link and dateline keywords apply to a new seed specification.

For maximum resiliency, no seed specification should share a switch with any other seed specification. Multiple seed specifications should use dateline configuration to ensure that torus-2QoS can grant path SL values that are constant, regardless of which seed was used to initiate topology discovery.

- **portgroup_max_ports max_ports** – This keyword specifies the maximum number of parallel inter-switch links, and also the maximum number of host ports per switch, that torus-2QoS can accommodate. The default value is 16. Torus-2QoS will log an error message during topology discovery if this parameter needs to be increased. If this keyword appears multiple times, the last instance prevails.
- **port_order p1 p2 p3 ...** – This keyword specifies the order in which CA ports on a destination switch are visited when computing routes. When the fabric contains switches connected with multiple parallel links, routes are distributed in a round-robin fashion across such links and so changing the order that CA ports are visited changes the distribution of routes across such links. This may be advantageous for some specific traffic patterns.

The default is to visit CA ports in increasing port order on destination switches. Duplicate values in the list will be ignored.

Example:

```
# Look for a 2D (since x radix is one) 4x5 torus.
torus 1 4 5
# y is radix-4 torus dimension, need both
# ym_link and yp_link configuration.
yp_link 0x200000 0x200005 # sw @ y=0,z=0 -> sw @ y=1,z=0
ym_link 0x200000 0x20000f # sw @ y=0,z=0 -> sw @ y=3,z=0
# z is not radix-4 torus dimension, only need one of
# zm_link or zp_link configuration.
zp_link 0x200000 0x200001 # sw @ y=0,z=0 -> sw @ y=0,z=1
next_seed
yp_link 0x20000b 0x200010 # sw @ y=2,z=1 -> sw @ y=3,z=1
ym_link 0x20000b 0x200006 # sw @ y=2,z=1 -> sw @ y=1,z=1
zp_link 0x20000b 0x20000c # sw @ y=2,z=1 -> sw @ y=2,z=2
y_dateline -2 # Move the dateline for this seed
z_dateline -1 # back to its original position.
```

```
# If OpenSM failover is configured, for maximum resiliency
# one instance should run on a host attached to a switch
# from the first seed, and another instance should run
# on a host attached to a switch from the second seed.
# Both instances should use this torus-2QoS.conf to ensure
# path SL values do not change in the event of SM failover.
# port_order defines the order on which the ports would be
# chosen for routing.
port_order 7 10 8 11 9 12 25 28 26 29 27 30
```

Routing Chains

The routing chains feature is offering a solution that enables one to configure different parts of the fabric and define a different routing engine to route each of them. The routings are done in a sequence (hence the name "chains") and any node in the fabric that is configured in more than one part is left with the routing updated by the last routing engine it was a part of.

Configuring Routing Chains

To configure routing chains:

1. Define the port groups.
2. Define topologies based on previously defined port groups.
3. Define configuration files for each routing engine.
4. Define routing engine chains over previously defined topologies and configuration files.

Defining Port Groups

The basic idea behind the port groups is the ability to divide the fabric into sub-groups and give each group an identifier that can be used to relate to all nodes in this group. The port groups is a separate feature from the routing chains but is a mandatory prerequisite for it. In addition, it is used to define the participants in each of the routing algorithms.

Defining a Port Group Policy File

To define a port group policy file, set the parameter **pgrp_policy_file** in the OpenSM configuration file (**pgrp_policy_file /etc/opensm/conf/port_groups_policy_file**).

Configuring a Port Group Policy

The port groups policy file details the port groups in the fabric. The policy file should be composed of one or more paragraphs that define a group. Each paragraph should begin with the line 'port-group' and end with the line **end-port-group** .

For example:

```
port-group
…port group qualifiers…
end-port-group
```

Port Group Qualifiers

Unlike the port group's beginning and end which do not require a colon, all qualifiers must end with a colon (':'). Also - a colon is a predefined mark that must not be used inside qualifier values. The inclusion of a colon in the name or the use of a port group will result in the policy's failure.

Rule Qualifier

Parameter	Description	Example
name	Each group must have a name. Without a name qualifier, the policy fails.	name: grp1
use	Optional qualifier that one can define in order to describe the usage of this port group (if undefined, an empty string is used as a default).	use: first port group

There are several qualifiers used to describe a rule that determines which ports will be added to the group. Each port group may include one or more rules out of the rules described in the below table (at least one rule must be defined for each port group).

Parameter	Description	Example
guid list	Comma separated list of GUIDs to include in the group. If no specific physical ports were configured, all physical ports of the guid are chosen. However, for each guid , one can detail specific physical ports to be included in the group. This can be done using the following syntax: • Specify a specific port in a guid to be chosen port-guid: 0x283@3 • Specify a specific list of ports in a guid to be chosen port-guid: 0x286@1/5/7 • Specify a specific range of ports in a guid to be chosen port-guid: 0x289@2-5 • Specify a list of specific ports and ports ranges in a guid to be chosen port-guid: 0x289@2-5/7/9-13/18 • Complex rule port-guid: 0x283@5-8/12/14, 0x286, 0x289/6/ 8/12	port-guid: 0x283, 0x286, 0x289
port guid range	It is possible to configure a range of guids to be chosen to the group. However, while using the range qualifier, it is impossible to detail specific physical ports. Note: A list of ranges cannot be specified. The below example is invalid and will cause the policy to fail: port-guid-range: 0x283-0x289, 0x290-0x295	port-guid-range: 0x283-0x289
port name	One can configure a list of hostnames as a rule. Hosts with a node description that is built out of these hostnames will be chosen. Since the node description contains the network card index as well, one might also specify a network card index and a physical port to be chosen. For example, the given configuration will cause only physical port 2 of a host with the node description 'kuku HCA-1' to be chosen. port and hca_idx parameters are optional. If the port is unspecified, all physical ports are chosen. If hca_idx is unspecified, all card numbers are chosen. Specifying a hostname is mandatory. One can configure a list of hostname/ port/hca_idx sets in the same qualifier as follows: port-name: hostname=kuku; port=2; hca_idx=1 , hostname=host1; port=3, hostname=host2 Note: port-name qualifier is not relevant for switches, but for HCA's only.	port-name: host-name=kuku; port=2; hca_idx=1
port regexp	One can define a regular expression so that only nodes with a matching node description will be chosen to the group. Note: This example shows how to choose nodes which their node description starts with 'SW'.	port-regexp: SW
	It is possible to specify one physical port to be chosen for matching nodes (there is no option to define a list or a range of ports). The given example will cause only nodes that match physical port 3 to be added to the group.	port-regexp: SW:3

Parameter	Description	Example
union rule	It is possible to define a rule that unites two different port groups. This means that all ports from both groups will be included in the united group.	union-rule: grp1, grp2
subtract rule	One can define a rule that subtracts one port group from another. The given rule, for example, will cause all the ports which are a part of grp1, but not included in grp2, to be chosen. In subtraction (unlike union), the order does matter, since the purpose is to subtract the second group from the first one. There is no option to define more than two groups for union/subtraction. However, one can unite/subtract groups which are a union or a subtraction themselves, as shown in the port groups policy file example.	subtract-rule: grp1, grp2

Predefined Port Groups

There are 3 predefined, automatically created port groups that are available for use, yet cannot be defined in the policy file (if a group in the policy is configured with the name of one of these predefined groups, the policy fails):

- **ALL** – a group that includes all nodes in the fabric
- **ALL_SWITCHES** – a group that includes all switches in the fabric
- **ALL_CAS** – a group that includes all HCAs in the fabric
- **ALL_ROUTERS** – a group that includes all routers in the fabric (supported in OpenSM starting from v4.9.0)

Port Groups Policy Examples

```
port-group
name: grp3
use: Subtract of groups grp1 and grp2
subtract-rule: grp1, grp2
end-port-group

port-group
name: grp1
port-guid: 0x281, 0x282, 0x283
end-port-group

port-group
name: grp2
port-guid-range: 0x282-0x286
port-name: hostname=server1 port=1
end-port-group

port-group
name: grp4
port-name: hostname=kika port=1 hca_idx=1
end-port-group

port-group
name: grp3
union-rule: grp3, grp4
end-port-group
```

Defining a Topologies Policy File

To define a topology policy file, set the parameter **topo_policy_file** in the OpenSM configuration file.

```
topo_policy_file /etc/opensm/conf/topo_policy_file.cfg
```

Configuring a Topology Policy

The topologies policy file details a list of topologies. The policy file should be composed of one or more paragraphs which define a topology. Each paragraph should begin with the line **topology** and end with the line **end-topology**.

For example:

```
topology
…topology qualifiers…
end-topology
```

Topology Qualifiers

> ⚠ Unlike topology and end-topology which do not require a colon, all qualifiers must end with a colon (':'). Also - a colon is a predefined mark that must not be used inside qualifier values. An inclusion of a column in the qualifier values will result in the policy's failure.

All topology qualifiers are mandatory. Absence of any of the below qualifiers will cause the policy parsing to fail.

Parameter	Description	Example
id	Topology ID. Supported values: Positive value. Must be unique.	**id: 1**
sw-grp	Name of the port group that includes all switches and switch ports to be used in this topology.	**sw-grp: ys_switches**
hca-grp	Name of the port group that includes all HCA's to be used in this topology.	**hca-grp: ys_hosts**

Configuration File per Routing Engine

Each engine in the routing chain can be provided by its own configuration file. Routing engine configuration file is the fraction of parameters defined in the main OpenSM configuration file. Some rules should be applied when defining a particular configuration file for a routing engine:

- Parameters that are not specified in specific routing engine configuration file are inherited from the main OpenSM configuration file.
- The following configuration parameters are taking effect only in the main OpenSM configuration file:
  - **qos** and **qos_*** settings like (**vl_arb** , **sl2vl** , etc.)
  - **lmc**
  - **routing_engine**

Defining a Routing Chain Policy File

In order to define a port group policy file, set the parameter **rch_policy_file** in the OpenSM configuration file.

```
rch_policy_file /etc/opensm/conf/chains_policy_file
```

First Routing Engine in the Chain

The first unicast engine in a routing chain must include all switches and HCAs in the fabric (topology ID must be 0). The path-bit parameter value is path-bit 0 and it cannot be changed.

Configuring a Routing Chains Policy

The routing chains policy file details the routing engines (and their fallback engines) used for the fabric's routing. The policy file should be composed of one or more paragraphs which defines an engine (or a fallback engine). Each paragraph should begin with the line **unicast-step** and end with the line **end-unicast-step**.

For example:

```
unicast-step
…routing engine qualifiers…
end-unicast-step
```

Routing Engine Qualifiers

> ⚠ Unlike unicast-step and end-unicast-step which do not require a colon, all qualifiers must end with a colon (**:**). Also - a colon is a predefined mark that must not be used inside qualifier values. An inclusion of a colon in the qualifier values will result in the policy's failure.

Parameter	Description	Example
id	Mandatory. • Legal values – **size_t** value (0 is illegal). • The engines in the policy chain are set according to an ascending id order, so it is highly crucial to verify that the id that is given to the engines match the order in which you would like the engines to be set.	**is: 1**
engine	This is a mandatory qualifier that describes the routing algorithm used within this unicast step. Currently, on the first phase of routing chains, legal values are **minhop** / **ftree** / **updn**.	**engine: minhop**
use	This is an optional qualifier that enables one to describe the usage of this unicast step. If undefined, an empty string is used as a default.	**use: ftree routing for for yellow stone nodes**
config	This is an optional qualifier that enables one to define a separate OpenSM config file for a specific unicast step. If undefined, all parameters are taken from main OpenSM configuration file.	**config: /etc/config/opensm2.cfg**
topology	Define the topology that this engine uses. • Legal value – ID of an existing topology that is defined in topologies policy (or zero that represents the entire fabric and not a specific topology). • Default value – If unspecified, a routing engine will relate to the entire fabric (as if topology zero was defined). • Notice: The first routing engine (the engine with the lowest id) MUST be configured with topology: 0 (entire fabric) or else, the routing chain parser will fail.	**topology: 1**

Parameter	Description	Example
`fallback-to`	This is an optional qualifier that enables one to define the current unicast step as a fallback to another unicast step. This can be done by defining the id of the unicast step that this step is a fallback to. • If undefined, the current unicast step is not a fallback. • If the value of this qualifier is a non-existent engine id, this step will be ignored. • A fallback step is meaningless if the step it is a fallback to did not fail. • It is impossible to define a fallback to a fall- back step (such definition will be ignored)	`-`
`path-bit`	This is an optional qualifier that enables one to define a specific LID offset to be used by the current unicast step. Setting lmc > 0 in main OpenSM configuration file is a prerequisite for assigning specific path-bit for the routing engine. Default value is 0 (if path-bit is not specified).	`Path-bit: 1`

Dump Files per Routing Engine

Each routing engine on the chain will dump its own data files if the appropriate `log_flags` is set (for instance `0x43`).

The files that are dumped by each engine are:

- `opensm-lid-matrix.dump`
- `opensm-lfts.dump`
- `opensm.fdbs`
- `opensm-subnet.lst`

These files should contain the relevant data for each engine topology.

> ⚠ `sl2vl` and `mcfdbs` files are dumped only once for the entire fabric and NOT by every routing engine.

- Each engine concatenates its ID and routing algorithm name in its dump files names, as follows:
  - `opensm-lid-matrix.2.minhop.dump`
  - `opensm.fdbs.3.ftree`
  - `opensm-subnet.4.updn.lst`
- If a fallback routing engine is used, both the routing engine that failed and the fallback engine that replaces it, dump their data.
  If, for example, engine 2 runs ftree and it has a fallback engine with 3 as its id that runs minhop, one should expect to find 2 sets of dump files, one for each engine:
  - `opensm-lid-matrix.2.ftree.dump`
  - `opensm-lid-matrix.3.minhop.dump`
  - `opensm.fdbs.2.ftree`
  - `opensm.fdbs.3.munhop`

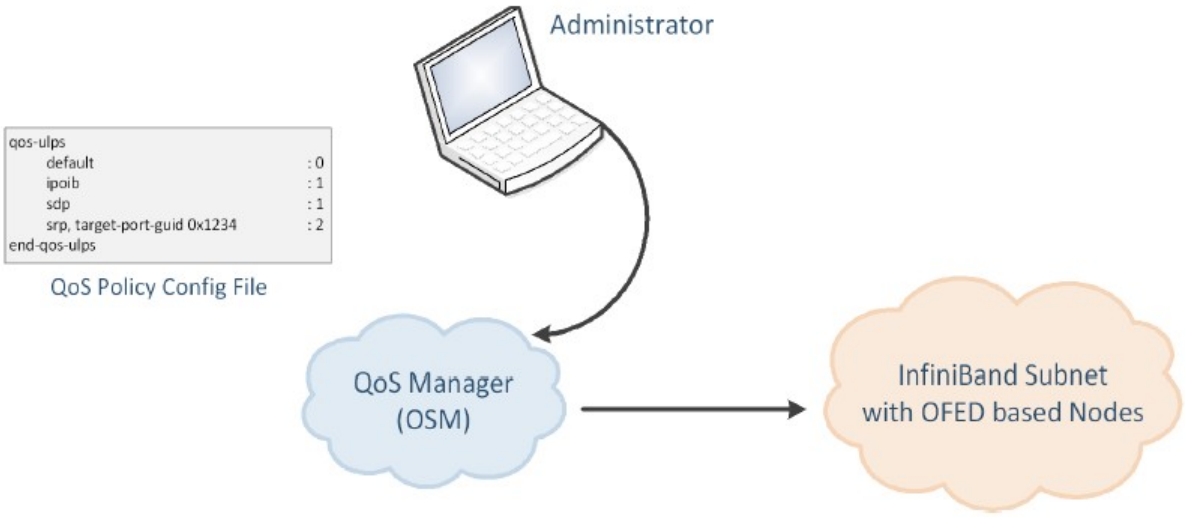
5.6.2.1.2.6 Unicast Routing Cache

Unicast routing cache prevents routing recalculation (which is a heavy task in a large cluster) when no topology change was detected during the heavy sweep, or when the topology change does not require new routing calculation (for example, when one or more CAs/RTRs/leaf switches going down, or one or more of these nodes coming back after being down).

5.6.2.1.2.7 Quality of Service Management in OpenSM

When Quality of Service (QoS) in OpenSM is enabled (using the `-Q` or `--qos` flags), OpenSM looks for a QoS Policy file. During fabric initialization and at every heavy sweep, OpenSM parses the QoS policy file, applies its settings to the discovered fabric elements, and enforces the provided policy on client requests. The overall flow for such requests is as follows:

- The request is matched against the defined matching rules such that the QoS level definition is found
- Given the QoS Level, a path(s) search is performed with the given restrictions imposed by that level



There are two ways to define QoS policy:

- Advanced – the advanced policy file syntax provides the administrator various ways to match a `PathRecord` / `MultiPathRecord` (PR/MPR) request, and to enforce various QoS constraints on the requested PR/MPR
- Simple – the simple policy file syntax enables the administrator to match PR/MPR requests by various ULPs and applications running on top of these ULPs

Advanced QoS Policy File

The QoS policy file has the following sections:

1. Port Groups (denoted by port-groups) - this section defines zero or more port groups that can be referred later by matching rules (see below). Port group lists ports by:

- Port GUID
- Port name, which is a combination of **NodeDescription** and IB port number
- PKey, which means that all the ports in the subnet that belong to partition with a given PKey belong to this port group
- Partition name, which means that all the ports in the subnet that belong to partition with a given name belong to this port group
- Node type, where possible node types are CA, SWITCH, ROUTER, ALL, and SELF (SM's port).

2. QoS Setup (denoted by **qos-setup**) - this section describes how to set up SL2VL and VL Arbitration tables on various nodes in the fabric. However, this is not supported in OFED. SL2VL and VLArb tables should be configured in the OpenSM options file (default location is **/var/cache/opensm/opensm.opts**).
3. QoS Levels (denoted by **qos-levels**) - each QoS Level defines Service Level (SL) and a few optional fields:
    - MTU limit
    - Rate limit
    - PKey
    - Packet lifetime
      When path(s) search is performed, it is done with regards to restriction that these QoS Level parameters impose. One QoS level that is mandatory to define is a DEFAULT QoS level. It is applied to a PR/MPR query that does not match any existing match rule. Similar to any other QoS Level, it can also be explicitly referred by any match rule.

- QoS Matching Rules (denoted by **qos-match-rules**) - each **PathRecord** / **MultiPathRecord** query that OpenSM receives is matched against the set of matching rules. Rules are scanned in order of appearance in the QoS policy file such as the first match takes precedence.
  Each rule has a name of QoS level that will be applied to the matching query. A default QoS level is applied to a query that did not match any rule.
  Queries can be matched by:
  - Source port group (whether a source port is a member of a specified group)
  - Destination port group (same as above, only for destination port)
  - PKey
  - QoS class
  - Service ID

  To match a certain matching rule, PR/MPR query has to match ALL the rule's criteria. However, not all the fields of the PR/MPR query have to appear in the matching rule.
  For instance, if the rule has a single criterion - Service ID, it will match any query that has this Service ID, disregarding rest of the query fields. However, if a certain query has only Service ID (which means that this is the only bit in the PR/MPR component mask that is on), it will not match any rule that has other matching criteria besides Service ID.

Simple QoS Policy Definition

Simple QoS policy definition comprises of a single section denoted by **qos-ulps**. Similar to the advanced QoS policy, it has a list of match rules and their QoS Level, but in this case a match rule has only one criterion: its goal is to match a certain ULP (or a certain application on top of this

ULP) PR/MPR request, and QoS Level has only one constraint – the service level (SL).
The simple policy section may appear in the policy file in combine with the advanced policy, or as a stand-alone policy definition. See more details and list of match rule criteria below.

Policy File Syntax Guidelines

- Leading and trailing blanks, as well as empty lines, are ignored, so the indentation in the example is just for better readability.
- Comments are started with the pound sign (#) and terminated by EOL.
- Any keyword should be the first non-blank in the line, unless it's a comment.
- Keywords that denote section/subsection start have matching closing keywords.
- Having a QoS Level named `DEFAULT` is a must – it is applied to PR/MPR requests that did not match any of the matching rules.
- Any section/subsection of the policy file is optional.

Examples of Advanced Policy Files

As mentioned earlier, any section of the policy file is optional, and the only mandatory part of the policy file is a default QoS Level.

Here is an example of the shortest policy file:

```
qos-levels
    qos-level
        name: DEFAULT
        sl: 0
    end-qos-level
end-qos-levels
```

Port groups section is missing because there are no match rules, which means that port groups are not referred anywhere, and there is no need defining them. And since this policy file doesn't have any matching rules, PR/MPR query will not match any rule, and OpenSM will enforce default QoS level. Essentially, the above example is equivalent to not having a QoS policy file at all.

The following example shows all the possible options and keywords in the policy file and their syntax:

```
#
# See the comments in the following example.
# They explain different keywords and their meaning.
#
port-groups

    port-group # using port GUIDs
        name: Storage
        # "use" is just a description that is used for logging
        #  Other than that, it is just a comment
        use: SRP Targets
        port-guid: 0x10000000000001, 0x10000000000005-0x1000000000FFFA
        port-guid: 0x1000000000FFFF
    end-port-group

    port-group
        name: Virtual Servers
        # The syntax of the port name is as follows:
        #   "node_description/Pnum".
        # node_description is compared to the NodeDescription of the node,
        # and "Pnum" is a port number on that node.
        port-name: “vs1 HCA-1/P1, vs2 HCA-1/P1”
    end-port-group

    # using partitions defined in the partition policy
    port-group
        name: Partitions
        partition: Part1
        pkey: 0x1234
    end-port-group

    # using node types: CA, ROUTER, SWITCH, SELF (for node that runs SM)
    # or ALL (for all the nodes in the subnet)
    port-group
        name: CAs and SM
        node-type: CA, SELF
```

```
    end-port-group

end-port-groups

qos-setup
    # This section of the policy file describes how to set up SL2VL and VL
    # Arbitration tables on various nodes in the fabric.
    # However, this is not supported in OFED - the section is parsed
    # and ignored. SL2VL and VLArb tables should be configured in the
    # OpenSM options file (by default - /var/cache/opensm/opensm.opts).
end-qos-setup

qos-levels

    # Having a QoS Level named "DEFAULT" is a must - it is applied to
    # PR/MPR requests that didn't match any of the matching rules.
    qos-level
        name: DEFAULT
        use: default QoS Level
        sl: 0
    end-qos-level

    # the whole set: SL, MTU-Limit, Rate-Limit, PKey, Packet Lifetime
    qos-level
        name: WholeSet
        sl: 1
        mtu-limit: 4
        rate-limit: 5
        pkey: 0x1234
        packet-life: 8
    end-qos-level

end-qos-levels

# Match rules are scanned in order of their appearance in the policy file.
# First matched rule takes precedence.
qos-match-rules

    # matching by single criteria: QoS class
    qos-match-rule
        use: by QoS class
        qos-class: 7-9,11
        # Name of qos-level to apply to the matching PR/MPR
        qos-level-name: WholeSet
    end-qos-match-rule

    # show matching by destination group and service id
    qos-match-rule
        use: Storage targets
        destination: Storage
        service-id: 0x10000000000001, 0x10000000000008-0x10000000000FFF
        qos-level-name: WholeSet
    end-qos-match-rule

    qos-match-rule
        source: Storage
        use: match by source group only
        qos-level-name: DEFAULT
    end-qos-match-rule
    qos-match-rule
        use: match by all parameters
        qos-class: 7-9,11
        source: Virtual Servers
        destination: Storage
        service-id: 0x0000000000010000-0x000000000001FFFF
        pkey: 0x0F00-0x0FFF
        qos-level-name: WholeSet
    end-qos-match-rule
end-qos-match-rules
```

Simple QoS Policy - Details and Examples

Simple QoS policy match rules are tailored for matching ULPs (or some application on top of a ULP) PR/MPR requests. This section has a list of per-ULP (or per-application) match rules and the SL that should be enforced on the matched PR/MPR query.

Match rules include:

- Default match rule that is applied to PR/MPR query that didn't match any of the other match rules
- IPoIB with a default PKey
- IPoIB with a specific PKey
- Any ULP/application with a specific Service ID in the PR/MPR query
- Any ULP/application with a specific PKey in the PR/MPR query
- Any ULP/application with a specific target IB port GUID in the PR/MPR query

Since any section of the policy file is optional, as long as basic rules of the file are kept (such as no referring to nonexistent port group, having default QoS Level, etc), the simple policy section (**qos-ulps**) can serve as a complete QoS policy file.

The shortest policy file in this case would be as follows:

```
qos-ulps
    default  : 0 #default SL
end-qos-ulps
```

It is equivalent to the previous example of the shortest policy file, and it is also equivalent to not having policy file at all. Below is an example of simple QoS policy with all the possible keywords:

```
qos-ulps
default                 :0 # default SL
sdp, port-num 30000     :0 # SL for application running on
                           # top of SDP when a destination
                           # TCP/IPport is 30000
sdp, port-num 10000-20000     : 0
sdp                     :1 # default SL for any other
                           # application running on top of SDP
rds                     :2 # SL for RDS traffic
ipoib, pkey 0x0001      :0 # SL for IPoIB on partition with
                           # pkey 0x0001
ipoib                   :4 # default IPoIB partition,
                           # pkey=0x7FFF
any, service-id 0x6234:6 # match any PR/MPR query with a
                           # specific Service ID
any, pkey 0x0ABC        :6 # match any PR/MPR query with a
                           # specific PKey
srp, target-port-guid 0x1234  : 5 # SRP when SRP Target is located
                           # on a specified IB port GUID
any, target-port-guid 0x0ABC-0xFFFFF : 6 # match any PR/MPR query
                           # with a specific target port GUID
end-qos-ulps
```

Similar to the advanced policy definition, matching of PR/MPR queries is done in order of appearance in the QoS policy file such as the first match takes precedence, except for the "default" rule, which is applied only if the query did not match any other rule. All other sections of the QoS policy file take precedence over the **qos-ulps** section.

That is, if a policy file has both **qos-match-rules** and **qos-ulps** sections, then any query is matched first against the rules in the **qos-match-rules** section, and only if there was no match, the query is matched against the rules in **qos-ulps** section.

Note that some of these match rules may overlap, so in order to use the simple QoS definition effectively, it is important to understand how each of the ULPs is matched.

IPoIB

IPoIB query is matched by PKey or by destination GID, in which case this is the GID of the multicast group that OpenSM creates for each IPoIB partition.

Default PKey for IPoIB partition is 0x7fff, so the following three match rules are equivalent:

```
ipoib:<SL>ipoib, pkey 0x7fff : <SL>
any, pkey 0x7fff : <SL>
```

SRP

Service ID for SRP varies from storage vendor to vendor, thus SRP query is matched by the target IB port GUID. The following two match rules are equivalent:

```
srp, target-port-guid 0x1234 : <SL>
```

```
any, target-port-guid 0x1234 : <SL>
```

Note that any of the above ULPs might contain target port GUID in the PR query, so in order for these queries not to be recognized by the QoS manager as SRP, the SRP match rule (or any match rule that refers to the target port GUID only) should be placed at the end of the **qos-ulps** match rules.

MPI

SL for MPI is manually configured by an MPI admin. OpenSM is not forcing any SL on the MPI traffic, which explains why it is the only ULP that did not appear in the **qos-ulps** section.

SL2VL Mapping and VL Arbitration

OpenSM cached options file has a set of QoS related configuration parameters, that are used to configure SL2VL mapping and VL arbitration on IB ports. These parameters are:

- Max VLs: the maximum number of VLs that will be on the subnet
- High limit: the limit of High Priority component of VL Arbitration table (IBA 7.6.9)
- VLArb low table: Low priority VL Arbitration table (IBA 7.6.9) template
- VLArb high table: High priority VL Arbitration table (IBA 7.6.9) template
- SL2VL: SL2VL Mapping table (IBA 7.6.6) template. It is a list of VLs corresponding to SLs 0-15 (Note that VL15 used here means drop this SL).

There are separate QoS configuration parameters sets for various target types: CAs, routers, switch external ports, and switch's enhanced port 0. The names of such parameters are prefixed by **qos_<type>_** string. Here is a full list of the currently supported sets:

- **qos_ca_** – QoS configuration parameters set for CAs.
- **qos_rtr_** – parameters set for routers.
- **qos_sw0_** – parameters set for switches' port 0.
- **qos_swe_** – parameters set for switches' external ports.

Here's the example of typical default values for CAs and switches' external ports (hard-coded in OpenSM initialization):

```
qos_ca_max_vls 15
qos_ca_high_limit 0
qos_ca_vlarb_high 0:4,1:0,2:0,3:0,4:0,5:0,6:0,7:0,8:0,9:0,10:0,11:0,12:0,13:0,14:0
qos_ca_vlarb_low 0:0,1:4,2:4,3:4,4:4,5:4,6:4,7:4,8:4,9:4,10:4,11:4,12:4,13:4,14:4
qos_ca_sl2vl 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,7
qos_swe_max_vls 15
qos_swe_high_limit 0
qos_swe_vlarb_high 0:4,1:0,2:0,3:0,4:0,5:0,6:0,7:0,8:0,9:0,10:0,11:0,12:0,13:0,14:0
qos_swe_vlarb_low 0:0,1:4,2:4,3:4,4:4,5:4,6:4,7:4,8:4,9:4,10:4,11:4,12:4,13:4,14:4
qos_swe_sl2vl 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,7
```

VL arbitration tables (both high and low) are lists of VL/Weight pairs. Each list entry contains a VL number (values from 0-14), and a weighting value (values 0-255), indicating the number of 64 byte units (credits) which may be transmitted from that VL when its turn in the arbitration occurs. A weight of 0 indicates that this entry should be skipped. If a list entry is programmed for VL15 or for a VL that is not supported or is not currently configured by the port, the port may either skip that entry or send from any supported VL for that entry.

The same VLs may be listed multiple times in the high- or low-priority arbitration tables, and, further, it can be listed in both tables. The limit of high-priority VLArb table (**qos_<type>_high_limit**) indicates the number of high-priority packets that can be transmitted

without an opportunity to send a low-priority packet. Specifically, the number of bytes that can be sent is **high_limit** times 4K bytes.

A **high_limit** value of 255 indicates that the byte limit is unbounded.

> ⚠ If the 255 value is used, the low priority VLs may be starved.

A value of 0 indicates that only a single packet from the high-priority table may be sent before an opportunity is given to the low-priority table.

Keep in mind that ports usually transmit packets of size equal to MTU. For instance, for 4KB MTU a single packet will require 64 credits, so in order to achieve effective VL arbitration for packets of 4KB MTU, the weighting values for each VL should be multiples of 64.

Below is an example of SL2VL and VL Arbitration configuration on subnet:

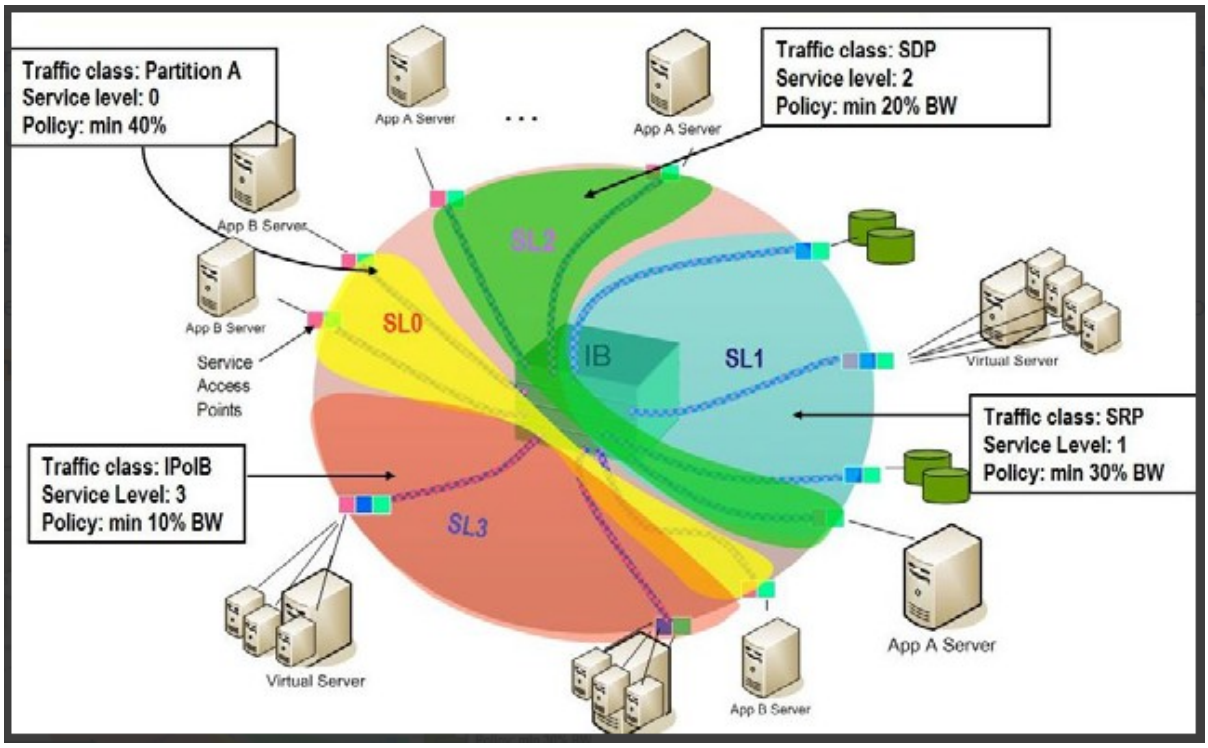
```
qos_ca_max_vls 15
qos_ca_high_limit 6
qos_ca_vlarb_high 0:4
qos_ca_vlarb_low 0:0,1:64,2:128,3:192,4:0,5:64,6:64,7:64
qos_ca_sl2vl 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,7
qos_swe_max_vls 15
qos_swe_high_limit 6
qos_swe_vlarb_high 0:4
qos_swe_vlarb_low 0:0,1:64,2:128,3:192,4:0,5:64,6:64,7:64
qos_swe_sl2vl 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,7
```

In this example, eight VLs—VL0 through VL7—are configured in the subnet. VL0 is defined as a high-priority VL and is limited to 6 × 4 KB, or 24 KB, per transmission burst. This configuration is suitable for traffic that requires low latency and uses a small MTU. The remaining VLs are defined as low-priority VLs with different weights, while VL4 is effectively disabled.

Deployment Example

The figure below shows an example of an InfiniBand subnet that has been configured by a QoS manager to provide different service levels for various ULPs.

QoS Deployment on InfiniBand Subnet Example



QoS Configuration Examples

The following are examples of QoS configuration for different cluster deployments. Each example provides the QoS level assignment and their administration via OpenSM configuration files.

Typical HPC Example: MPI and Lustre

Assignment of QoS Levels

- MPI
  - Separate from I/O load
  - Min BW of 70%
- Storage Control (Lustre MDS)
  - Low latency
- Storage Data (Lustre OST)
  - Min BW 30%

Administration

- MPI is assigned an SL via the command line
  `host1# mpirun –sl 0`
- OpenSM QoS policy file

```
qos-ulps
   default                                 :0 # default SL (for MPI)
   any, target-port-guid OST1,OST2,OST3,OST4   :1 # SL for Lustre OST
   any, target-port-guid MDS1,MDS2             :2 # SL for Lustre MDS
end-qos-ulps
```

  Note: In this policy file example, replace `OST*` and `MDS*` with the real port GUIDs.
- OpenSM options file

```
qos_max_vls 8
qos_high_limit 0
qos_vlarb_high 2:1
qos_vlarb_low 0:96,1:224
qos_sl2vl 0,1,2,3,4,5,6,7,15,15,15,15,15,15,15,15
```

EDC SOA (2-tier): IPoIB and SRP

The following is an example of QoS configuration for a typical enterprise data center (EDC) with service oriented architecture (SOA), with IPoIB carrying all application traffic and SRP used for storage.

QoS Levels

- Application traffic
  - IPoIB (UD and CM) and SDP
  - Isolated from storage
  - Min BW of 50%
- SRP
  - Min BW 50%
  - Bottleneck at storage nodes

Administration

- OpenSM QoS policy file

```
qos-ulps
   default                                 :0
   ipoib                                   :1
   sdp                                     :1
   srp, target-port-guid SRPT1,SRPT2,SRPT3     :2
end-qos-ulps
```

Note: In this policy file example, replace **SRPT*** with the real SRP Target port GUIDs.

- OpenSM options file

```
qos_max_vls 8
qos_high_limit 0
qos_vlarb_high 1:32,2:32
qos_vlarb_low 0:1,
qos_sl2vl 0,1,2,3,4,5,6,7,15,15,15,15,15,15,15,15
```

EDC (3-tier): IPoIB, RDS, SRP

The following is an example of QoS configuration for an enterprise data center (EDC), with IPoIB carrying all application traffic, RDS for database traffic, and SRP used for storage.

QoS Levels

- Management traffic (ssh)
  - IPoIB management VLAN (partition A)
  - Min BW 10%
- Application traffic
  - IPoIB application VLAN (partition B)
  - Isolated from storage and database
  - Min BW of 30%
- Database Cluster traffic

        - RDS
        - Min BW of 30%
    - SRP
        - Min BW 30%
        - Bottleneck at storage nodes

Administration

- OpenSM QoS policy file

```
qos-ulps
   default                                  :0
   ipoib, pkey 0x8001                       :1
   ipoib, pkey 0x8002                       :2
   rds                                      :3
   srp, target-port-guid SRPT1, SRPT2, SRPT3   :4
end-qos-ulps
```

Note: In the following policy file example, replace SRPT* with the real SRP Initiator port GUIDs.

- OpenSM options file

```
qos_max_vls 8
qos_high_limit 0
qos_vlarb_high 1:32,2:96,3:96,4:96
qos_vlarb_low 0:1
qos_sl2vl 0,1,2,3,4,5,6,7,15,15,15,15,15,15,15,15
```

- Partition configuration file

```
Default=0x7fff,ipoib : ALL=full;PartA=0x8001, sl=1, ipoib : ALL=full;
```

Enhanced QoS

Enhanced QoS provides a higher resolution of QoS at the service level (SL). Users can configure rate limit values per SL for physical ports, virtual ports, and port groups, using **enhanced_qos_policy_file** configuration parameter.

Valid values of this parameter:

- Full path to the policy file through which Enhanced QoS Manager is configured
- **null** – to disable the Enhanced QoS Manager (default value)

> ⚠ To enable Enhanced QoS Manager, QoS must be enabled in OpenSM.

Enhanced QoS Policy File

The policy file is comprised of three sections:

- **BW_NAMES**: Used to define bandwidth setting and name (currently, rate limit is the only setting). Bandwidth names can be used in **BW_RULES** and **VPORT_BW_RULES** sections. Bandwidth names are defined using the syntax:

```
<name> = <rate limit in 1Mbps units>
```

Example:

```
My_bandwidth = 50
```

- **BW_RULES** : Used to define the rules that map the bandwidth setting to a specific SL of a specific GUID.
  Bandwidth rules are defined using the syntax:

```
<guid>|<port group name> = <sl id>:<bandwidth name>, <sl id>:<bandwidth name>…
```

  Examples:

```
0x2c90000000025 = 5:My_bandwidth, 7:My_bandwidth
Port_grp1 = 3:My_bandwidth, 9:My_bandwidth
```

- **VPORT_BW_RULES** : Used to define the rules that map the bandwidth setting to a specific SL of a specific virtual port GUID.
  Bandwidth rules are defined using the syntax:

```
<guid>= <sl id>:<bandwidth name>, <sl id>:<bandwidth name>…
```

  Example:

```
0x2c90000000026= 5:My_bandwidth, 7:My_bandwidth
```

Special Keywords

- Keyword “all” allows setting a rate limit of all SLs to some BW for a specific physical or virtual port. It is possible to combine “all” with specific SL rate limits.
  Example:
  0x2c90000000025 = all:BW1,SL3:BW2
  In this case, SL3 will be assigned BW2 rate limit, while the rest of SLs get BW1 rate limit.
- "default" is a well-known name which can be used to define a default rule used for any GUID with no defined rule.
  If no default rule is defined, any GUID without a specific rule will be configured with unlimited rate limit for all SLs.
  Keyword “all” is also applicable to the default rule. Default rule is local to each section.

Special Subnet Manager Configuration Options

New SM configuration option **enhanced_qos_vport0_unlimit_default_rl** was added to **opensm.conf** .

The possible values for this configuration option are:

- TRUE: For specific virtual port0 GUID, SLs not mentioned in bandwidth rule will be set to unlimited bandwidth (0) regardless of the default rule of the **VPORT_BW_RULES** section. Virtual port0 GUIDs not mentioned in **VPORT_BW_SECTION** will be set to unlimited BW on all SLs.
- FALSE: The GUID of virtual port0 is treated as any other virtual port in **VPORT_BW_SECTION** . SM should be signaled by HUP once the option is changed.

Default: TRUE

Notes

- When rate limit is set to 0, it means that the bandwidth is unlimited.
- Any unspecified SL in a rule will be set to 0 (unlimited) rate limit automatically if no default rule is specified.
- Failure to complete policy file parsing leads to an undefined behavior. User must confirm no relevant error messages in SM log in order to ensure Enhanced QoS Manager is configured properly.
- A file with only **BW_NAMES** and **BW_RULES** keywords configures the network with an unlimited rate limit.
- HCA physical port GUID can be specified in **BW_RULES** and **VPORT_BW_RULES** sections.
- In **BW_RULES** section, the rate limit assigned to a specific SL will limit the total BW that can be sent through the PF on a given SL.
- In **VPORT_BW_RULES** section, the rate limit assigned to a specific SL will limit only the traffic sent from the IB interface corresponding to the physical port GUID (virtual port0 IB interface). The traffic sent from other virtual IB interfaces will not be limited if no specific rules are defined.

Policy File Example

All physical ports in the fabric are with a rate limit of 50Mb/s on SL1, except for GUID **0x2c90000000025**, which is configured with rate limit of 25Mb/s on SL1. In this example, the traffic on SLs (other than SL1) is unlimited.

All virtual ports in the fabric (except virtual port0 of all physical ports) will be rate-limited to 15Mbps for all SLs because of the default rule of **VPORT_BW_RULES** section.

Virtual port GUID **0x2c90000000026** is configured with a rate limit of 10Mb/s on SL3. The rest of the SLs on this virtual port will get a rate limit of 15 Mbps because of the default rule of **VPORT_BW_RULES** section.

```
------------------------------------------------------------------------
BW_NAMES
bw1 = 50
bw2 = 25
bw3 = 15
bw4 = 10

BW_RULES
default= 1:bw1
0x2c90000000025= 1:bw2

VPORT_BW_RULES
default= all:bw3
0x2c90000000026= 3:bw4

------------------------------------------------------------------------
```

5.6.2.1.2.8 Running OpenSM on a ConnectX-8 Host

Creating an SMI Interface

To run OpenSM or InfiniBand management tools such as **ibnetdiscover** on a host equipped with a ConnectX-8 adapter, first create an SMI interface for the selected InfiniBand device. This requirement applies to standard InfiniBand fabrics and is not limited to multiplane-subnet deployments.

1. Identify the available InfiniBand devices by running:

```
ibstat
```

2. Select the device on which OpenSM will run. For example, to create an SMI interface named `smi1` for `mlx5_0`, run:

```
sudo /opt/mellanox/iproute2/sbin/rdma dev add smi1 type SMI parent mlx5_0
```

   Replace `mlx5_0` with the name of the device selected in the previous step.

3. Run `ibstat` again to verify that the SMI interface was created successfully:

```
ibstat
```

   The new `smi1` interface should appear in the output with its associated ports:

```
CA 'smi1'
        CA type: MT4133
        Number of ports: 4
        Firmware version: 82.48.1000
        Hardware version: 0
        Node GUID: 0x70792d0300942df0
        System image GUID: 0x70792d0300942df0
        Port 1:
                State: Active
                Physical state: LinkUp
                Rate: 200
                Base lid: 51
                LMC: 0
                SM lid: 51
                Capability mask: 0xa750e848
                Port GUID: 0x70792d0300942df0
                Link layer: InfiniBand
        Port 2:
                State: Active
                Physical state: LinkUp
                Rate: 200
                Base lid: 51
                LMC: 0
                SM lid: 51
                Capability mask: 0xa750e848
                Port GUID: 0x70792d0300942df0
                Link layer: InfiniBand
        Port 3:
                State: Active
                Physical state: LinkUp
                Rate: 200
                Base lid: 51
                LMC: 0
                SM lid: 51
                Capability mask: 0xa750e848
                Port GUID: 0x70792d0300942df0
                Link layer: InfiniBand
        Port 4:
                State: Active
                Physical state: LinkUp
                Rate: 200
                Base lid: 51
                LMC: 0
                SM lid: 51
                Capability mask: 0xa750e848
                Port GUID: 0x70792d0300942df0
                Link layer: InfiniBand
```

5.6.2.1.2.9 Routing Configuration for Multiplane Subnets

When configuring routing for multiplane subnets in XDR topologies, adhere to the following rules and restrictions:

- Routing mode: For XDR topologies, only the `ar_updn` routing mode is supported. The `ar_ftree` routing mode is no longer supported for XDR topologies.
- Asymmetric flow: The `ar_tree_asymmetric_flow` parameter only accepts the following values: `0`, `1`, or `3`.

- Root GUID: The **root_guid** file must be provided manually by the customer.

> For further information on how to set up Adaptive Routing (AR) and SHIELD, please refer to the *Adaptive Routing, HBF, and SHIELD* documentation.

5.6.2.1.2.10 Adaptive Routing Manager and Self-Healing Networking

Adaptive Routing Manager supports advanced InfiniBand features; Adaptive Routing (AR) and Self-Healing Networking.

For information on how to set up AR and Self-Healing Networking, please refer to HowTo Configure Adaptive Routing and Self-Healing Networking Community post.

DOS MAD Prevention

DOS MAD prevention is achieved by assigning a threshold for each agent's RX. Agent's RX threshold provides a protection mechanism to the host memory by limiting the agents' RX with a threshold. Incoming MADs above the threshold are dropped and are not queued to the agent's RX.

To enable DOS MAD Prevention:

1. Go to **/etc/modprobe.d/mlnx.conf**.
2. Add to the file the option below.

```
ib_umad enable_rx_threshold 1
```

The threshold value can be controlled from the user-space via **libibumad**.

To change the value, use the following API:

```
int umad_update_threshold(int fd, int threshold);

@fd: file descriptor, agent's RX associated to this fd.
@threshold: new threshold value
```

5.6.2.1.2.11 IB Router Support in OpenSM

In order to enable the IB router in OpenSM, the following parameters should be configured:

IB Router Parameters for OpenSM

Parameter	Description	Default Value
rtr_pr_flow_label	Defines whether the SM should create alias GUIDs required for router support for each port. Defines flow label value to use in response for path records related to the router.	0 (Disabled)
rtr_pr_tclass	Defines TClass value to use in response for path records related to the router	0
rtr_pr_sl	Defines SL value to use in response for path records related to router.	0

Parameter	Description	Default Value
rtr_p_mtu	Defines MTU value to use in response for path records related to the router.	4 (IB_MTU_LEN_2048)
rtr_pr_rate	Defines rate value to use in response for path records related to the router.	16 (IB_PATH_RECORD_RATE_100_GBS)

5.6.2.1.2.12 OpenSM Activity Report

OpenSM can produce an activity report in a form of a dump file which details the different activities done in the SM. Activities are divided into subjects. The OpenSM Supported Activities table below specifies the different activities currently supported in the SM activity report. Reporting of each subject can be enabled individually using the configuration parameter activity_report_subjects :

- Valid values –
  - Comma separated list of subjects to dump. The current supported subjects are:
    - "mc" - activity IDs 1, 2 and 8
    - "prtn" - activity IDs 3, 4, and 5
    - "virt" - activity IDs 6 and 7
    - "routing" - activity IDs 8-12
  - Two predefined values can be configured as well:
    - "all" - dump all subjects
    - "none" - disable the feature by dumping none of the subjects
- Default value: "none"

OpenSM Supported Activities

ACtivity ID	Activity Name	Additional Fields	Comments	Description
1	mcm_member	• MLid • MGid • Port Guid • Join State	Join state: • 1 - Join • -1 - Leave	Member joined/left MC group
2	mcg_change	• MLid • MGid • Change	Change: • 0 - Create • 1 - Delete	MC group created/deleted
3	prtn_guid_add	• Port Guid • PKey • Block index • Pkey Index		GUID added to partition
4	prtn_create	-PKey • Prtn Name		Partition created
5	prtn_delete	• PKey • Delete Reason	Delete Reason: • 0 – empty prtn • 1 – duplicate prtn • 2 – sm shutdown	Partition deleted

ACtivity ID	Activity Name	Additional Fields	Comments	Description
6	`port_virt_discover`	• Port Guid • Top Index		Port virtualization discovered
7	`vport_state_change`	• Port Guid • VPort Guid • VPort Index • VNode Guid • VPort State	VPort State: • 1 - Down • 2 - Init • 3 - ARMED • 4 - Active	Vport state changed
8	`mcg_tree_calc`	mlid		MCast group tree calculated
9	`routing_succeed`	routing engine name		Routing done successfully
10	`routing_failed`	routing engine name		Routing failed
11	`ucast_cache_invalidated`			ucast cache invalidated
12	`ucast_cache_routing_done`			ucast cache routing done

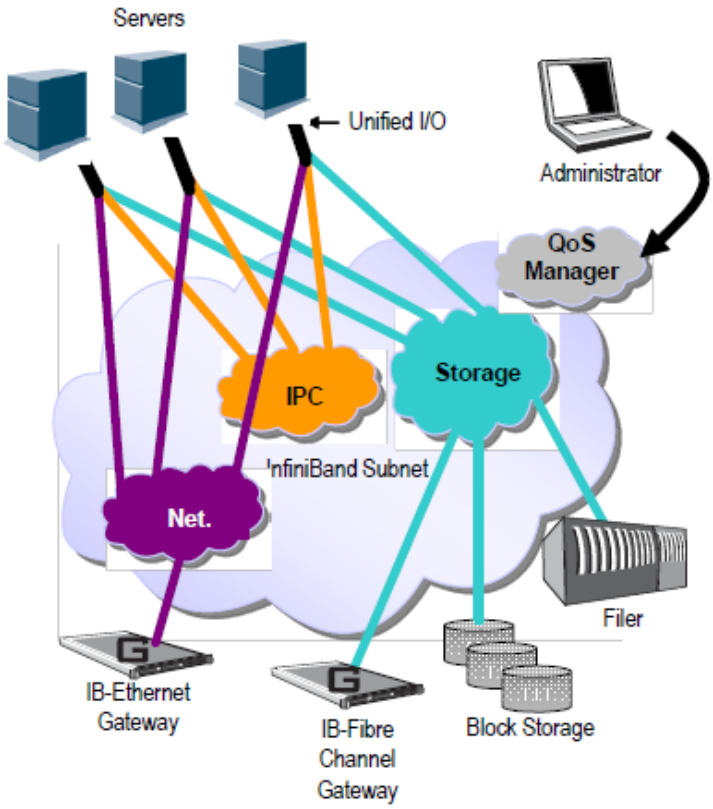
5.6.2.1.2.13 Offsweep Balancing

When working with minhop/dor/updn, subnet manager can re-balance routing during idle time (between sweeps).

- `offsweep_balancing_enabled` – enables/disables the feature. Examples:
  - `offsweep_balancing_enabled = TRUE`
  - `offsweep_balancing_enabled = FALSE` (default)
- `offsweep_balancing_window` – defines window of seconds to wait after sweep before starting the re-balance process. Applicable only if `offsweep_balancing_enabled=TRUE`. Example:
  - `offsweep_balancing_window = 180` (default)

5.6.2.1.3 InfiniBand QoS

Quality of service (QoS) requirements stem from the realization of I/O consolidation over an IB network. As multiple applications and ULPs share the same fabric, a means is needed to control their use of network resources.



The basic need is to differentiate the service levels provided to different traffic flows, such that a policy can be enforced and can control each flow utilization of fabric resources.

The InfiniBand Architecture Specification defines several hardware features and management interfaces for supporting QoS:

Up to 15 Virtual Lanes (VL) carry traffic in a non-blocking manner

- Arbitration between traffic of different VLs is performed by a two-priority-level weighted round robin arbiter. The arbiter is programmable with a sequence of (VL, weight) pairs and a maximal number of high priority credits to be processed before low priority is served
- Packets carry class of service marking in the range 0 to 15 in their header SL field
- Each switch can map the incoming packet by its SL to a particular output VL, based on a programmable table VL=SL-to-VL-MAP (in-port, out-port, SL)
- The Subnet Administrator controls the parameters of each communication flow by providing them as a response to Path Record (PR) or Multi-path Record (MPR) queries

DiffServ architecture (IETF RFC 2474 & 2475) is widely used in highly dynamic fabrics. The following subsections provide the functional definition of the various software elements that enable a DiffServ-like architecture over the NVIDIA OFED software stack.

5.6.2.1.3.1 QoS Architecture

QoS functionality is split between the SM/SA, CMA and the various ULPs. We take the "chronology approach" to describe how the overall system works.

1. The network manager (human) provides a set of rules (policy) that define how the network is being configured and how its resources are split to different QoS-Levels. The policy also define how to decide which QoS-Level each application or ULP or service use.
2. The SM analyzes the provided policy to see if it is realizable and performs the necessary fab- ric setup. Part of this policy defines the default QoS-Level of each partition. The SA is enhanced to match the requested Source, Destination, QoS-Class, Service-ID, PKey against the policy, so clients (ULPs, programs) can obtain a policy enforced QoS. The SM may also set up partitions with appropriate IPoIB broadcast group. This broadcast group carries its QoS attributes: SL, MTU, RATE, and Packet Lifetime.
3. IPoIB is being setup. IPoIB uses the SL, MTU, RATE and Packet Lifetime available on the multicast group which forms the broadcast group of this partition.
4. MPI which provides non IB based connection management should be configured to run using hard coded SLs. It uses these SLs for every QP being opened.
5. ULPs that use CM interface (like SRP) have their own pre-assigned Service-ID and use it while obtaining **PathRecord** / **MultiPathRecord** (PR/MPR) for establishing connections. The SA receiving the PR/MPR matches it against the policy and returns the appropriate PR/MPR including SL, MTU, RATE and Lifetime.
6. ULPs and programs (e.g. SDP) use CMA to establish RC connection provide the CMA the target IP and port number. ULPs might also provide QoS-Class. The CMA then creates Service-ID for the ULP and passes this ID and optional QoS-Class in the PR/MPR request. The resulting PR/MPR is used for configuring the connection QP.

PathRecord and Multi Path Record Enhancement for QoS:

As mentioned above, the **PathRecord** and **MultiPathRecord** attributes are enhanced to carry the Service-ID which is a 64bit value. A new field QoS-Class is also provided.

A new capability bit describes the SM QoS support in the SA class port info. This approach provides an easy migration path for existing access layer and ULPs by not introducing new set of PR/MPR attributes.

5.6.2.1.3.2 Supported Policy

The QoS policy, which is specified in a stand-alone file, is divided into the following four subsections:

Port Group

A set of CAs, Routers or Switches that share the same settings. A port group might be a partition defined by the partition manager policy, list of GUIDs, or list of port names based on **NodeDescription**.

Fabric Setup

Defines how the SL2VL and VLArb tables should be set up.

> ⚠ In OFED this part of the policy is ignored. SL2VL and VLArb tables should be configured in the OpenSM options file (opensm.opts).

QoS-Levels Definition

This section defines the possible sets of parameters for QoS that a client might be mapped to. Each set holds SL and optionally: Max MTU, Max Rate, Packet Lifetime and Path Bits.

> ⚠ Path Bits are not implemented in OFED.

Matching Rules

A list of rules that match an incoming PR/MPR request to a QoS-Level. The rules are processed in order such as the first match is applied. Each rule is built out of a set of match expressions which should all match for the rule to apply. The matching expressions are defined for the following fields:

- SRC and DST to lists of port groups
- Service-ID to a list of Service-ID values or ranges
- QoS-Class to a list of QoS-Class values or ranges

5.6.2.1.3.3 CMA Features

The CMA interface supports Service-ID through the notion of port space as a prefix to the port number, which is part of the sockaddr provided to **rdma_resolve_add()**. The CMA also allows the ULP (like SDP) to propagate a request for a specific QoS-Class. The CMA uses the provided QoS-Class and Service-ID in the sent PR/MPR.

IPoIB

IPoIB queries the SA for its broadcast group information and uses the SL, MTU, RATE and Packet Lifetime available on the multicast group which forms this broadcast group.

SRP

The current SRP implementation uses its own CM callbacks (not CMA). So SRP fills in the Service-ID in the PR/MPR by itself and use that information in setting up the QP.

SRP Service-ID is defined by the SRP target I/O Controller (it also complies with IBTA Service- ID rules). The Service-ID is reported by the I/O Controller in the **ServiceEntries** DMA attribute and should be used in the PR/MPR if the SA reports its ability to handle QoS PR/MPRs.

5.6.2.1.4 IP Over InfiniBand

5.6.2.1.4.1 Upper Layer Protocol (ULP)

The IP over IB (IPoIB) ULP driver is a network interface implementation over InfiniBand. IPoIB encapsulates IP datagrams over an InfiniBand Datagram transport service. The IPoIB driver, ib_ipoib, exploits the following capabilities:

- VLAN simulation over an InfiniBand network via child interfaces
- High availability (HA) via Bonding
- Varies MTU values:
    - up to 4k in Datagram mode
- Uses any ConnectX® IB ports (one or two)
- Inserts IP/UDP/TCP checksum on outgoing packets
- Calculates checksum on received packets

- Support net device TSO through ConnectX® LSO capability to defragment large datagrams to MTU quantas.

IPoIB also supports the following software based enhancements:

- Giant Receive Offload
- NAPI
- Ethtool support

5.6.2.1.4.2 Enhanced IPoIB

Enhanced IPoIB feature enables offloading ULP basic capabilities to a lower vendor specific driver, in order to optimize IPoIB data path. This will allow IPoIB to support multiple stateless offloads, such as RSS/TSS, and better utilize the features supported, enabling IPoIB datagram to reach peak performance in both bandwidth and latency.

Enhanced IPoIB supports/performs the following:

- Stateless offloads (RSS, TSS)
- Multi queues
- Interrupt moderation
- Multi partitions optimizations
- Sharing send/receive Work Queues
- Vendor specific optimizations
- UD mode only

5.6.2.1.4.3 Port Configuration

The physical port MTU (indicates the port capability) default value is 4k, whereas the IPoIB port MTU ("logical" MTU) default value is 2k as it is set by the OpenSM.
To change the IPoIB MTU to 4k, edit the OpenSM partition file in the section of IPoIB setting as follow:

```
Default=0xffff, ipoib, mtu=5 : ALL=full;
```

Where:

- **mtu=5** indicates that all IPoIB ports in the fabric are using 4k MTU, (**mtu=4** indicates 2k MTU)

5.6.2.1.4.4 IPoIB Configuration

Unless you have run the installation script mlnxofedinstall with the flag '-n', then IPoIB has not been configured by the installation. The configuration of IPoIB requires assigning an IP address and a subnet mask to each HCA port, like any other network adapter card (i.e., you need to prepare a file called ifcfg-ib<n> for each port). The first port on the first HCA in the host is called interface ib0, the second port is called ib1, and so on.

IPoIB configuration can be based on DHCP or on a static configuration that you need to supply (see below). You can also apply a manual configuration that persists only until the next reboot or driver restart (see below).

IPoIB Configuration Based on DHCP

Setting an IPoIB interface configuration based on DHCP is performed similarly to the configuration of Ethernet interfaces. In other words, you need to make sure that IPoIB configuration files include the following line:

- For RedHat:

```
BOOTPROTO=dhcp
```

- For SLES:

```
BOOTPROTO='dchp'
```

> If IPoIB configuration files are included, **ifcfg-ib<n>** files will be installed under:
> - **/etc/sysconfig/network-scripts/** on a RedHat machine
> - **/etc/sysconfig/network/** on a SuSE machine.

> A patch for DHCP may be required for supporting IPoIB. For further information, please see the REAME file available under the **docs/dhcp/** directory.

> Red Hat Enterprise Linux 7 supports assigning static IP addresses to InfiniBand IPoIB interfaces. However, as these interfaces do not have a normal hardware Ethernet address, a different method of specifying a unique identifier for the IPoIB interface must be used. The standard is to use the option dhcp-client-identifier= construct to specify the IPoIB interface's dhcp-client-identifier field. The DHCP server host construct supports at most one hardware Ethernet and one dhcp-client-identifier entry per host stanza. However, there may be more than one fixed-address entry and the DHCP server will automatically respond with an address that is appropriate for the network that the DHCP request was received on.

Standard DHCP fields holding MAC addresses are not large enough to contain an IPoIB hardware address. To overcome this problem, DHCP over InfiniBand messages convey a client identifier field used to identify the DHCP session. This client identifier field can be used to associate an IP address with a client identifier value, such that the DHCP server will grant the same IP address to any client that conveys this client identifier.

The length of the client identifier field is not fixed in the specification. For the *NVIDIA OFED for Linux* package, it is recommended to have IPoIB use the same format that FlexBoot uses for this client identifier.

DHCP Server

In order for the DHCP server to provide configuration records for clients, an appropriate configuration file needs to be created. By default, the DHCP server looks for a configuration file called **dhcpd.conf** under **/etc** . You can either edit this file or create a new one and provide its full path to the DHCP server using the -cf flag (See a file example at docs/dhcpd.conf).
The DHCP server must run on a machine which has loaded the IPoIB module. To run the DHCP server from the command line, enter:

```
dhcpd <IB network interface name> -d
```

Example:

```
host1# dhcpd ib0 -d
```

DHCP Client (Optional)

A DHCP client can be used if you need to prepare a diskless machine with an IB driver.

In order to use a DHCP client identifier, you need to first create a configuration file that defines the DHCP client identifier.

Then run the DHCP client with this file using the following command:

```
dhclient -cf <client conf file> <IB network interface name>
```

Example of a configuration file for the ConnectX (PCI Device ID 26428), called **dhclient.conf**:

```
The value indicates a hexadecimal number interface "ib1" {
send dhcp-client-identifier
ff:00:00:00:00:02:00:00:02:c9:00:00:02:c9:03:00:00:10:39;
}
```

Example of a configuration file for InfiniHost III Ex (PCI Device ID 25218), called **dhclient.conf**:

```
The value indicates a hexadecimal number interface "ib1" {
send dhcp-client-identifier
20:00:55:04:01:fe:80:00:00:00:00:00:00:02:c9:02:00:23:13:92;
}
```

To use the configuration file, run:

```
host1# dhclient -cf dhclient.conf ib1
```

Static IPoIB Configuration

If you wish to use an IPoIB configuration that is not based on DHCP, you need to supply the installation script with a configuration file (using the '-n' option) containing the full IP configuration. The IPoIB configuration file can specify either or both of the following data for an IPoIB interface:

- A static IPoIB configuration
- An IPoIB configuration based on an Ethernet configuration
  See your Linux distribution documentation for additional information about configuring IP addresses.

The following code lines are an excerpt from a sample IPoIB configuration file:

```
# Static settings; all values provided by this file
IPADDR_ib0=10.4.3.175
NETMASK_ib0=255.255.0.0
NETWORK_ib0=10.4.0.0
BROADCAST_ib0=10.4.255.255
ONBOOT_ib0=1
# Based on eth0; each '*' will be replaced with a corresponding octet
# from eth0.
LAN_INTERFACE_ib0=eth0
IPADDR_ib0=10.4.'*'.'*'
NETMASK_ib0=255.255.0.0
NETWORK_ib0=10.4.0.0
BROADCAST_ib0=10.4.255.255
ONBOOT_ib0=1
# Based on the first eth<n> interface that is found (for n=0,1,...);
# each '*' will be replaced with a corresponding octet from eth<n>.
LAN_INTERFACE_ib0=
IPADDR_ib0=10.4.'*'.'*'
NETMASK_ib0=255.255.0.0
NETWORK_ib0=10.4.0.0
BROADCAST_ib0=10.4.255.255
ONBOOT_ib0=1
```

Manually Configuring IPoIB

> ⚠ This manual configuration persists only until the next reboot or driver restart.

To manually configure IPoIB for the default IB partition (VLAN), perform the following steps:

1. Configure the interface by entering the ifconfig command with the following items:
   - The appropriate IB interface (ib0, ib1, etc.)
   - The IP address that you want to assign to the interface
   - The netmask keyword
   - The subnet mask that you want to assign to the interface

   The following example shows how to configure an IB interface:

   ```
   host1$ ifconfig ib0 10.4.3.175 netmask 255.255.0.0
   ```

2. (Optional) Verify the configuration by entering the ifconfig command with the appropriate interface identifier **ib#** argument.

   The following example shows how to verify the configuration:

   ```
   host1$ ifconfig ib0
   b0  Link encap:UNSPEC  HWaddr 80-00-04-04-FE-80-00-00-00-00-00-00-00-00-00-00
   inet addr:10.4.3.175  Bcast:10.4.255.255  Mask:255.255.0.0
   UP BROADCAST MULTICAST  MTU:65520  Metric:1
   RX packets:0 errors:0 dropped:0 overruns:0 frame:0
   TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
   collisions:0 txqueuelen:128
   RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
   ```

3. Repeat the first two steps on the remaining interface(s).

5.6.2.1.4.5 Sub-interfaces

You can create sub-interfaces for a primary IPoIB interface to provide traffic isolation. Each such sub-interface (also called a child interface) has a different IP and network addresses from the primary (parent) interface. The default Partition Key (PKey), ff:ff, applies to the primary (parent) interface.

This section describes how to:

- Create a subinterface

- Remove a subinterface

Creating a Subinterface

In the following procedure, ib0 is used as an example of an IB sub-interface.

To create a child interface (sub-interface), follow this procedure:

1. Decide on the PKey to be used in the subnet (valid values can be 0 or any 16-bit unsigned value). The actual PKey used is a 16-bit number with the most significant bit set. For example, a value of 1 will give a PKey with the value 0x8001.
2. Create a child interface by running:

   ```
   host1$ echo <PKey> > /sys/class/net/<IB subinterface>/create_child
   ```

   Example:

   ```
   host1$ echo 1 > /sys/class/net/ib0/create_child
   ```

   This will create the interface ib0.8001.
3. Verify the configuration of this interface by running:

   ```
   host1$ ifconfig <subinterface>.<subinterface PKey>
   ```

   Using the example of the previous step:

   ```
   host1$ ifconfig ib0.8001
   ib0.8001  Link encap:UNSPEC  HWaddr 80-00-00-4A-FE-80-00-00-00-00-00-00-00-00-00-00
   BROADCAST MULTICAST  MTU:2044  Metric:1
   RX packets:0 errors:0 dropped:0 overruns:0 frame:0
   TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
   collisions:0 txqueuelen:128
   RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
   ```
4. As can be seen, the interface does not have IP or network addresses. To configure those, you should follow the manual configuration procedure described in "Manually Configuring IPoIB" section above.
5. To be able to use this interface, a configuration of the Subnet Manager is needed so that the PKey chosen, which defines a broadcast address, be recognized.

Removing a Subinterface

To remove a child interface (subinterface), run:

```
echo <subinterface PKey> /sys/class/net/<ib_interface>/delete_child
```

Using the example of the second step from the previous chapter:

```
echo 0x8001 > /sys/class/net/ib0/delete_child
```

Note that when deleting the interface you must use the PKey value with the most significant bit set (e.g., 0x8000 in the example above).

5.6.2.1.4.6 Verifying IPoIB Functionality

To verify your configuration and IPoIB functionality are successful, perform the following steps:

1. Verify the IPoIB functionality by using the **ifconfig** command.
   The following example shows how two IB nodes are used to verify IPoIB functionality. In the following example, IB node 1 is at 10.4.3.175, and IB node 2 is at 10.4.3.176:

   ```
   host1# ifconfig ib0 10.4.3.175 netmask 255.255.0.0
   host2# ifconfig ib0 10.4.3.176 netmask 255.255.0.0
   ```

2. Enter the ping command from 10.4.3.175 to 10.4.3.176.
3. The following example shows how to enter the ping command:

   ```
   host1# ping -c 5 10.4.3.176
   PING 10.4.3.176 (10.4.3.176) 56(84) bytes of data.
   64 bytes from 10.4.3.176: icmp_seq=0 ttl=64 time=0.079 ms
   64 bytes from 10.4.3.176: icmp_seq=1 ttl=64 time=0.044 ms
   64 bytes from 10.4.3.176: icmp_seq=2 ttl=64 time=0.055 ms
   64 bytes from 10.4.3.176: icmp_seq=3 ttl=64 time=0.049 ms
   64 bytes from 10.4.3.176: icmp_seq=4 ttl=64 time=0.065 ms
   --- 10.4.3.176 ping statistics ---
   5 packets transmitted, 5 received, 0% packet loss, time 3999ms rtt min/avg/max/mdev = 0.044/0.058/0.079/
   0.014 ms, pipe 2
   ```

5.6.2.1.4.7 Bonding IPoIB

To create an interface configuration script for the ibX and bondX interfaces, you should use the standard syntax (depending on your OS).

Bonding of IPoIB interfaces is accomplished in the same manner as would bonding of Ethernet interfaces: via the Linux Bonding Driver.

- Network Script files for IPoIB slaves are named after the IPoIB interfaces (e.g: ifcfg- ib0)
- The only meaningful bonding policy in IPoIB is HA (bonding mode number 1, or active-backup)
- Bonding parameter "fail_over_mac" is meaningless in IPoIB interfaces, hence, the only supported value is the default: 0

For a persistent bonding IPoIB Network configuration, use the same Linux Network Scripts semantics, with the following exceptions/ additions:

- In the bonding master configuration file (e.g: ifcfg-bond0), in addition to Linux bonding semantics, use the following parameter: MTU=65520

  For IPoIB slaves, use MTU=2044. If you do not set the correct MTU or do not set MTU at all, performance of the interface might decrease.

- In the bonding slave configuration file (e.g: ifcfg-ib0), use the same Linux Network Scripts semantics. In particular: DEVICE=ib0

- In the bonding slave configuration file (e.g: ifcfg-ib0.8003), the line TYPE=InfiniBand is necessary when using bonding over devices configured with partitions (**p_key**)
- For RHEL, add the following lines in **/etc/modprobe.b/bond.conf** :

```
alias bond0 bonding
```

- For SLES users:
  It is necessary to update the **MANDATORY_DEVICES** environment variable in **/etc/sysconfig/network/config** with the names of the IPoIB slave devices (e.g. ib0, ib1, etc.). Otherwise, bonding master may be created before IPoIB slave interfaces at boot time.
  It is possible to have multiple IPoIB bonding masters and a mix of IPoIB bonding master and Ethernet bonding master. However, It is NOT possible to mix Ethernet and IPoIB slaves under the same bonding master.

⚠ Restarting openibd does no keep the bonding configuration via Network Scripts. You have to restart the network service in order to bring up the bonding master. After the configuration is saved, restart the network service by running: /etc/init.d/network restart.

5.6.2.1.4.8 Dynamic PKey Change

Dynamic PKey change means the PKey can be changed (add/removed) in the SM database and the interface that is attached to that PKey is updated immediately without the need to restart the driver.
If the PKey is already configured in the port by the SM, the child-interface can be used immediately. If not, the interface will be ready to use only when SM adds the relevant PKey value to the port after the creation of the child interface. No additional configuration is required once the child-interface is created.

5.6.2.1.4.9 Precision Time Protocol (PTP) over IPoIB

This feature allows for accurate synchronization between the distributed entities over the network. The synchronization is based on symmetric Round Trip Time (RTT) between the master and slave devices.

This feature is enabled by default, and is also supported over PKey interfaces.

For more on the PTP feature, refer to Running Linux PTP with ConnectX-4/ConnectX-5/ ConnectX-6 Community post.

5.6.2.1.4.10 One Pulse Per Second (1PPS) over IPoIB

1PPS is a time synchronization feature that allows the adapter to be able to send or receive 1 pulse per second on a dedicated pin on the adapter card using an SMA connector (SubMiniature version A). Only one pin is supported and could be configured as 1PPS in or 1PPS out.

5.6.2.1.5 Advanced Transport

5.6.2.1.5.1 Atomic Operations in mlx5 Driver

The mlx5 driver supports atomic operations; however, due to potential endianness mismatches between requester and responder systems, specific configuration is required to ensure correct behavior.

To enable atomic operations in environments where such an endianness contradiction may exist, the application must explicitly configure the Queue Pair (QP) during creation.

Use the **ibv_create_qp()** function and set the following flag in the **create_flags** field:

```
IBV_QP_CREATE_ATOMIC_BE_REPLY
```

This flag ensures that the QP is configured to handle big-endian formatted atomic replies, allowing atomic operations to complete correctly across systems with differing byte orders.

5.6.2.1.5.2 XRC - eXtended Reliable Connected Transport Service for InfiniBand

XRC (eXtended Reliable Connected transport) is an InfiniBand transport type designed to improve scalability in large-scale deployments by reducing the number of required QPs and associated memory resources. It enables more efficient *all-to-all* process connectivity, particularly in environments with many multicore end-nodes.

By decoupling the send and receive sides of a QP, XRC significantly reduces memory usage and improves resource management, making it ideal for large clusters.

For detailed technical background, refer to "Annex A14 Supplement to InfiniBand Architecture Specification Volume 1.2.1".

API Usage

A new set of verbs APIs has been introduced to support XRC in user-space applications. While the legacy XRC API is still supported in both binary and source form, it is deprecated. Developers are strongly encouraged to use the new API.

Key Verbs for XRC:

- **ibv_open_xrcd()** / **ibv_close_xrcd()** – Open and close XRC domain objects
- **ibv_create_srq_ex()** – Create an SRQ with extended attributes
- **ibv_get_srq_num()** – Retrieve SRQ number
- **ibv_create_qp_ex()** – Create a QP with extended attributes (including XRC target/source)
- **ibv_open_qp()** – Open an existing QP from a remote XRC domain

Testing and Reference

For basic testing and example usage of the XRC transport, refer to the **ibv_xsrq_pingpong** utility. It provides a working reference for integrating XRC into user applications.

For complete documentation of each API function and its options, consult the corresponding man pages.

5.6.2.1.5.3 Dynamically Connected Transport

Dynamically Connected Transport (DCT) is an advanced RDMA transport service designed to improve scalability and efficiency in environments with sparse or dynamic communication patterns. It allows Reliable Connection (RC)-type QPs to dynamically connect and disconnect from remote nodes as needed, rather than maintaining persistent connections.

This on-demand connection model reduces system-wide resource usage and enhances performance by:

- Minimizing the total number of QPs required.
- Lowering the memory footprint.
- Reducing connection setup overhead.
- Improving on-chip cache utilization.

As a result, DCT is particularly beneficial for large-scale or highly dynamic deployments where traditional RC QP scaling becomes a bottleneck.

⚠ DCT is supported only by the mlx5 driver.

⚠
- NVIDIA® ConnectX®-4 devices support DCT v0.
- NVIDIA® ConnectX®-5 and later devices support DCT v1.

DCT v0 and DCT v1 are not interoperable. Ensure all participating nodes are using the same DCT version to maintain compatibility.

5.6.2.1.5.4 MPI Tag Matching and Rendezvous Offloads

⚠ Supported in NVIDIA® ConnectX®-5 and above adapter cards.

MPI Tag Matching and Rendezvous Offloads is a technology developed by NVIDIA to offload the processing of MPI (Message Passing Interface) communications from the host CPU to the network adapter. This offload mechanism is designed to optimize message matching and reduce memory overhead in high-performance computing environments.

By leveraging this feature:

- MPI messages are delivered directly to the user's buffer with zero-copy semantics, eliminating the need for intermediate memory copies.
- Rendezvous protocol progress is handled entirely by the network adapter, without host CPU intervention.
- The CPU is freed to continue executing application-level computation while the adapter concurrently manages data transfers.

This offload model significantly improves performance and scalability for MPI-based applications by overlapping computation with communication and reducing CPU load.

For more information, refer to the community post Understanding MPI Tag Matching and Rendezvous Offloads (ConnectX-5).

5.6.2.1.6 Optimized Memory Access

5.6.2.1.6.1 Memory Region Re-registration

Memory region re-registration allows the user to change attributes of the memory region. The user may change the PD, access flags or the address and length of the memory region. Memory region supports contagious pages allocation. Consequently, it de-registers memory region followed by register memory region. Where possible, resources are reused instead of de-allocated and reallocated.

Example:

```
int ibv_rereg_mr(struct ibv_mr *mr, int flags, struct ibv_pd *pd, void *addr, size_t length, uint64_t access,
struct ibv_rereg_mr_attr *attr);
```

@mr:	The memory region to modify.
@flags:	A bit-mask used to indicate which of the following properties of the memory region are being modified. Flags should be one of: **IBV_REREG_MR_CHANGE_TRANSLATION** /* Change translation (location and length) */ **IBV_REREG_MR_CHANGE_PD** /* Change protection domain*/ **IBV_REREG_MR_CHANGE_ACCESS** /* Change access flags*/
@pd:	If **IBV_REREG_MR_CHANGE_PD** is set in flags, this field specifies the new protection domain to associated with the memory region, otherwise, this parameter is ignored.
@addr:	If **IBV_REREG_MR_CHANGE_TRANSLATION** is set in flags, this field specifies the start of the virtual address to use in the new translation, otherwise, this parameter is ignored.
@length:	If **IBV_REREG_MR_CHANGE_TRANSLATION** is set in flags, this field specifies the length of the virtual address to use in the new translation, otherwise, this parameter is ignored.
@access:	If **IBV_REREG_MR_CHANGE_ACCESS** is set in flags, this field specifies the new memory access rights, otherwise, this parameter is ignored. Could be one of the following: **IBV_ACCESS_LOCAL_WRITE** **IBV_ACCESS_REMOTE_WRITE** **IBV_ACCESS_REMOTE_READ** **IBV_ACCESS_ALLOCATE_MR** /* Let the library allocate the memory for the user, tries to get contiguous pages */
@attr:	Future extensions

ibv_rereg_mr returns 0 on success, or the value of an errno on failure (which indicates the error reason). In case of an error, the MR is in undefined state. The user needs to call **ibv_dereg_mr** in order to release it.

Please note that if the MR (Memory Region) is created as a Shared MR and a translation is requested, after the call, the MR is no longer a shared MR. Moreover, Re-registration of MRs that uses NVIDIA PeerDirect™ technology are not supported.

5.6.2.1.6.2 Memory Window

Memory Window allows the application to have a more flexible control over remote access to its memory. It is available only on physical functions/native machines The two types of Memory Windows supported are: type 1 and type 2B.
Memory Windows are intended for situations where the application wants to:

- Grant and revoke remote access rights to a registered region in a dynamic fashion with less of a performance penalty
- Grant different remote access rights to different remote agents and/or grant those rights over different ranges within registered region

For further information, please refer to the InfiniBand specification document.

Memory Windows API cannot co-work with peer memory clients (PeerDirect).

Query Capabilities

Memory Windows are available if and only the hardware supports it. To verify whether Memory Windows are available, run **ibv_query_device**.

For example:

```
struct ibv_device_attr device_attr = {.comp_mask = IBV_DEVICE_ATTR_RESERVED - 1};
ibv_query_device(context, & device_attr);
if (device_attr.exp_device_cap_flags & IBV_DEVICE_MEM_WINDOW ||
                device_attr.exp_device_cap_flags & IBV_DEVICE_MW_TYPE_2B) {
/* Memory window is supported */
```

Memory Window Allocation

Allocating memory window is done by calling the **ibv_alloc_mw** verb.

```
type_mw = IBV_MW_TYPE_2/ IBV_MW_TYPE_1
mw = ibv_alloc_mw(pd, type_mw);
```

Binding Memory Windows

After being allocated, memory window should be bound to a registered memory region. Memory Region should have been registered using the IBV_ACCESS_MW_BIND access flag.

For further information on how to bind memory windows, please see rdma-core man page.

Invalidating Memory Window

Before rebinding Memory Window type 2, it must be invalidated using **ibv_post_send**; see here.

Deallocating Memory Window

Deallocating memory window is done using the ibv_dealloc_mw verb.

```
ibv_dealloc_mw(mw);
```

5.6.2.1.6.3 User-Mode Memory Registration (UMR)

User-mode Memory Registration (UMR) is a fast registration mode which uses send queue. The UMR support enables the usage of RDMA operations and scatters the data at the remote side through the definition of appropriate memory keys on the remote side.
UMR enables the user to:

- Create indirect memory keys from previously registered memory regions, including creation of KLM's from previous KLM's. There are not data alignment or length restrictions associated with the memory regions used to define the new KLM's.
- Create memory regions, which support the definition of regular non-contiguous memory regions.

5.6.2.1.6.4 On-Demand-Paging (ODP)

On-Demand-Paging (ODP) is a technique to alleviate much of the shortcomings of memory registration. Applications no longer need to pin down the underlying physical pages of the address space, and track the validity of the mappings. Rather, the HCA requests the latest translations from the OS when pages are not present, and the OS invalidates translations which are no longer valid due to either non-present pages or mapping changes. ODP does not support contiguous pages.
ODP can be further divided into 2 subclasses: Explicit and Implicit ODP.

- Explicit ODP
  In Explicit ODP, applications still register memory buffers for communication, but this operation is used to define access control for IO rather than pin-down the pages. ODP Memory Region (MR) does not need to have valid mappings at registration time.
- Implicit ODP
  In Implicit ODP, applications are provided with a special memory key that represents their complete address space. This all IO accesses referencing this key (subject to the access rights associated with the key) does not need to register any virtual address range.

Query Capabilities

On-Demand Paging is available if both the hardware and the kernel support it. To verify whether ODP is supported, run **ibv_query_device**.

For further information, please refer to the **ibv_query_device** manual page.

Registering ODP Explicit and Implicit MR

ODP Explicit MR is registered after allocating the necessary resources (e.g. PD, buffer), while ODP implicit MR registration provides an implicit lkey that represents the complete address space.

For further information, please refer to the **ibv_reg_mr** manual page.

De-registering ODP MR

ODP MR is deregistered the same way a regular MR is deregistered:

```
ibv_dereg_mr(mr);
```

Advice MR Verb

The driver can pre-fetch a given range of pages and map them for access from the HCA. The advice MR verb is applicable for ODP MRs only.
For further information, please refer to the **ibv_advise_mr** manual page.

ODP Statistics

To aid in debugging and performance measurements and tuning, ODP support includes an extensive set of statistics.

For further information, please refer to rdma-statistics manual page.

5.6.2.1.6.5 Inline-Receive

The HCA may write received data to the Receive CQE. Inline-Receive saves PCIe Read transaction since the HCA does not need to read the scatter list. Therefore, it improves performance in case of short receive-messages.

On poll CQ, the driver copies the received data from CQE to the user's buffers.

Inline-Receive is enabled by default and is transparent to the user application. To disable it globally, set **MLX5_SCATTER_TO_CQE** environment variable to the value of 0. Otherwise, disable it on a specific QP using **mlx5dv_create_qp()** with **MLX5DV_QP_CREATE_DISABLE_SCATTER_TO_CQE**.

For further information, please refer to the manual page of **mlx5dv_create_qp()**.

5.6.2.1.7 NVIDIA PeerDirect

NVIDIA PeerDirect™ enables direct memory access between InfiniBand (IB) Core and peer memory clients (e.g., GPU cards) through a dedicated API. This allows RDMA-based applications (over InfiniBand or RoCE) to leverage the computing power of peer devices and the RDMA interconnect simultaneously—without copying data between peer-to-peer (P2P) devices.

A common use case is GPUDirect RDMA, which utilizes PeerDirect for efficient GPU communication.

For a detailed description of the API, refer to **docs/readme_and_user_manual/PEER_MEMORY_API.txt** in the MLNX_OFED installation package.

5.6.2.1.7.1 PeerDirect Async

The PeerDirect Async subsystem allows hardware peer devices (such as GPUs or dedicated accelerator cards) to control the HCA (Host Channel Adapter) directly for critical-path operations, reducing CPU involvement. This is achieved through a set of verb calls and data structures that provide the application with an abstract representation of operation sequences meant to be executed by the peer device.

5.6.2.1.7.2 Relaxed Ordering (RSYNC)

This feature is only supported on ConnectX-5 adapter cards and above.

In GPU systems that utilize relaxed memory ordering, an RSYNC callback is used to ensure memory consistency. This callback is registered and implemented via an external module provided by the system vendor. Once the module is loaded, it registers the callback with MLNX_OFED, which will then use it to maintain proper memory operation ordering.

5.6.2.1.8 CPU Overhead Distribution

When creating a Completion Queue (CQ) using the **ibv_create_cq()** API, the **comp_vector** argument determines the assignment of the CQ to an Event Queue (EQ).

If **comp_vector** is set to **0** and the CPU core executing the verb is not core 0, the driver automatically assigns the CQ to the completion EQ with the fewest CQs currently reporting to it. This strategy helps balance CPU overhead by distributing CQs across available completion EQs.

To explicitly assign a CQ to a specific EQ, set the desired EQ index in the **comp_vector** argument.

5.6.2.1.9 Out-of-order Data Placement

This feature is only supported on:

- NVIDIA® ConnectX®-5 adapters and above
- RC and XRC QPs
- DC transport

In certain fabric configurations, InfiniBand packets belonging to the same QP may traverse different network paths from source to destination. This can lead to packets arriving out of order.

Instead of dropping these packets and triggering retransmissions, this feature enables them to be accepted and placed directly into host memory, even if they arrive out of order. This improves overall performance by:

- Enhancing network utilization
- Reducing latency

When out-of-order messages are received, data is placed in host memory in the order it arrives, without requiring strict in-order delivery.

For setup instructions, refer to the community post HowTo Configure Adaptive Routing and SHIELD.

5.6.2.1.10 IB Router

An InfiniBand (IB) router enables communication between two or more IB subnets. This allows the network to scale beyond 40,000 end-ports, supports separation and fault resilience across subnet "islands," and facilitates connectivity between subnets with different topologies.

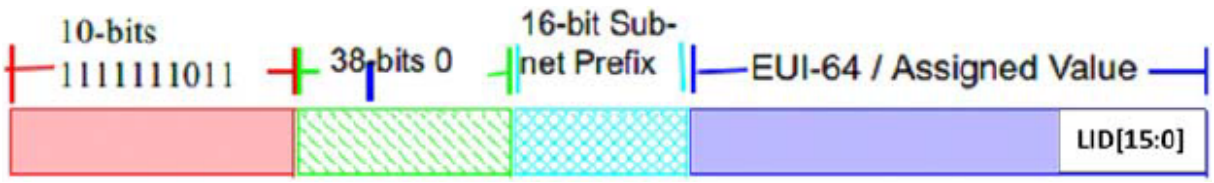
Forwarding between subnets is performed using Global Routing Header (GRH) lookup. The router's basic functionality includes:

- Removing the existing Layer 2 Local Routing Header (LRH)
- Performing routing
- Looking up the routing table using the GID extracted from the GRH

- Constructing a new LRH for the destination, based on the routing table

The DLID in the new LRH is derived using a simplified GID-to-LID mapping, where LID = the lower 16 bits of the GID. This approach eliminates the need for ARP queries or lookups.

Local Unicast GID Format

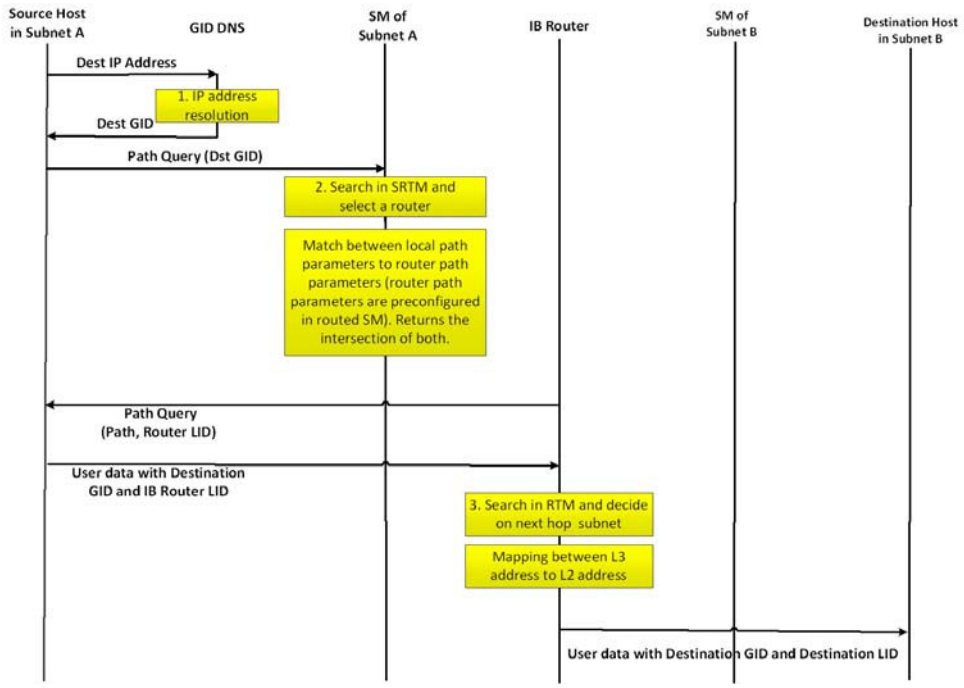


To enable this mechanism, the Subnet Manager (SM) assigns an alias GID to each host. The alias GID format is:

```
alias GID = {subnet_prefix[127:64], reserved[63:16], LID[15:0]}
```

Hosts must use these alias GIDs when transmitting traffic to peers located in remote subnets.

Host-to-Host IB Router Unicast Flow



For information on the architecture and functionality of IB Router, refer to IB Router Architecture and Functionality Community post.

For information on IB Router configuration, refer to HowTo Configure IB Routers Community post.

5.6.2.1.11 MAD Congestion Control

Subnet Administration (SA) Management Datagrams (MADs) are General Management Packets (GMPs) used to communicate with the SA entity within an InfiniBand subnet. The SA is typically part of the subnet manager and exists as a single active instance. As a result, congestion at the SA communication level can occur.

Congestion control is handled by limiting the number of outstanding MADs, which are MADs that have been sent but have not yet received a response. A FIFO queue holds additional SA MADs whose transmission is delayed due to exceeding the **max_outstanding** threshold.

The length of this queue is controlled by **queue_size**, which helps prevent excessive memory use. If the FIFO queue becomes full, new SA MADs are dropped, and the drops counter is incremented.

Each queued MAD also has an associated timeout (**time_sa_mad**). When this timeout expires, the MAD is removed from the queue, and the user is notified of the expiration.

This feature is implemented per CA port, and configuration is available via sysfs:

```
/sys/class/infiniband/mlx5_0/mad_sa_cc/
    1/
        drops
        max_outstanding
        queue_size
        time_sa_mad
    2/
        drops
        max_outstanding
        queue_size
        time_sa_mad
```

5.6.2.1.11.1 Usage Examples

- To print the current value:

```
cat /sys/class/infiniband/mlx5_0/mad_sa_cc/1/max_outstanding
# Output: 16
```

- To change the current value:

```
echo 32 > /sys/class/infiniband/mlx5_0/mad_sa_cc/1/max_outstanding
cat /sys/class/infiniband/mlx5_0/mad_sa_cc/1/max_outstanding
# Output: 32
```

- To reset the drops counter:

```
echo 0 > /sys/class/infiniband/mlx5_0/mad_sa_cc/1/drops
```

Valid parameter ranges:

Parameter	Range	Default Values
max_oustanding	1–2^20	16
queue_size	16–2^20	16

Parameter	Range	Default Values
time_sa_mad	1 ms–10000	20 ms

5.6.2.2 Ethernet Network

The chapter contains the following sections:

- Ethernet Interface
- Ethernet QoS
- Ethtool
- Checksum Offload
- Ignore FCS Errors
- Flow Control
- Explicit Congestion Notification
- RSS Support
- Time Stamping
- Flow Steering
- Wake-on-LAN
- Q-in-Q Tunneling
- VLAN Stripping in Linux Verbs
- Offloaded Traffic Sniffer
- Dump Configuration
- Local Loopback Disable
- kTLS Offloads
- IPsec Crypto Offload
- IPsec Packet Offload
- MACsec Full Offload

5.6.2.2.1 Ethernet Interface

5.6.2.2.1.1 Port Type Management Configuration

Ports of ConnectX-4 adapter cards and above can be individually configured to work as InfiniBand or Ethernet ports. If you wish to change the port type, use the mlxconfig script after the driver is loaded.

For further information on how to set the port type, please refer to the MFT User Manual.

5.6.2.2.1.2 Counters

Counters are used to provide information about how well an operating system, an application, a service, or a driver is performing. The counter data help determine system bottlenecks and fine-tune the system and application performance. The operating system, network, and devices provide counter data that an application can consume to provide users with a graphical view of how well the system is performing.

The counter index is a Queue Pair (QP) attribute given in the QP context. Multiple QPs may be associated with the same counter set. If multiple QPs share the same counter, the counter value will represent the cumulative total.

RoCE Counters

- RoCE counters are available only through sysfs located under:
  - `# /sys/class/infiniband/<device>/ports/*/hw_counters/`
  - `# /sys/class/infiniband/<device>/hw_counters/`
  - `# /sys/class/infiniband/<device>/ports/*/counters/`

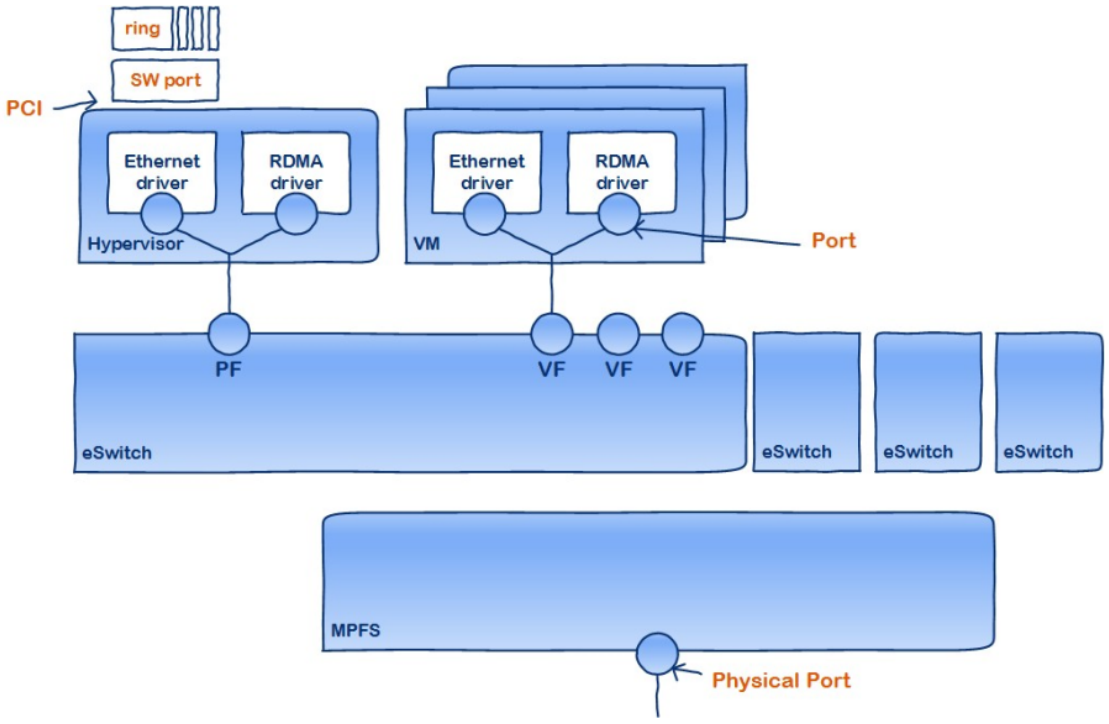
For mlx5 port and RoCE counters, refer to the Understanding mlx5 Linux Counters community post.

SR-IOV Counters

PFs can also read VFs' port counters through sysfs located under `/sys/class/net/<interface_name>/device/sriov/<index>/stats/` .

ethtool Counters

The ethtool counters are counted in different places, according to which they are divided into groups. Each counters group may also have different counter types.



For the full list of supported ethtool counters, refer to the Understanding mlx5 ethtool Counters community post.

5.6.2.2.1.3 Persistent Naming

To avoid network interface renaming after boot or driver restart, set the desired constant interface name in the **/etc/udev/rules.d/70-persistent-net.rules** file.

- Example for Ethernet interfaces:

```
PCI device 15b3:1019 (mlx5_core)
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{address}=="00:02:c9:fa:c3:50", ATTR{dev_id}=="0x0",
ATTR{type}=="1", KERNEL=="eth*", NAME="eth1"
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{address}=="00:02:c9:fa:c3:51", ATTR{dev_id}=="0x0",
ATTR{type}=="1", KERNEL=="eth*", NAME="eth2"
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{address}=="00:02:c9:e9:56:a1", ATTR{dev_id}=="0x0",
ATTR{type}=="1", KERNEL=="eth*", NAME="eth3"
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{address}=="00:02:c9:e9:56:a2", ATTR{dev_id}=="0x0",
ATTR{type}=="1", KERNEL=="eth*", NAME="eth4"
```

- Example for IPoIB interfaces:

```
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{dev_id}=="0x0", ATTR{type}=="32", NAME="ib0"
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{dev_id}=="0x1", ATTR{type}=="32", NAME="ib1"
```

5.6.2.2.1.4 Interrupt Request (IRQ) Naming

Once IRQs are allocated by the driver, they are named **mlx5_comp<x>@pci:<pci_addr>**. The IRQ name is constant and is not affected by the interface state.

The mlx5_core driver allocates all IRQs during loading time to support the maximum possible number of channels. Once the driver is up, no further IRQs are freed or allocated. Changing the number of working channels does not re-allocate or free the IRQs.

5.6.2.2.2 Ethernet QoS

Quality of service (QoS) is a mechanism of assigning a priority to a network flow (socket, **rdma_cm** connection) and manage its guarantees, limitations and its priority over other flows. This is accomplished by mapping the user's priority to a hardware TC (traffic class) through a 2/3 stage process. The TC is assigned with the QoS attributes and the different flows behave accordingly.

5.6.2.2.2.1 Mapping Traffic to Traffic Classes

Mapping traffic to TCs consists of several actions which are user controllable, some controlled by the application itself and others by the system/network administrators.
The following is the general mapping traffic to Traffic Classes flow:

1. The application sets the required type of service (ToS).
2. The ToS is translated into a Socket Priority (**sk_prio**).
3. The **sk_prio** is mapped to a user priority (UP) by the system administrator (some applications set **sk_prio** directly).
4. The UP is mapped to TC by the network/system administrator.
5. TCs hold the actual QoS parameters

QoS can be applied on the following types of traffic. However, the general QoS flow may vary among them:

- Plain Ethernet – Applications use regular inet sockets and the traffic passes via the kernel Ethernet driver
- RoCE – Applications use the RDMA API to transmit using Queue Pairs (QPs)
- Raw Ethernet QP – Application use VERBs API to transmit using a Raw Ethernet QP

5.6.2.2.2.2 Plain Ethernet Quality of Service Mapping

Applications use regular inet sockets and the traffic passes via the kernel Ethernet driver. The following is the Plain Ethernet QoS mapping flow:

1. The application sets the ToS of the socket using setsockopt (IP_TOS, value).
2. ToS is translated into the sk_prio using a fixed translation:

```
TOS 0 <=> sk_prio 0
TOS 8 <=> sk_prio 2
TOS 24 <=> sk_prio 4
TOS 16 <=> sk_prio 6
```

3. The Socket Priority is mapped to the UP in the following conditions:
    a. If the underlying device is a VLAN device, egress_map is used controlled by the vconfig command. This is per VLAN mapping.
    b. If the underlying device is not a VLAN device, the mapping is done in the driver.
4. The UP is mapped to the TC as configured by the mlnx_qos tool or by the lldpad daemon if DCBX is used.

⚠ Socket applications can use setsockopt (SK_PRIO, value) to directly set the **sk_prio** of the socket. In this case, the ToS to **sk_prio** fixed mapping is not needed. This allows the application and the administrator to utilize more than the 4 values possible via ToS.

⚠ In the case of a VLAN interface, the UP obtained according to the above mapping is also used in the VLAN tag of the traffic.

5.6.2.2.2.3 RoCE Quality of Service Mapping

Applications use RDMA-CM API to create and use QPs. The following is the RoCE QoS mapping flow:

1. The application sets the ToS of the QP using the rdma_set_option option(RDMA_OPTION_ID_TOS, value).
2. ToS is translated into the Socket Priority (sk_prio) using a fixed translation:

```
TOS 0 <=> sk_prio 0
TOS 8 <=> sk_prio 2
TOS 24 <=> sk_prio 4
TOS 16 <=> sk_prio 6
```

3. The Socket Priority is mapped to the User Priority (UP) using the tc command.
    - In the case of a VLAN device where the parent real device is used for the purpose of this mapping

- If the underlying device is a VLAN device, and the parent real device was not used for the mapping, the VLAN device's egress_map is used

4. UP is mapped to the TC as configured by the mlnx_qos tool or by the lldpad daemon if DCBX is used.

⚠ With RoCE, there can only be 4 predefined ToS values for the purpose of QoS mapping.

5.6.2.2.2.4 Map Priorities with set_egress_map

For RoCE old kernels that do not support set_egress_map, use the tc_wrap script to map between sk_prio and UP. Use tc_wrap with option -u. For example:

```
tc_wrap -i <ethX> -u <skprio2up mapping>
```

5.6.2.2.2.5 Quality of Service Properties

The different QoS properties that can be assigned to a TC are:

- Strict Priority
- Enhanced Transmission Selection (ETS)
- Rate Limit
- Trust State
- Receive Buffer
- DCBX Control Mode

Strict Priority

When setting a TC's transmission algorithm to be 'strict', then this TC has absolute (strict) priority over other TC strict priorities coming before it (as determined by the TC number: TC 7 is the highest priority, TC 0 is lowest). It also has an absolute priority over nonstrict TCs (ETS).
This property needs to be used with care, as it may easily cause starvation of other TCs.
A higher strict priority TC is always given the first chance to transmit. Only if the highest strict priority TC has nothing more to transmit, will the next highest TC be considered.
Nonstrict priority TCs will be considered last to transmit.
This property is extremely useful for low latency low bandwidth traffic that needs to get immediate service when it exists, but is not of high volume to starve other transmitters in the system.

Enhanced Transmission Selection (ETS)

Enhanced Transmission Selection standard (ETS) exploits the time periods in which the offered load of a particular Traffic Class (TC) is less than its minimum allocated bandwidth by allowing the difference to be available to other traffic classes.
After servicing the strict priority TCs, the amount of bandwidth (BW) left on the wire may be split among other TCs according to a minimal guarantee policy.

If, for instance, TC0 is set to 80% guarantee and TC1 to 20% (the TCs sum must be 100), then the BW left after servicing all strict priority TCs will be split according to this ratio.
Since this is a minimum guarantee, there is no maximum enforcement. This means, in the same example, that if TC1 did not use its share of 20%, the reminder will be used by TC0.
ETS is configured using the mlnx_qos tool (mlnx_qos) which allows you to:

- Assign a transmission algorithm to each TC (strict or ETS)
- Set minimal BW guarantee to ETS TCs
  Usage:

```
mlnx_qos -i \[options\]
```

Rate Limit

Rate limit defines a maximum bandwidth allowed for a TC. Please note that 10% deviation from the requested values is considered acceptable.

Trust State

Trust state enables prioritizing sent/received packets based on packet fields.
The default trust state is PCP. Ethernet packets are prioritized based on the value of the field (PCP/DSCP).
For further information on how to configure Trust mode, please refer to HowTo Configure Trust State on NVIDIA Adapters community post.

Setting the Trust State mode shall be done before enabling SR-IOV in order to propagate the Trust State to the VFs.

Receive Buffer

By default, the receive buffer configuration is controlled automatically. Users can override the receive buffer size and receive buffer's xon and xoff thresholds using mlnx_qos tool.
For further information, please refer to HowTo Tune the Receive buffers on NVIDIA Adapters community post.

DCBX Control Mode

DCBX settings, such as "ETS" and "strict priority" can be controlled by firmware or software. When DCBX is controlled by firmware, changes of QoS settings cannot be done by the software. The DCBX control mode is configured using the mlnx_qos -d os/fw command.
For further information on how to configure the DCBX control mode, please refer to mlnx_qos community post.

5.6.2.2.2.6 Quality of Service Tools

mlnx_qos

mlnx_qos is a centralized tool used to configure QoS features of the local host. It communicates directly with the driver thus does not require setting up a DCBX daemon on the system.
The mlnx_qos tool enables the administrator of the system to:

- Inspect the current QoS mappings and configuration
  The tool will also display maps configured by TC and vconfig set_egress_map tools, in order to give a centralized view of all QoS mappings.
- Set UP to TC mapping
- Assign a transmission algorithm to each TC (strict or ETS)
- Set minimal BW guarantee to ETS TCs
- Set rate limit to TCs
- Set DCBX control mode
- Set cable length
- Set trust state

For an unlimited ratelimit, set the ratelimit to 0.

Usage:

```
mlnx_qos -i <interface> \[options\]
```

Options:

--version	Show the program's version number and exit
-h, --help	Show this help message and exit
-f LIST, --pfc=LIST	Set priority flow control for each priority. LIST is a comma separated value for each priority starting from 0 to 7. Example: 0,0,0,0,1,1,1,1 enable PFC on TC4-7
-p LIST, --prio_tc=LIST	Maps UPs to TCs. LIST is 8 comma-separated TC numbers. Example: 0,0,0,0,1,1,1,1 maps UPs 0-3 to TC0, and UPs 4-7 to TC1
-s LIST, --tsa=LIST	Transmission algorithm for each TC. LIST is comma separated algorithm names for each TC. Possible algorithms: strict, ets and vendor. Example: vendor,strict,ets,ets,ets,ets,ets,ets sets TC0 to vendor, TC1 to strict, TC2-7 to ets
-t LIST, --tcbw=LIST	Set the minimally guaranteed %BW for ETS TCs. LIST is comma-separated percents for each TC. Values set to TCs that are not configured to ETS algorithm are ignored but must be present. Example: if TC0,TC2 are set to ETS, then 10,0,90,0,0,0,0,0will set TC0 to 10% and TC2 to 90%. Percents must sum to 100
-r LIST, --ratelimit=LIST	Rate limit for TCs (in Gbps). LIST is a comma-separated Gbps limit for each TC. Example: 1,8,8 will limit TC0 to 1Gbps, and TC1,TC2 to 8 Gbps each
-d DCBX, --dcbx=DCBX	Set dcbx mode to firmware controlled(fw) or OS controlled(os). Note, when in OS mode, mlnx_qos should not be used in parallel with other dcbx tools, such as lldptool
--trust=TRUST	set priority trust state to pcp or dscp

--dscp2prio=DSCP2PRIO	Set/del a (dscp,prio) mapping. Example 'set,30,2' maps dscp 30 to priority 2. 'del,30,2' resets the dscp 30 mapping back to the default setting priority 0
--cable_len=CABLE_LEN	Set cable_len for buffer's xoff and xon thresholds
-i INTF, --interface=INTF	Interface name
-a	Show all interface's TCs

Get current configuration:

```
ofed_scripts/utils/mlnx_qos -i ens1f0
DCBX mode: OS controlled
Priority trust state: dscp
dscp2prio mapping:
        prio:0 dscp:07,06,05,04,03,02,01,00,
        prio:1 dscp:15,14,13,12,11,10,09,08,
        prio:2 dscp:23,22,21,20,19,18,17,16,
        prio:3 dscp:31,30,29,28,27,26,25,24,
        prio:4 dscp:39,38,37,36,35,34,33,32,
        prio:5 dscp:47,46,45,44,43,42,41,40,
        prio:6 dscp:55,54,53,52,51,50,49,48,
        prio:7 dscp:63,62,61,60,59,58,57,56,
Cable len: 7
PFC configuration:
        priority 0 1 2 3 4 5 6 7
        enabled 0 0 0 0 0 0 0 0
tc: 0 ratelimit: unlimited, tsa: vendor
         priority: 1
tc: 1 ratelimit: unlimited, tsa: vendor
         priority: 0
tc: 2 ratelimit: unlimited, tsa: vendor
         priority: 2
tc: 3 ratelimit: unlimited, tsa: vendor
         priority: 3
tc: 4 ratelimit: unlimited, tsa: vendor
         priority: 4
tc: 5 ratelimit: unlimited, tsa: vendor
         priority: 5
tc: 6 ratelimit: unlimited, tsa: vendor
         priority: 6
tc: 7 ratelimit: unlimited, tsa: vendor
         priority: 7
```

Set rate limit (3Gb/s for tc0 4Gb/s for tc1 and 2Gb/s for tc2):

```
# mlnx_qos -i <interface> -p 0,1,2 -r 3,4,2
tc: 0 ratelimit: 3 Gbps, tsa: strict
         up:  0
                 skprio: 0
                 skprio: 1
                 skprio: 2 (tos: 8)
                 skprio: 3
                 skprio: 4 (tos: 24)
                 skprio: 5
                 skprio: 6 (tos: 16)
                 skprio: 7
                 skprio: 8
                 skprio: 9
                 skprio: 10
                 skprio: 11
                 skprio: 12
                 skprio: 13
                 skprio: 14
                 skprio: 15
         up:  3
         up:  4
         up:  5
         up:  6
         up:  7
tc: 1 ratelimit: 4 Gbps, tsa: strict
         up:  1
tc: 2 ratelimit: 2 Gbps, tsa: strict
         up:  2
```

Configure QoS
(map UP 0,7 to tc0,1,2,3 to tc1 and 4,5,6 to tc2. Set tc0,tc1 as ets and tc2 as strict. Divide ets 30 % for tc0 and 70% for tc1):

```
# mlnx_qos -i <interface> -s ets,ets,strict -p 0,1,1,1,2,2,2 -t 30,70
tc: 0 ratelimit: 3 Gbps, tsa: ets, bw: 30%
         up:  0
                 skprio: 0
                 skprio: 1
                 skprio: 2 (tos: 8)
                 skprio: 3
                 skprio: 4 (tos: 24)
                 skprio: 5
                 skprio: 6 (tos: 16)
                 skprio: 7
                 skprio: 8
                 skprio: 9
                 skprio: 10
                 skprio: 11
                 skprio: 12
                 skprio: 13
                 skprio: 14
                 skprio: 15
  up:  7
tc: 1 ratelimit: 4 Gbps, tsa: ets, bw: 70%
         up:  1
         up:  2
         up:  3
tc: 2 ratelimit: 2 Gbps, tsa: strict
         up:  4
         up:  5
         up:  6
```

tc and tc_wrap.py

The tc tool is used to create 8 Traffic Classes (TCs).
The tool will either use the sysfs (/sys/class/net/<ethX>/qos/tc_num) or the tc tool to create the TCs.

Usage

```
tc_wrap.py -i <interface> \[options\]
```

Options

--version	show program's version number and exit
-h, --help	show this help message and exit
-u SKPRIO_UP, --skprio_up=SKPRIO_UP	maps sk_prio to priority for RoCE. LIST is <=16 comma separated priority. index of element is sk_prio
-i INTF, --interface=INTF	Interface name

Example

Run:

```
tc_wrap.py -i enp139s0
```

Output:

```
Tarrfic classes are set to 8

UP  0
    skprio: 0 (vlan 5)
UP  1
    skprio: 1 (vlan 5)
UP  2
    skprio: 2 (vlan 5 tos: 8)
```

```
UP  3
    skprio: 3 (vlan 5)
UP  4
    skprio: 4 (vlan 5 tos: 24)
UP  5
    skprio: 5 (vlan 5)
UP  6
    skprio: 6 (vlan 5 tos: 16)
UP  7
    skprio: 7 (vlan 5)
```

Additional Tools

tc tool compiled with the sch_mqprio module is required to support kernel v2.6.32 or higher. This is a part of iproute2 package v2.6.32-19 or higher. Otherwise, an alternative custom sysfs interface is available.

- mlnx_qos tool (package: ofed-scripts) requires python version 2.5 < = X
- tc_wrap.py (package: ofed-scripts) requires python version 2.5 < = X

5.6.2.2.2.7 Packet Pacing

ConnectX-4 and above devices allow packet pacing (traffic shaping) per flow. This capability is achieved by mapping a flow to a dedicated send queue and setting a rate limit on that Send queue.
Note the following:

- Up to 512 send queues are supported
- 16 different rates are supported
- The rates can vary from 1 Mbps to line rate in 1 Mbps resolution
- Multiple queues can be mapped to the same rate (each queue is paced independently)
- It is possible to configure rate limit per CPU and per flow in parallel

System Requirements

- Driver v3.3 or higher
- Linux kernel v4.1 or higher
- ConnectX-4 or ConnectX-4 Lx adapter cards with an official firmware version

Packet Pacing Configuration

⚠ This configuration is non-persistent and does not survive driver restart.

1. Firmware Activation:

   To activate Packet Pacing in the firmware:

   First, make sure MFT service (mst) is started:

   ```
   # mst start
   ```

   Then run:

   ```
   #echo "MLNX_RAW_TLV_FILE" > /tmp/mlxconfig_raw.txt
   #echo "0x00000004 0x0000010c 0x00000000 0x00000001" >> /tmp/mlxconfig_raw.txt
   #yes | mlxconfig -d <mst_dev> -f /tmp/mlxconfig_raw.txt set_raw > /dev/null
   #reboot /mlxfwreset
   ```

   To deactivate Packet Pacing in the firmware, *run:*

```
#echo "MLNX_RAW_TLV_FILE" > /tmp/mlxconfig_raw.txt
#echo "0x00000004 0x0000010c 0x00000000 0x00000000" >> /tmp/mlxconfig_raw.txt
#yes | mlxconfig -d <mst_dev >-f /tmp/mlxconfig_raw.txt set_raw > /dev/null
#reboot /mlxfwreset
```

2. Driver Activation:
   There are two operation modes for Packet Pacing:

   a. Rate limit per CPU core:
      When XPS is enabled, traffic from a CPU core will be sent using the corresponding send queue. By limiting the rate on that queue, the transmit rate on that CPU core will be limited. For example:

      ```
      echo 300 > /sys/class/net/ens2f1/queues/tx-0/tx_maxrate
      ```

      In this case, the rate on Core 0 (tx-0) is limited to 300Mbit/sec.

   b. Rate limit per flow:

      i. The driver allows opening up to 512 additional send queues using the following command:

      ```
      ethtool -L ens2f1 other 1200
      ```

      In this case, 1200 additional queues are opened

      ii. Create flow mapping.
      Users can map a certain destination IP and/or destination layer 4 Port to a specific send queue. The match precedence is as follows:

      - IP + L4 Port
      - IP only
      - L4 Port only
      - No match (the flow would be mapped to default queues)

      To create flow mapping:
      Configure the destination IP. Write the IP address in hexadecimal representation to the relevant sysfs entry. For example, to map IP address 192.168.1.1 (0xc0a80101) to send queue 310, run the following command:

      ```
      echo 0xc0a80101 > /sys/class/net/ens2f1/queues/tx-310/flow_map/dst_ip
      ```

      To map Destination L4 3333 port (either TCP or UDP) to the same queue, run:

      ```
      echo 3333 > /sys/class/net/ens2f1/queues/tx-310/flow_map/dst_port
      ```

      From this point on, all traffic destined to the given IP address and L4 port will be sent using send queue 310. All other traffic will be sent using the original send queue.

      iii. Limit the rate of this flow using the following command:

```
echo 100 > /sys/class/net/ens2f1/queues/tx-310/tx_maxrate
```

⚠ Each queue supports only a single IP+Port combination.

5.6.2.2.3 Ethtool

Ethtool is a standard Linux utility for controlling network drivers and hardware, particularly for wired Ethernet devices. It can be used to:

- Get identification and diagnostic information
- Get extended device statistics
- Control speed, duplex, auto-negotiation and flow control for Ethernet devices
- Control checksum offload and other hardware offload features
- Control DMA ring sizes and interrupt moderation
- Flash device firmware using a .mfa2 image

Ethtool Supported Options

Options	Description
ethtool --set-priv-flags eth<x> <priv flag> <on/off>	Enables/disables driver feature matching the given private flag.
ethtool --show-priv-flags eth<x>	Shows driver private flags and their states (ON/OFF).
ethtool -a eth<x>	Queries the pause frame settings.
ethtool -A eth<x> [rx on|off] [tx on|off]	Sets the pause frame settings.
ethtool -c eth<x>	Queries interrupt coalescing settings.
ethtool -C eth<x> [pkt-rate-low N] [pkt-rate-high N] [rx-usecs-low N] [rx-usecs-high N]	Sets the values for packet rate limits and for moderation time high and low values.
ethtool -C eth<x> [rx-usecs N] [rx-frames N]	Sets the interrupt coalescing setting. rx-frames will be enforced immediately, rx-usecs will be enforced only when adaptive moderation is disabled. **Note**: usec settings correspond to the time to wait after the *last* packet is sent/received before triggering an interrupt.
ethtool -C eth<x> adaptive-rx on|off	Enables/disables adaptive interrupt moderation. By default, the driver uses adaptive interrupt moderation for the receive path, which adjusts the moderation time to the traffic pattern.
ethtool -C eth<x> adaptive-tx on|off	**Note**: Supported by mlx5e for ConnectX-4 and above adapter cards. Enables/disables adaptive interrupt moderation. By default, the driver uses adaptive interrupt moderation for the transmit path, which adjusts the moderation parameters (time/frames) to the traffic pattern.
ethtool -g eth<x>	Queries the ring size values.
ethtool -G eth<x> [rx <N>] [tx <N>]	Modifies the ring size.

Options	Description
ethtool -i eth<x>	Checks driver and device information. For example: driver: mlx5_core version: 5.1-0.4.0 firmware-version: 4.6.4046 (MT_QEMU000000) expansion-rom-version: bus-info: 0000:07:00.0 supports-statistics: yes supports-test: yes supports-eeprom-access: no supports-register-dump: no supports-priv-flags: yes
ethtool -k eth<x>	Queries the stateless offload status.
ethtool -K eth<x> [rx on|off] [tx on|off] [sg on|off] [tso on|off] [lro on|off] [gro on|off] [gso on|off] [rxvlan on|off] [txvlan on|off] [ntuple on/off] [rxhash on/off] [rx-all on/off] [rx-fcs on/off]	Sets the stateless offload status. TCP Segmentation Offload (TSO), Generic Segmentation Offload (GSO): increase outbound throughput by reducing CPU overhead. It works by queuing up large buffers and letting the network interface card split them into separate packets. Large Receive Offload (LRO): increases inbound throughput of high-bandwidth network connections by reducing CPU overhead. It works by aggregating multiple incoming packets from a single stream into a larger buffer before they are passed higher up the networking stack, thus reducing the number of packets that have to be processed. LRO is available in kernel versions < 3.1 for untagged traffic. Hardware VLAN insertion Offload (txvlan): When enabled, the sent VLAN tag will be inserted into the packet by the hardware. **Note**: LRO will be done whenever possible. Otherwise GRO will be done. Generic Receive Offload (GRO) is available throughout all kernels. Hardware VLAN Striping Offload (rxvlan): When enabled received VLAN traffic will be stripped from the VLAN tag by the hardware. RX FCS (rx-fcs): Keeps FCS field in the received packets.Sets the stateless offload status. RX FCS validation (rx-all): Ignores FCS validation on the received packets.
ethtool -l eth<x>	Shows the number of channels.
ethtool -L eth<x> [rx <N>] [tx <N>]	Sets the number of channels. **Notes**: • This also resets the RSS table to its default distribution, which is uniform across the cores on the NUMA (non-uniform memory access) node that is closer to the NIC. • For ConnectX®-4 cards, use ethtool -L eth<x> combined <N> to set both RX and TX channels.
ethtool -m|--dump-module-eeprom eth<x> [raw on|off] [hex on|off] [offset N] [length N]	Queries/decodes the cable module eeprom information.
ethtool -p|--identify DEVNAME	Enables visual identification of the port by LED blinking [TIME-IN-SECONDS].

<table>
<tr><th>Options</th><th>Description</th></tr>
<tr><td>ethtool -p|--identify eth<x> <LED duration></td><td>Allows users to identify interface's physical port by turning the ports LED on for a number of seconds.
Note: The limit for the LED duration is 65535 seconds.</td></tr>
<tr><td>ethtool -S eth<x></td><td>Obtains additional device statistics.</td></tr>
<tr><td>ethtool -s eth<x> advertise <N> autoneg on</td><td>Changes the advertised link modes to requested link modes <N>
To check the link modes' hex values, run <man ethtool> and to check the supported link modes, run ethtool eth<x>
For advertising new link modes, make sure to configure the entire bitmap as follows:
<table>
<tr><td>200GAUI-4 / 200GBASE-CR4/KR4</td><td>0x7c000000000000000</td></tr>
<tr><td>100GAUI-2 / 100GBASE-CR2 / KR2</td><td>0x3E00000000000000</td></tr>
<tr><td>CAUI-4 / 100GBASE-CR4 / KR4</td><td>0xF000000000</td></tr>
<tr><td>50GAUI-1 / LAUI-1/ 50GBASE-CR / KR</td><td>0x1F0000000000000</td></tr>
<tr><td>50GAUI-2 / LAUI-2/ 50GBASE-CR2/KR2</td><td>0x10C00000000</td></tr>
<tr><td>XLAUI-4/XLPPI-4 // 40G</td><td>0x7800000</td></tr>
<tr><td>25GAUI-1/ 25GBASE-CR / KR</td><td>0x380000000</td></tr>
<tr><td>XFI / XAUI-1 // 10G</td><td>0x7C0000181000</td></tr>
<tr><td>5GBASE-R</td><td>0x1000000000000</td></tr>
<tr><td>2.5GBASE-X / 2.5GMII</td><td>0x820000000000</td></tr>
<tr><td>1000BASE-X / SGMII</td><td>0x20000020020</td></tr>
</table>
Notes:
Both previous and new link modes configurations are supported, however, they must be run separately.
Any link mode configuration on Kernels below v5.1 and ConnectX-6 HCAs will result in the advertisement of the full capabilities.
<autoneg on> only sends a hint to the driver that the user wants to modify advertised link modes and not speed.</td></tr>
<tr><td>ethtool -s eth<x> msglvl [N]</td><td>Changes the current driver message level.</td></tr>
<tr><td>ethtool -s eth<x> speed <SPEED> autoneg off</td><td>Changes the link speed to requested <SPEED>. To check the supported speeds, run ethtool eth<x> .
Note: <autoneg off> does not set autoneg OFF, it only hints the driver to set a specific speed.</td></tr>
<tr><td>ethtool -t eth<x></td><td>Performs a self-diagnostics test.</td></tr>
<tr><td>ethtool -T eth<x></td><td>Shows time stamping capabilities</td></tr>
<tr><td>ethtool -x eth<x></td><td>Retrieves the receive flow hash indirection table.</td></tr>
<tr><td>ethtool -X eth<x> equal a b c...</td><td>Sets the receive flow hash indirection table.
Note: The RSS table configuration is reset whenever the number of channels is modified (using ethtool -L command).</td></tr>
<tr><td>ethtool --show-fec eth<x></td><td>Queries current Forward Error Correction (FEC) encoding in case FEC is supported.
Note: An output of "baser" implies Firecode encoding.</td></tr>
</table>

Options	Description
ethtool --set-fec eth<x> encoding auto|off|rs|baser	Configures Forward Error Correction (FEC). **Note**: 'baser' encoding applies to the Firecode encoding, and 'auto' regards the HCA's default.
ethtool -f|--flash <devname> FILE [N]	Flash firmware image on the device using the specified .mfa2 file (FILE). By default, the command flashes all the regions on the device unless a region number (N) is specified.

5.6.2.2.4 Checksum Offload

The following "receive IP/L4 checksum offload" modes are supported.

- **CHECKSUM_UNNECESSARY** – When this mode is used, the driver indicates to the Linux networking stack that the hardware successfully validated the IP and L4 checksum so the Linux networking stack does not need to deal with IP/L4 checksum validation.
- **CHECKSUM_COMPLETE** – When this mode is used, the driver still reports to the OS the calculated by hardware checksum value. This allows accelerating checksum validation in Linux networking stack, since it does not have to calculate the whole checksum including payload by itself.
- **CHECKSUM_NONE** – When this mode is used, the driver indicates to the Linux networking stack that it must calculate and validate the IP/L4 checksum.

5.6.2.2.5 Ignore FCS Errors

Upon packet reception, the Frame Check Sequence (FCS) field undergoes checksum validation. If the validation fails, the packet is dropped.

When the FCS option is enabled (disabled by default), the device skips FCS validation, accepting packets regardless of whether the FCS is valid. This behavior can be useful in specific diagnostic or experimental scenarios, but enabling FCS is generally not recommended for production environments.

To enable or disable FCS validation, use the **ethtool** option:

```
ethtool --offload <interface> rx-fcs on|off
```

5.6.2.2.6 Flow Control

5.6.2.2.6.1 Priority Flow Control (PFC)

Priority Flow Control (PFC) IEEE 802.1Qbb applies pause functionality to specific classes of traffic on the Ethernet link. For example, PFC can provide lossless service for the RoCE traffic and best-effort service for the standard Ethernet traffic. PFC can provide different levels of service to specific classes of Ethernet traffic (using IEEE 802.1p traffic classes).

Configuring PFC on ConnectX-4 and above

1. Enable PFC on the desired priority:

```
mlnx_qos -i <ethX> --pfc <0/1>,<0/1>,<0/1>,<0/1>,<0/1>,<0/1>,<0/1>,<0/1>
```

Example (Priority=4):

```
mlnx_qos -i eth1 --pfc 0,0,0,0,1,0,0,0
```

2. Create a VLAN interface:

```
vconfig add <ethX> <VLAN_ID>
```

Example (VLAN_ID=5):

```
vconfig add eth1 5
```

3. Set egress mapping:
   a. For Ethernet traffic:

```
vconfig set_egress_map <vlan_einterface> <skprio> <up>
```

Example (skprio=3, up=5):

```
vconfig set_egress_map eth1.5 3 5
```

4. Create 8 Traffic Classes (TCs):

```
tc_wrap.py -i <interface>
```

5. Enable PFC on the switch.
   For information on how to enable PFC on your respective switch, please refer to Switch FC/PFC Configuration sections in the RDMA/RoCE Solutions Community page.

PFC Configuration Using LLDP DCBX

PFC Configuration on Hosts

PFC Auto-Configuration Using LLDP Tool in the OS

1. Start lldpad daemon on host.

```
lldpad -d
```

OR

```
service lldpad start
```

2. Send lldpad packets to the switch.

```
lldptool set-lldp -i <ethX>  adminStatus=rxtx
lldptool -T -i <ethX>   -V  sysName enableTx=yess
lldptool -T -i <ethX>   -V  portDesc enableTx=yess
lldptool -T -i <ethX>   -V  sysDesc enableTx=yess
lldptool -T -i <ethX>   -V sysCap enableTx=yess
lldptool -T -i <ethX>   -V mngAddr enableTx=yess
lldptool -T -i <ethX>   -V PFC enableTx=yes;
lldptool -T -I <ethX>   -V CEE-DCBX enableTx=yess
```

3. Set the PFC parameters.

- For the CEE protocol, use dcbtool:

```
dcbtool sc <ethX> pfc pfcup:<xxxxxxxx>
```

Example:

```
dcbtool sc eth6 pfc pfcup:01110001
```

where:

[pfcup:xxxxxxx]	Enables/disables priority flow control. From left to right (priorities 0-7) - x can be equal to either 0 or 1. 1 indicates that the priority is configured to transmit priority pause.

- For IEEE protocol, use lldptool:

```
lldptool -T -i <ethX> -V PFC enabled=x,x,x,x,x,x,x,x
```

Example:

```
lldptool -T -i eth2 -V PFC enabled=1,2,4
```

where:

enabled	Displays or sets the priorities with PFC enabled. The set attribute takes a comma-separated list of priorities to enable, or the string none to disable all priorities.

PFC Auto-Configuration Using LLDP in the Firmware (for mlx5 driver)

There are two ways to configure PFC and ETS on the server:

1. Local Configuration - Configuring each server manually.
2. Remote Configuration - Configuring PFC and ETS on the switch, after which the switch will pass the configuration to the server using LLDP DCBX TLVs.
   There are two ways to implement the remote configuration using mlx5 driver:
   a. Configuring the adapter firmware to enable DCBX.
   b. Configuring the host to enable DCBX.

For further information on how to auto-configure PFC using LLDP in the firmware, refer to the HowTo Auto-Config PFC and ETS on ConnectX-4 via LLDP DCBX Community post.

PFC Configuration on Switches

1. In order to enable DCBX, LLDP should first be enabled:

```
switch (config) # lldp
show lldp interfaces ethernet remote
```

2. Add DCBX to the list of supported TLVs per required interface.
   For IEEE DCBX:

```
switch (config) # interface 1/1
switch (config interface ethernet 1/1) # lldp tlv-select dcbx
```

   For CEE DCBX:

```
switch (config) # interface 1/1
switch (config interface ethernet 1/1) # lldp tlv-select dcbx-cee
```

3. [Optional] Application Priority can be configured on the switch, with the required ethertype and priority. For example, IP packet, priority 1:

```
switch (config) # dcb application-priority 0x8100 1
```

4. Make sure PFC is enabled on the host (for enabling PFC on the host, refer to PFC Configuration on Hosts section above). Once it is enabled, it will be passed in the LLDP TLVs.
5. Enable PFC with the desired priority on the Ethernet port.

```
dcb priority-flow-control enable force
dcb priority-flow-control priority <priority> enable
interface ethernet <port> dcb priority-flow-control mode on force
```

Example - Enabling PFC with priority 3 on port 1/1:

```
dcb priority-flow-control enable force
dcb priority-flow-control priority 3 enable
interface ethernet 1/1 dcb priority-flow-control mode on force
```

Priority Counters

Several ingress and egress counters per priority are supported. Run ethtool -S to get the full list of port counters.

ConnectX-4 Counters

- Rx and Tx Counters:
  - Packets
  - Bytes
  - Octets
  - Frames
  - Pause
  - Pause frames
  - Pause Duration
  - Pause Transition

Example

```
# ethtool -S eth35 | grep prio4
     prio4_rx_octets: 62147780800
     prio4_rx_frames: 14885696
     prio4_tx_octets: 0
     prio4_tx_frames: 0
     prio4_rx_pause: 0
     prio4_rx_pause_duration: 0
     prio4_tx_pause: 26832
     prio4_tx_pause_duration: 14508
     prio4_rx_pause_transition: 0
```

⚠ The Pause counters in ConnectX-4 are visible via ethtool only for priorities on which PFC is enabled.

PFC Storm Prevention

PFC storm prevention enables toggling between default and auto modes.
The stall prevention timeout is configured to 8 seconds by default. Auto mode sets the stall prevention timeout to be 100 msec.
The feature can be controlled using sysfs in the following directory: /sys/class/net/eth*/settings/pfc_stall_prevention

- To query the PFC stall prevention mode:

```
cat /sys/class/net/eth*/settings/pfc_stall_prevention
```

Example

```
$ cat /sys/class/net/ens6/settings/pfc_stall_prevention
default
```

- To configure the PFC stall prevention mode:

```
echo <option> /sys/class/net/<interface>/settings/pfc_stall_prevention
```

The following two counters were added to the ethtool -S:

- tx_pause_storm_warning_events - when the device is stalled for a period longer than a pre-configured watermark, the counter increases, allowing the debug utility an insight into current device status.
- tx_pause_storm_error_events - when the device is stalled for a period longer than a pre-configured timeout, the pause transmission is disabled, and the counter increase.

5.6.2.2.6.2 Dropless Receive Queue (RQ)

Dropless RQ feature enables the driver to notify the FW when SW receive queues are overloaded. This scenario takes place when the handling of SW receive queue is slower than the handling of the HW receive queues.
When this feature is enabled, a packet that is received while the receive queue is full will not be immediately dropped. The FW will accumulate these packets assuming posting of new WQEs will resume shortly. If received WQEs are not posted after a certain period of time, out_of_buffer counter will increase, indicating that the packet has been dropped.
This feature is disabled by default. In order to activate it, ensure that Flow Control feature is also enabled.

To enable the feature, run:

```
ethtool --set-priv-flags ens6 dropless_rq on
```

To get the feature state, run:

```
ethtool --show-priv-flags DEVNAME
```

Output example:

```
Private flags for DEVNAME:
rx_cqe_moder   : on
rx_cqe_compress: off
sniffer        : off
dropless_rq    : off
hw_lro         : off
```

To disable the feature, run:

```
ethtool --set-priv-flags ens6 dropless_rq off
```

5.6.2.2.7 Explicit Congestion Notification

Explicit congestion notification (ECN), an extension to the IP protocol, enables reliable communication by signaling congestion without dropping packets. To ensure reliable operation, all nodes along the communication path (including routers) must support ECN.

ECN is represented by two bits in the traffic control IP header and is implemented in this context for RoCE v2.

5.6.2.2.7.1 Enabling ECN

To enable ECN on the hosts:

1. Enable ECN in sysfs.

   ```
   /sys/class/net/<interface>/ecn/<protocol>/enable/<X>
   ```

2. Query the attribute.

   ```
   cat /sys/class/net/<interface>/ecn/<protocol>/params/<requested attribute>
   ```

3. Modify the attribute.

   ```
   echo <value> /sys/class/net/<interface>/ecn/<protocol>/params/<requested attribute>
   ```

ECN supports the following algorithms:

- **r_roce_ecn_rp** - Reaction point
- **r_roce_ecn_np** - Notification point

Each algorithm has a set of relevant parameters and statistics, which are defined per device, per port, per priority.

To query whether ECN is enabled per Priority X:

```
cat /sys/class/net/<interface>/ecn/<protocol>/enable/X
```

To read ECN configurable parameters:

```
cat /sys/class/net/<interface>/ecn/<protocol>/requested attributes
```

To enable ECN for each priority per protocol:

```
echo 1 > /sys/class/net/<interface>/ecn/<protocol>/enable/X
```

To modify ECN configurable parameters:

```
echo <value> > /sys/class/net/<interface>/ecn/<protocol>/requested attributes
```

Where:

- X – priority {0..7}
- protocol – **roce_rp** ; **roce_np**
- Requested attributes – Next Slide for each protocol.

5.6.2.2.8 RSS Support

The device supports using XOR as the RSS distribution function, in addition to the default Toeplitz function.

The XOR function provides better distribution across the driver's receive queues for a small number of streams, ensuring each TCP/UDP stream is assigned to a different queue.

To switch between the Toeplitz and XOR RSS hash functions, use the **sysfs** interface at: **/sys/class/net/eth*/settings/hfunc** .

To query the operational and supported hash functions:

```
cat /sys/class/net/eth*/settings/hfunc
```

Example:

```
cat /sys/class/net/eth2/settings/hfunc
Operational hfunc: toeplitz
Supported hfuncs: xor toeplitz
```

To change the operational hash function:

```
echo xor > /sys/class/net/eth*/settings/hfunc
```

5.6.2.2.8.1 RSS Verbs Support

Receive Side Scaling (RSS) technology allows spreading incoming traffic between different receive descriptor queues. Assigning each queue to different CPU cores allows better load balancing of the incoming traffic and improves performance.
This technology was extended to user space by the verbs layer and can be used for RAW ETH QP.

5.6.2.2.8.2 RSS Flow Steering

Steering rules classify incoming packets and deliver a specific traffic type (e.g. TCP/UDP, IP only) or a specific flow to "RX Hash" QP. "RX Hash" QP is responsible for spreading the traffic it handles between the Receive Work Queues using RX hash and Indirection Table. The Receive Work Queue can point to different CQs that can be associated with different CPU cores.

5.6.2.2.8.3 Verbs

The following verbs should be used to achieve this task in both control and data path. Details per verb should be referenced from its man page.

- **ibv_create_wq** , **ibv_modify_wq** , **ibv_destory_wq**
- **ibv_create_rwq_ind_table** , **ibv_destroy_rwq_ind_table**
- **ibv_create_qp_ex** with specific RX configuration to create the "RX hash" QP

5.6.2.2.9 Time Stamping

Time stamping is the process of keeping track of the creation of a packet. A time-stamping service supports assertions of proof that a datum existed before a particular time. Incoming packets are time-stamped before they are distributed on the PCI depending on the congestion in the PCI buffers. Outgoing packets are time-stamped very close to placing them on the wire.

5.6.2.2.9.1 Configuring Time Stamping

Time stamping is off by default and should be enabled before use.

Enabling Time Stamping for Socket

To enable time stamping for a socket, call **setsockopt()** with **SO_TIMESTAMPING** and with the following flags:

SOF_TIMESTAMPING_TX_HARDWARE	Try to obtain send time-stamp in hardware
SOF_TIMESTAMPING_TX_SOFTWARE	If **SOF_TIMESTAMPING_TX_HARDWARE** is off or fails, then do it in software
SOF_TIMESTAMPING_RX_HARDWARE	Return the original, unmodified time-stamp as generated by the hardware
SOF_TIMESTAMPING_RX_SOFTWARE	If **SOF_TIMESTAMPING_RX_HARDWARE** is off or fails, then do it in software
SOF_TIMESTAMPING_RAW_HARDWARE	Return original raw hardware time-stamp
SOF_TIMESTAMPING_SYS_HARDWARE	Return hardware time-stamp transformed into the system time base
SOF_TIMESTAMPING_SOFTWARE	Return system time-stamp generated in software
SOF_TIMESTAMPING_TX/RX	Determine how time-stamps are generated
SOF_TIMESTAMPING_RAW/SYS	Determine how they are reported

Enabling Time Stamping for Network Device

To enable time stamping for a network device, admin privileged users may enable/disable time stamping through calling ioctl (sock, SIOCSH-WTSTAMP, &ifreq) with the following values:

- Send side time sampling, enabled by **ifreq.hwtstamp_config.tx_type** when:

```
/* possible values for hwtstamp_config->tx_type */
enum hwtstamp_tx_types {
        /*
         * No outgoing packet will need hardware time stamping;
         * should a packet arrive which asks for it, no hardware
         * time stamping will be done.
        */
        HWTSTAMP_TX_OFF,

        /*
         * Enables hardware time stamping for outgoing packets;
         * the sender of the packet decides which are to be
         * time stamped by setting %SOF_TIMESTAMPING_TX_SOFTWARE
         * before sending the packet.
         */
        HWTSTAMP_TX_ON,
/*
         * Enables time stamping for outgoing packets just as
         * HWTSTAMP_TX_ON does, but also enables time stamp insertion
         * directly into Sync packets. In this case, transmitted Sync
         * packets will not received a time stamp via the socket error
         * queue.
         */
        HWTSTAMP_TX_ONESTEP_SYNC,
};
Note: for send side time stamping currently only HWTSTAMP_TX_OFF and
HWTSTAMP_TX_ON are supported.
```

- Receive side time sampling, enabled by **ifreq.hwtstamp_config.rx_filter** when:

```
/* possible values for hwtstamp_config->rx_filter */
enum hwtstamp_rx_filters {
        /* time stamp no incoming packet at all */
        HWTSTAMP_FILTER_NONE,

        /* time stamp any incoming packet */
        HWTSTAMP_FILTER_ALL,
      /* return value: time stamp all packets requested plus some others */
        HWTSTAMP_FILTER_SOME,

        /* PTP v1, UDP, any kind of event packet */
        HWTSTAMP_FILTER_PTP_V1_L4_EVENT,
        /* PTP v1, UDP, Sync packet */
        HWTSTAMP_FILTER_PTP_V1_L4_SYNC,
        /* PTP v1, UDP, Delay_req packet */
        HWTSTAMP_FILTER_PTP_V1_L4_DELAY_REQ,
        /* PTP v2, UDP, any kind of event packet */
        HWTSTAMP_FILTER_PTP_V2_L4_EVENT,
        /* PTP v2, UDP, Sync packet */
        HWTSTAMP_FILTER_PTP_V2_L4_SYNC,
        /* PTP v2, UDP, Delay_req packet */
        HWTSTAMP_FILTER_PTP_V2_L4_DELAY_REQ,

        /* 802.AS1, Ethernet, any kind of event packet */
        HWTSTAMP_FILTER_PTP_V2_L2_EVENT,
        /* 802.AS1, Ethernet, Sync packet */
        HWTSTAMP_FILTER_PTP_V2_L2_SYNC,
        /* 802.AS1, Ethernet, Delay_req packet */
        HWTSTAMP_FILTER_PTP_V2_L2_DELAY_REQ,

        /* PTP v2/802.AS1, any layer, any kind of event packet */
        HWTSTAMP_FILTER_PTP_V2_EVENT,
        /* PTP v2/802.AS1, any layer, Sync packet */
        HWTSTAMP_FILTER_PTP_V2_SYNC,
        /* PTP v2/802.AS1, any layer, Delay_req packet */
        HWTSTAMP_FILTER_PTP_V2_DELAY_REQ,
};
Note: for receive side time stamping currently only HWTSTAMP_FILTER_NONE and
HWTSTAMP_FILTER_ALL are supported.
```

Getting Time Stamping

Once time stamping is enabled time stamp is placed in the socket Ancillary data. recvmsg() can be used to get this control message for regular incoming packets. For send time stamps the outgoing packet is looped back to the socket's error queue with the send time-stamp(s) attached. It can be received with recvmsg (**flags=MSG_ERRQUEUE**). The call returns the original outgoing packet data including all headers prepended down to and including the link layer, the **scm_time-**

stamping control message and a **sock_extended_err** control message with **ee_errno==ENOMSG** and **ee_origin==SO_EE_ORIGIN_TIMESTAMPING**. A socket with such a pending bounced packet is ready for reading as far as select() is concerned. If the outgoing packet has to be fragmented, then only the first fragment is time stamped and returned to the sending socket.

⚠ When time stamping is enabled, VLAN stripping is disabled. For more info please refer to **Documentation/networking/timestamping.txt** in kernel.org.

⚠ On ConnectX-4 adapters and above, when time stamping is enabled, RX CQE compression is disabled (features are mutually exclusive).

Time Stamping Capabilities via ethtool

To display time-stamping capabilities via ethtool:

```
ethtool -T eth<x>
```

Example output:

```
ethtool -T eth0
Time stamping parameters for p2p1:
Capabilities:
                                hardware-transmit           (SOF_TIMESTAMPING_TX_HARDWARE)
                                software-transmit           (SOF_TIMESTAMPING_TX_SOFTWARE)
                                hardware-receive            (SOF_TIMESTAMPING_RX_HARDWARE)
                                software-receive            (SOF_TIMESTAMPING_RX_SOFTWARE)
                                software-system-clock       (SOF_TIMESTAMPING_SOFTWARE)
                                hardware-raw-clock          (SOF_TIMESTAMPING_RAW_HARDWARE)
PTP Hardware Clock: 1
Hardware Transmit Timestamp Modes:
off                              (HWTSTAMP_TX_OFF)
on                               (HWTSTAMP_TX_ON)

Hardware Receive Filter Modes:
none                            (HWTSTAMP_FILTER_NONE)
all                             (HWTSTAMP_FILTER_ALL)
```

For more details on PTP Hardware Clock, please refer to: https://www.kernel.org/doc/Documentation/ptp/ptp.txt

Steering PTP Traffic to Single RX Ring

As a result of Receive Side Steering (RSS) PTP traffic coming to UDP ports 319 and 320, it may reach the user space application in an out of order manner. In order to prevent this, PTP traffic needs to be steered to single RX ring using ethtool.

Example:

```
# ethtool -u ens7
8 RX rings available
Total 0 rules
# ethtool -U ens7 flow-type udp4 dst-port 319 action 0 loc 1
# ethtool -U ens7 flow-type udp4 dst-port 320 action 0 loc 0
# ethtool -u ens7
8 RX rings available
Total 2 rules
Filter: 0
Rule Type: UDP over IPv4
Src IP addr: 0.0.0.0 mask: 255.255.255.255
Dest IP addr: 0.0.0.0 mask: 255.255.255.255
TOS: 0x0 mask: 0xff
Src port: 0 mask: 0xffff
Dest port: 320 mask: 0x0
Action: Direct to queue 0
Filter: 1
Rule Type: UDP over IPv4
Src IP addr: 0.0.0.0 mask: 255.255.255.255
Dest IP addr: 0.0.0.0 mask: 255.255.255.255
```

```
TOS: 0x0 mask: 0xff
Src port: 0 mask: 0xffff
Dest port: 319 mask: 0x0
Action: Direct to queue 0
```

Tx Port Time Stamping

> ⚠ This feature is supported on ConnectX-6 Dx and above adapter cards only.

Transmitted packet time-stamping accuracy can be improved when using a timestamp generated at the port level instead of a timestamp generated upon CQE creation. Tx port time stamping better reflects the actual time of a packet's transmission.

Normal Send queues (SQs) are open with CQE time-stamp support. When this feature is enabled, the driver is expected to open extra Tx port time-stamped SQ per traffic class (TC).

The stream must meet the following conditions in order to be transmitted through a Tx port time-stamped SQ.

1. **SKBTX_HW_TSTAMP** flag was set at **tx_flag** (**SO_TIMESTAMPING** was set via **setsockopt()** or similarly)
2. Packet type is:
    - Non-IP, with EtherType of PTP over IEEE 802.3 (0x88f7); or
    - UDP over IPv4/IPv6

This feature is disabled by default in order to avoid extra SQ memory allocations. The feature can be enabled or disabled using the following command:

```
ethtool --set-priv-flags <ifs-name> tx_port_ts <on/off>
```

PTP Cyc2time Hardware Translation Offload

> ⚠ This feature is supported on ConnectX-6 Dx and above adapter cards only.

The device timestamp operates in one of two modes: real-time or free-running internal time.

- Free-Running Internal Time Mode – In this mode, the device clock cannot be edited. Driver or user space must adjust the timestamps to real-time nanoseconds.
- Real-Time Mode – In this mode, the hardware clock can be adjusted, providing timestamps already translated into real-time nanoseconds.

Both modes are global per device. Once a mode is configured, all clock-related features (e.g., PPS, CQE TS, PCIe BAR) operate exclusively with the selected clock mode.

By default, the hardware is set to free-running internal time mode. The driver adjusts the hardware real-time clock based on PTP daemon clock updates.

Only Physical Functions (PFs) can modify the hardware real-time clock. Adjustments from Virtual Functions (VFs) are ignored (treated as no-operations). If multiple PFs attempt to modify the clock, the device selects one as the designated clock provider. If the designated provider fails to send updates within a specific period, the device may replace it with another PF. Other inputs during this period are also treated as no-operations.

Timestamp Format

CQE hardware timestamp format for ConnectX-6 Dx and ConnectX-6 Lx NICs is 64 bit, as follows:

```
{32bit sec, 32 bit nsec}
```

Configuration

To enable the feature:

1. Set **REAL_TIME_CLOCK_ENABLE** in **NV_CONFIG** via mlxconfig
2. Restart the driver.

Limitations

- Administrator must restart the driver and perform a FW reset for the configuration to take effect. Otherwise, mismatch between HW and driver timestamp mode might occur.
- Once real time mode is activated on a given device (see configuration section), version 5.3 or newer must run on all device functions. Any older driver running on a device function at this configuration will fail to open any traffic queues (RDMA or ETH), hence becoming dysfunctional.
- In real time mode, all device functions must be PTP-synchronized by a single clock domain —do not use multiple GMs for different functions on the same device.
- Regarding hardware clock ownership, the hardware is configured only from a single elected function; other function settings are ignored by the device. There is no indication as to which function is the hardware-clock's owner. After an internal timeout without modifying the hardware clock, a function losses the hardware-clock's ownership and is open to be grasped by any of the functions.
- All PFs/VFs within the same device must sync to the same 1588 master clock. If multiple masters are used, the device will use a single elected function. This might lead to wrong clock representation by device, wrong 1588 TLVs and hiccups on replacement of elected function.
- This feature is supported on ConnectX-6 Dx and above adapter cards only.

5.6.2.2.9.2 RoCE Time Stamping

RoCE time stamping allows you to stamp packets when they are sent to the wire/received from the wire. The time-stamp is given in raw hardware cycles but could be easily converted into hardware referenced nanoseconds based time. Additionally, it enables you to query the hardware for the hardware time, thus stamp other application's event and compare time.

Query Capabilities

Time stamping is available if and only the hardware reports it is capable of reporting it. To verify whether RoCE time stamping is available, run **ibv_query_device_ex**.

For further information, please see ibv_query_device_ex manual page.

Creating a Time Stamping Completion Queue

To get time stamps, a suitable extended completion queue (CQ) must be created via a special call to ibv_create_cq_ex verb.

For further information, please see ibv_create_cq_ex manual page.

> ⚠ Time Stamping in not available when CQE zipping is used.

Querying the Hardware Time

Querying the hardware for time is done via the **ibv_query_rt_values_ex** verb. For example:

For further information, please see ibv_query_rt_values_ex manual page.

5.6.2.2.9.3 One Pulse Per Second (1PPS)

1PPS is a time synchronization feature that allows the adapter to be able to send or receive 1 pulse per second on a dedicated pin on the adapter card using an SMA connector (SubMiniature version A). Only one pin is supported and could be configured as 1PPS in or 1PPS out.

For further information, refer to HowTo Test 1PPS on NVIDIA Adapters Community post.

5.6.2.2.10 Flow Steering

Flow steering is a new model which steers network flows based on flow specifications to specific QPs. Those flows can be either unicast or multicast network flows. In order to maintain flexibility, domains and priorities are used. Flow steering uses a methodology of flow attribute, which is a combination of L2-L4 flow specifications, a destination QP and a priority. Flow steering rules may be inserted either by using ethtool or by using InfiniBand verbs. The verbs abstraction uses different terminology from the flow attribute (ibv_flow_attr), defined by a combination of specifications (struct ibv_flow_spec_*).

5.6.2.2.10.1 Flow Steering Support

All flow steering features are enabled in the supported adapter cards.

5.6.2.2.10.2 Flow Domains and Priorities

Flow steering defines the concept of domain and priority. Each domain represents a user agent that can attach a flow. The domains are prioritized. A higher priority domain will always supersede a lower priority domain when their flow specifications overlap. Setting a lower priority value will result in a higher priority.
In addition to the domain, there is a priority within each of the domains. Each domain can have at most 2^12 priorities in accordance with its needs.
The following are the domains at a descending order of priority:

- User Verbs allows a user application QP to be attached to a specified flow when using **ibv_create_flow** and **ibv_destroy_flow** verbs
  - **ibv_create_flow**

    ```
    struct ibv_flow *ibv_create_flow(struct ibv_qp *qp, struct ibv_flow_attr
    *flow)
    ```

    Input parameters:
    - **struct ibv_qp** - the attached QP.

- **struct ibv_flow_attr** - attaches the QP to the flow specified. The flow contains mandatory control parameters and optional L2, L3 and L4 headers. The optional headers are detected by setting the size and **num_of_specs** fields: **struct ibv_flow_attr** can be followed by the optional flow headers structs:

```
struct ibv_flow_spec_eth
struct ibv_flow_spec_ipv4
struct ibv_flow_spec_tcp_udp
struct ibv_flow_spec_ipv6
```

  For further information, please refer to the **ibv_create_flow** man page.

- **ibv_destroy_flow**

```
int ibv_destroy_flow(struct ibv_flow *flow_id)
```

  Input parameters:
  ibv_destroy_flow requires struct **ibv_low** which is the return value of **ibv_create_flow** in case of success.
  Output parameters:
  Returns 0 on success, or the value of errno on failure.
  For further information, please refer to the **ibv_destroy_flow** man page.

5.6.2.2.10.3 Ethtool

Ethtool domain is used to attach an RX ring, specifically its QP to a specified flow. Please refer to the most recent ethtool man page for all the ways to specify a flow.

Examples:

- ethtool –U eth5 flow-type ether dst 00:11:22:33:44:55 loc 5 action 2
  All packets that contain the above destination MAC address are to be steered into rx-ring 2 (its underlying QP), with priority 5 (within the ethtool domain)
- ethtool –U eth5 flow-type tcp4 src-ip 1.2.3.4 dst-port 8888 loc 5 action 2
  All packets that contain the above destination IP address and source port are to be steered into rx- ring 2. When destination MAC is not given, the user's destination MAC is filled automatically.
- ethtool -U eth5 flow-type ether dst 00:11:22:33:44:55 vlan 45 m 0xf000 loc 5 action 2
  All packets that contain the above destination MAC address and specific VLAN are steered into ring 2. Please pay attention to the VLAN's mask 0xf000. It is required in order to add such a rule.
- ethtool –u eth5
  Shows all of ethtool's steering rule

When configuring two rules with the same priority, the second rule will overwrite the first one, so this ethtool interface is effectively a table. Inserting Flow Steering rules in the kernel requires support from both the ethtool in the user space and in kernel (v2.6.28).

5.6.2.2.10.4 Accelerated Receive Flow Steering (aRFS)

Receive Flow Steering (RFS) and Accelerated Receive Flow Steering (aRFS) are kernel features currently available in most distributions. For RFS, packets are forwarded based on the location of

the application consuming the packet. aRFS boosts the speed of RFS by adding support for the hardware. By usingaRFS(unlike RFS), the packets are directed to a CPU that is local to the thread running the application.

aRFSis an in-kernel-logic responsible for load balancing between CPUs by attaching flows to CPUs that are used by flow's owner applications. This domain allows the aRFS mechanism to use the flow steering infrastructure to support the aRFS logic by implementing the ndo_rx_flow_steer, which, in turn, calls the underlying flow steering mechanism with the aRFS domain.

To configure RFS:

Configure the RFS flow table entries (globally and per core).
Note: The functionality remains disabled until explicitly configured (by default it is 0).

- The number of entries in the global flow table is set as follows:

  /proc/sys/net/core/rps_sock_flow_entries

- The number of entries in the per-queue flow table are set as follows:

  /sys/class/net/<dev>/queues/rx-<n>/rps_flow_cnt

Example:

```
# echo 32768 > /proc/sys/net/core/rps_sock_flow_entries
# NUM_CHANNELS=`ethtool -l ens6 | grep "Combined:" | tail -1 | awk '{print $2}'`
# for f in `seq 0 $((NUM_CHANNELS-1))`; do echo 32768 > /sys/class/net/ens6/queues/rx-$f/rps_flow_cnt; done
```

To Configure aRFS:

The aRFS feature requires explicit configuration in order to enable it. Enabling the aRFS requires enabling the 'ntuple' flag via the ethtool.
For example, to enable ntuple for eth0, run:

```
ethtool -K eth0 ntuple on
```

aRFS requires the kernel to be compiled with the **CONFIG_RFS_ACCEL** option. This option is available in kernels 2.6.39 and above. Furthermore, aRFS requires Device Managed Flow Steering support.

RFS cannot function if LRO is enabled. LRO can be disabled via ethtool.

5.6.2.2.10.5 Flow Steering Dump Tool

The mlx_fs_dump is a python tool that prints the steering rules in a readable manner. Python v2.7 or above, as well as pip, anytree and termcolor libraries are required to be installed on the host.

Running example:

```
./ofed_scripts/utils/mlx_fs_dump -d /dev/mst/mt4115_pciconf0
FT: 9 (level: 0x18, type: NIC_RX)
+-- FG: 0x15 (MISC)
    |-- FTE: 0x0 (FWD) to (TIR:0x7e) out.ethtype:IPv4 out.ip_prot:UDP out.udp_dport:0x140
    +-- FTE: 0x1 (FWD) to (TIR:0x7e) out.ethtype:IPv4 out.ip_prot:UDP out.udp_dport:0x13f
...
```

For further information on the mlx_fs_dump tool, please refer to mlx_fs_dump Community post.

5.6.2.2.11 Wake-on-LAN

Wake-on-LAN (WoL) is a technology that allows a network professional to remotely power on a computer or to wake it up from sleep mode.

- To enable WoL:

  ```
  ethtool -s <interface> wol g
  ```

- To get WoL:

  ```
  ethtool <interface> | grep Wake-on Wake-on: g
  ```

  Where **g** is the magic packet activity.

5.6.2.2.12 Q-in-Q Tunneling

Q-in-Q tunneling (hardware-accelerated 802.1ad VLAN) enables the creation of a Layer 2 Ethernet connection between two servers. It allows segregation of different VLAN traffic on a link or bundling multiple VLANs into a single VLAN. This is achieved by adding a service VLAN tag before the user's existing 802.1Q VLAN tags.

> For details on Q-in-Q support in virtualized environments (SR-IOV), refer to "Q-in-Q Encapsulation per VF in Linux".

To enable device support for accelerated 802.1ad VLAN:

1. Turn on the new ethtool private flag **phv-bit** (disabled by default).

   ```
   $ ethtool --set-priv-flags eth1 phv-bit on
   ```

   Enabling this flag sets the **phv_en** port capability.
2. Change the interface device features by turning on the ethtool device feature **tx-vlan-stag-hw-insert** (disabled by default).

```
$ ethtool -K eth1 tx-vlan-stag-hw-insert on
```

Once the private flag and the ethtool device feature are set, the device will be ready for 802.1ad VLAN acceleration.

> ⚠ The **phv-bit** private flag setting is available for the PF only. The VF can use the VLAN acceleration by setting the **tx-vlan-stag-hw-insert** parameter only if the private flag **phv-bit** is enabled by the PF. If the PF enables/disables the **phv-bit** flag after the VF driver is up, the configuration will take place only after the VF driver is restarted.

5.6.2.2.13 VLAN Stripping in Linux Verbs

> ⚠ This capability is now accessible from userspace using the verbs.

VLAN stripping adds access to the device's ability to offload the Customer VLAN (cVLAN) header stripping from an incoming packet, thus achieving acceleration of VLAN handing in receive flow.

It is configured per WQ option. You can either enable it upon creation or modify it later using the appropriate verbs (**ibv_create_wq** / **ibv_modify_wq**).

5.6.2.2.14 Offloaded Traffic Sniffer

> ⚠ To be able to activate this feature, make sure libpcap library v1.9 or above is installed on your setup.
>
> To download libpcap, please visit https://www.tcpdump.org/.

Offloaded Traffic Sniffer allows bypass kernel traffic (such as RoCE, VMA, and DPDK) to be captured by existing packet analyzer, such as tcpdump.

To capture the interface's bypass kernel traffic, run tcpdump on the RDMA device.

For examples on how to dump RDMA traffic using the Inbox tcpdump tool for ConnectX-4 adapter cards and above, click here.

> ⚠ Note that enabling Offloaded Traffic Sniffer can cause bypass kernel traffic speed degradation.

> ⚠ In case you do not wish to install libpcap on your setup, you can use docker to run the tcpdump. For further information, please see https://hub.docker.com/r/mellanox/tcpdump-rdma.

5.6.2.2.15 Dump Configuration

This feature helps dumping driver and firmware configuration using ethtool. It creates a backup of the configuration files into a specified dump file.

5.6.2.2.15.1 Dump Parameters (Bitmap Flag)

The following bitmap parameters are used to set the type of dump. If a value is not set, the default value used is "0".

Bitmap Parameters

Value	Description
1	MST dump
2	Ring dump (Software context information for SQs, EQs, RQs, CQs)
3	MST dump + Ring dump (1+2)
4	Clear this parameter

5.6.2.2.15.2 Configuration

In order to configure this feature, follow the steps below:

1. Set the dump bitmap parameter by running -W (uppercase) with the desired bitmap parameter value (see Bitmap Parameters table above). In the following example, the bitmap parameter value is 3.

   ```
   ethtool -W ens1f0 3
   ```

2. Dump the file by running -w (lowercase) with the desired configuration file name.

   ```
   ethtool -w ens1f0 data /tmp/dump.bin
   ```

3. [Optional] To get the bitmap parameter value, version and size of the dump, run the command above without the file name.

   ```
   ethtool -w ens1f0
   flag: 3, version: 1, length: 4312
   ```

4. To open the dump file, run:

   ```
   mlnx_dump_parser -f /tmp/dump.bin -m mst_dump_demo.txt -r ring_dump_demo.txt
   Version: 1 Flag: 3 Number of blocks: 123 Length 327584
   MCION module number: 0 status: | present |
   DRIVER VERSION: 1-23 (03 Mar 2015)
   DEVICE NAME 0000:81:00.0:ens1f0
   Parsing Complete!
   ```

   where:

-f	For the file to be parsed (the file that was just created)
-m	For the mst dump file
-r	For the ring dump file

For further information, refer to HowTo Dump Driver Configuration (via ethtool) Community post.
Output:

```
# mlnx_dump_parser -f /tmp/dump.bin -m mst_dump_demo.txt -r ring_dump_demo.txt
Version: 1 Flag: 3 Number of blocks: 123 Length 327584
MCION module number: 0 status: | present |
DRIVER VERSION: 1-23 (03 Mar 2015)
DEVICE NAME 0000:81:00.0:ens1f0
Parsing Complete!
```

5. Open the files.
   a. The MST dump file will look as follows. In order to analyze it, contact NVIDIA Support.

```
cat mst_dump_demo.txt
0x00000000 0x01002000
0x00000004 0x00000000
0x00000008 0x00000000
0x0000000c 0x00000000
0x00000010 0x00000000
0x00000014 0x00000000
0x00000018 0x00000000
...
```

   b. The Ring dump file can help developers debug ring-related issues, and it looks as follows:

```
# cat ring_dump_demo.txt
SQ TYPE: 3, WQN: 102, PI: 0, CI: 0, STRIDE: 6, SIZE: 1024...
SQ TYPE: 3, WQN: 102, PI: 0, CI: 0, STRIDE: 6, SIZE: 1024, WQE_NUM: 65536, GROUP_IP: 0
CQ TYPE: 5, WQN: 20, PI: 0, CI: 0, STRIDE: 6, SIZE: 1024, WQE_NUM: 1024, GROUP_IP: 0
RQ TYPE: 4, WQN: 103, PI: 15, CI: 0, STRIDE: 5, SIZE: 16, WQE_NUM: 512, GROUP_IP: 0
CQ TYPE: 5, WQN: 21, PI: 0, CI: 0, STRIDE: 6, SIZE: 16384, WQE_NUM: 16384, GROUP_IP: 0
EQ TYPE: 6, CI: 1, SIZE: 0, IRQN: 109, EQN: 19, NENT: 2048, MASK: 0, INDEX: 0, GROUP_ID: 0
SQ TYPE: 3, WQN: 106, PI: 0, CI: 0, STRIDE: 6, SIZE: 1024, WQE_NUM: 65536, GROUP_IP: 1
CQ TYPE: 5, WQN: 23, PI: 0, CI: 0, STRIDE: 6, SIZE: 1024, WQE_NUM: 1024, GROUP_IP: 1
RQ TYPE: 4, WQN: 107, PI: 15, CI: 0, STRIDE: 5, SIZE: 16, WQE_NUM: 512, GROUP_IP: 1
CQ TYPE: 5, WQN: 24, PI: 0, CI: 0, STRIDE: 6, SIZE: 16384, WQE_NUM: 16384, GROUP_IP: 1
EQ TYPE: 6, CI: 1, SIZE: 0, IRQN: 110, EQN: 20, NENT: 2048, MASK: 0, INDEX: 1, GROUP_ID: 1
SQ TYPE: 3, WQN: 110, PI: 0, CI: 0, STRIDE: 6, SIZE: 1024, WQE_NUM: 65536, GROUP_IP: 2
CQ TYPE: 5, WQN: 26, PI: 0, CI: 0, STRIDE: 6, SIZE: 1024, WQE_NUM: 1024, GROUP_IP: 2
RQ TYPE: 4, WQN: 111, PI: 15, CI: 0, STRIDE: 5, SIZE: 16, WQE_NUM: 512, GROUP_IP: 2
CQ TYPE: 5, WQN: 27, PI: 0, CI: 0, STRIDE: 6, SIZE: 16384, WQE_NUM: 16384, GROUP_IP: 2
...
```

5.6.2.2.16 Local Loopback Disable

Local Loopback Disable allows users to force the disablement of local loopback on the virtual port (vport). This disables both unicast and multicast loopback in the hardware.

- To enable Local Loopback Disable, run:

```
echo 1 > /sys/class/net/<ifname>/settings/force_local_lb_disable"
```

- To disable Local Loopback Disable, run:

```
echo 0 > /sys/class/net/<ifname>/settings/force_local_lb_disable"
```

> ⚠ When turned off, the driver configures the loopback mode according to its own logic.

5.6.2.2.17 kTLS Offloads

> ⚠ This feature is supported on NVIDIA® ConnectX®-6 Dx and NVIDIA® BlueField®-2 crypto devices onwards.

Transport Layer Security (TLS) is a widely-deployed protocol used for securing TCP connections on the Internet. TLS is also a required feature for HTTP/2, the latest web standard. Kernel implementation of TLS (kTLS) provides new opportunities for offloading the protocol into the hardware.

TLS data-path offload allows the NIC to accelerate encryption, decryption and authentication of AES-GCM. TLS offload handles data as it goes through the device without storing any data, but only updating context. If the packet cannot be encrypted/decrypted by the device, then a software fallback handles the packet.

5.6.2.2.17.1 Establishing a kTLS Connection

To avoid unnecessary complexity in the kernel, the TLS handshake is kept in the user space. A full TLS connection using the socket is done using the following scheme:

1. Call **`connect()`** or **`accept()`** on a standard TCP file descriptor.
2. Use a user space TLS library to complete a handshake.
3. Create a new KTLS socket file descriptor.
4. Extract the TLS Initialization Vectors (IVs), session keys, and sequence IDs from the TLS library. Use the **`setsockopt`** function on the kTLS file descriptor (FD) to pass them to the kernel.
5. Use standard **`read()`**, **`write()`**, **`sendfile()`** and **`splice()`** system calls on the KTLS FD.

Drivers can offer Tx and Rx packet encryption/decryption offload from the kernel into the NIC hardware. Upon receipt of a non-data TLS message (a control message), the kTLS socket returns an error, and the message is left on the original TCP socket instead. The kTLS socket is automatically unattached. Transfer of control back to the original encrypted FD is done by calling **`getsockopt`** to receive the current sequence numbers, and inserting them into the TLS library.

5.6.2.2.17.2 Kernel Support

For support in the kernel, make sure the following flags are set as follows.

- **`CONFIG_TLS=y`**
- **`CONFIG_TLS_DEVICE=y | m`**

> ⚠ For kTLS Tx device offloads with OFED drivers, kernel TLS module (kernel/net/tls) must be aligned to kernel v5.3 and above.
>
> For kTLS Rx device offloads with OFED drivers, kernel TLS module (kernel/net/tls) must be aligned to kernel v5.9 and above.

5.6.2.2.17.3 Configuring kTLS Offloads

To enable kTLS Tx offload, run:

```
ethtool -K <ifs> tls-hw-tx-offload on
```

To enable kTLS Rx offload, run:

```
ethtool -K <ifs> tls-hw-rx-offload on
```

For further information on TLS offloads, please visit the following kernel documentation:

- Kernel TLS Offload
- Kernel TLS

5.6.2.2.17.4 OpenSSL with kTLS Offload

OpenSSL version 3.0.0 or above is required to support kTLS TX/RX offloads.

Supported OpenSSL version is available to download from distro packages, or can be downloaded and compiled from the OpenSSL github.

For a configuration example, please refer to the DOCA TLS Offload Guide.

5.6.2.2.18 IPsec Crypto Offload

This feature is supported on crypto-enabled products of NVIDIA® BlueField®-2 DPUs, and NVIDIA® ConnectX®-6 Dx and ConnectX-7 adapters (but not of ConnectX-6).

Newer/future crypto-enabled DPU and adapter product generations should also support the feature, unless explicitly stated in their documentation.

For BlueField-2 and ConnectX-6 Dx devices only: If your target application will utilize bandwidth of 100Gb/s or higher, where a substantial part of the bandwidth will be allocated for IPsec traffic, please refer to the *NVIDIA BlueField-2 DPUs Product Release Notes* or *NVIDIA ConnectX-6 Dx Adapters Product Release Notes* to learn about a potential bandwidth limitation. To access the relevant product release notes, please contact your NVIDIA sales representative.

IPsec crypto offload feature, also known as IPsec inline offload or IPsec aware offload feature enables the user to offload IPsec crypto encryption and decryption operations to the hardware.

The hardware implementation only supports AES-GCM encryption scheme.

5.6.2.2.18.1 Enabling IPsec Crypto Offload

To enable the feature, support in both kernel and adapter firmware is required.

- To add IPsec crypto offload support in the kernel, set the following flags accordingly:

```
CONFIG_XFRM_OFFLOAD=y
CONFIG_INET_ESP_OFFLOAD=m
CONFIG_INET6_ESP_OFFLOAD=m
```

> These flags are enabled by default in RedHat 8 and Ubuntu 18.04.0.

- To check whether IPsec crypto offload is supported in firmware, look for the following string in the dmesg:

```
mlx5e: IPSec ESP acceleration enabled
```

5.6.2.2.18.2 Configuring Security Associations for IPsec Offloads

To program the inline offload security associations (SA), add the option **offload dev <netdev interface> dir out/in** in the **ip xfrm state** command for transmitting and receiving SA.

- Transmit inline offload SA xfrm command example:

```
sudo ip xfrm state add src 192.168.1.64/24 dst 192.168.1.65/24 proto esp spi 0x46dc6204 reqid 0x46dc6204
 mode transport aead 'rfc4106(gcm(aes))'
0x60bd6c3eafba371a46411830fd56c53af93883261ed1fb26767820ff493f43ba35b0dcca 128 offload dev p4p1 dir out sel
src 192.168.1.64 dst 192.168.1.65
```

- Receive inline offload SA xfrm command example:

```
sudo ip xfrm state add src 192.168.1.65/24 dst 192.168.1.64/24 proto esp spi 0xaea0846c reqid 0xaea0846c
 mode transport aead 'rfc4106(gcm(aes))'
0x81d5c3167c912c1dd50dab0cb4b6d815b6ace8844304db362215a258cd19deda8f89deda 128 offload dev p4p1 dir in sel
src 192.168.1.65 dst 192.168.1.64
```

Example of setting xfrm policies:

- First server:

```
+ sudo ip xfrm state add src 192.168.1.64/24 dst 192.168.1.65/24 proto esp spi 0x28f39549 reqid 0x28f39549
 mode transport aead 'rfc4106(gcm(aes))'
0x492e8ffe718a95a00c1893ea61afc64997f4732848ccfe6ea07db483175cb18de9ae411a 128 offload dev enp4s0 dir out
sel src 192.168.1.64 dst 192.168.1.65
+ sudo ip xfrm state add src 192.168.1.65/24 dst 192.168.1.64/24 proto esp spi 0x622a73b4 reqid 0x622a73b4
 mode transport aead 'rfc4106(gcm(aes))'
0x093bfee2212802d626716815f862da31bcc7d9c44cfe3ab8049e7604b2feb1254869d25b 128 offload dev enp4s0 dir in
sel src 192.168.1.65 dst 192.168.1.64
+ sudo ip xfrm policy add src 192.168.1.64 dst 192.168.1.65 dir out tmpl src 192.168.1.64/24 dst
192.168.1.65/24 proto esp reqid 0x28f39549 mode transport
+ sudo ip xfrm policy add src 192.168.1.65 dst 192.168.1.64 dir in tmpl src 192.168.1.65/24 dst
192.168.1.64/24 proto esp reqid 0x622a73b4 mode transport
+ sudo ip xfrm policy add src 192.168.1.65 dst 192.168.1.64 dir fwd tmpl src 192.168.1.65/24 dst
192.168.1.64/24 proto esp reqid 0x622a73b4 mode transport
```

- Second server:

```
+ ssh -A -t root@l-csi-0921d /bin/bash
+ set -e
+ '[' 0 == 1 ']'
+ sudo ip xfrm state add src 192.168.1.64/24 dst 192.168.1.65/24 proto esp spi 0x28f39549 reqid 0x28f39549
 mode transport aead 'rfc4106(gcm(aes))'
0x492e8ffe718a95a00c1893ea61afc64997f4732848ccfe6ea07db483175cb18de9ae411a 128 offload dev enp4s0 dir in
sel src 192.168.1.64 dst 192.168.1.65
+ sudo ip xfrm state add src 192.168.1.65/24 dst 192.168.1.64/24 proto esp spi 0x622a73b4 reqid 0x622a73b4
 mode transport aead 'rfc4106(gcm(aes))'
0x093bfee2212802d626716815f862da31bcc7d9c44cfe3ab8049e7604b2feb1254869d25b 128 offload dev enp4s0 dir out
sel src 192.168.1.65 dst 192.168.1.64
+ sudo ip xfrm policy add src 192.168.1.65 dst 192.168.1.64 dir out tmpl src 192.168.1.65/24 dst
192.168.1.64/24 proto esp reqid 0x622a73b4 mode transport
```

```
+ sudo ip xfrm policy add src 192.168.1.64 dst 192.168.1.65 dir in tmpl src 192.168.1.64/24 dst
192.168.1.65/24 proto esp reqid 0x28f39549 mode transport
+ sudo ip xfrm policy add src 192.168.1.64 dst 192.168.1.65 dir fwd tmpl src 192.168.1.64/24 dst
192.168.1.65/24 proto esp reqid 0x28f39549 mode transport
+ echo 'IPSec tunnel configured successfully'
```

5.6.2.2.19 IPsec Packet Offload

> This feature is supported on crypto-enabled products of BlueField-2 DPUs, as well as on ConnectX-6 Dx, ConnectX-6 Lx and ConnectX-7 adapter cards. Note that it is not supported on ConnectX-6 cards.
>
> Newer/future crypto-enabled DPU and adapter product generations should also support this feature, unless explicitly stated otherwise in their documentation.

> When using NVIDIA® BlueField®-2 DPUs and NVIDIA® ConnectX®-6 Dx adapters only: If your target application utilizes 100Gb/s or a higher bandwidth, where a substantial part of the bandwidth is allocated for IPsec traffic, please refer to the relevant DPU or adapter card Product Release Notes to learn about a potential bandwidth limitation. To access the Release Notes, visit NVIDIA Networking's documentation website, or contact your NVIDIA sales representative.

> ConnectX-6 Dx adapters only support Full Offload: Encrypted Overlay (where a Hypervisor controls IPsec offload - See for example OVS IPsec - https://docs.openvswitch.org/en/latest/tutorials/ipsec/) in a Linux OS with NVIDIA drivers.

> This feature requires Linux kernel v6.6, or higher.

> IPsec tunnel mode is supported in alpha-level when controlled by VM (VF capability) only.

This feature is designed to enable IPsec full offload in switchdev mode. The **ip-xfrm** command is used to configure IPsec states and policies, and it is similar to legacy mode configuration. However, there are several limitations to the use of full offload in this mode:

1. Only IPsec Transport Mode and Tunnel Mode are supported.
2. The first IPsec TX state/policy is not allowed to be offloaded if any offloaded TC rule exists, and the same applies for the first RX state/policy. More specifically, IPsec RX/TX tables must be created before offloading any TC rule. For this reason, it is a common practice to configure IPsec rules before adding any TC rule.

Following is an example for IPsec configuration with a VXLan tunnel:

- Enable switchdev mode:

```
echo 1 > /sys/class/net/$PF0 /device/sriov_numvfs
echo 0000:08:00.2 > /sys/bus/pci/drivers/mlx5_core/unbind
devlink dev param set pci/0000:08:00.0 name flow_steering_mode value dmfs cmode runtime
devlink dev eswitch set pci/0000:08:00.0 mode switchdev
echo 0000:08:00.2 > /sys/bus/pci/drivers/mlx5_core/bind
```

- Configure PF/VF/REP netdevices, and place a VF in a namespace:

```
ifconfig $PF $LOCAL_TUN/16 up
ip l set dev $PF mtu 2000

ifconfig $REP up
ip netns add ns0
ip link set dev $VF netns ns0
ip netns exec ns0 ifconfig $VF $IP/16 up
```

- Configure IPsec states and policies:

```
ip xfrm state add src $LOCAL_TUN/16 dst $REMOTE_IP/16 proto esp spi 0xb29ed314 reqid 0xb29ed314 mode
transport aead 'rfc4106(gcm(aes))' 0x20f01f80a26f633d85617465686c32552c92c42f 128 offload packet dev $PF
dir out sel src $LOCAL_TUN/16 dst $REMOTE_IP/16 flag esn replay-window 64
ip xfrm state add src $REMOTE_IP/16 dst $LOCAL_TUN/16 proto esp spi 0xc35aa26e reqid 0xc35aa26e mode
transport aead 'rfc4106(gcm(aes))' 0x6cb228189b4c6e82e66e46920a2cde39187de4ba 128 offload packet dev $PF
dir in sel src $REMOTE_IP/16 dst $LOCAL_TUN/16 flag esn replay-window 64

ip xfrm policy add src $LOCAL_TUN dst $REMOTE_IP offload packet dev $PF dir out tmpl src $LOCAL_TUN/16 dst
$REMOTE_IP/16 proto esp reqid 0xb29ed314 mode transport priority 12
ip xfrm policy add src $REMOTE_IP dst $LOCAL_TUN offload packet dev $PF dir in tmpl src $REMOTE_IP/16 dst
$LOCAL_TUN/16 proto esp reqid 0xc35aa26e mode transport priority 12
```

- Configure Openvswitch:

```
ovs-vsctl add-br br-ovs
ovs-vsctl add-port br-ovs $REP
ovs-vsctl add-port br-ovs vxlan1 -- set interface vxlan1 type=vxlan options:local_ip=$LOCAL_TUN
options:remote_ip=$REMOTE_IP options:key=$VXLAN_ID options:dst_port=4789
```

5.6.2.2.19.1 IPsec Full Offload for RDMA Traffic

This IPsec Full Offload for RDMA Traffic option provides a significant performance improvement compared to the software IPsec counterpart, and enables the use of IPsec over RoCE packets, which are outside the network stack and cannot be used without full hardware offload. As a result, users can leverage the benefits of the IPsec protocol with RoCE V2, even when using SR-IOV VFs.

The configuration steps for this feature should be identical to the steps mentioned above, but if this feature is supported, the traffic that will be sent can also be RoCEV2 IPsec traffic.

To configure this feature:

1. Enable IPsec over VF. For more information, please see IPsec Functionality.
2. Configure IPsec policies and states on the relevant VF net device. This should be identical to the software configuration of IPsec rules, which can be done using one of the following implementation options:

Command	Offload Request Parameter
iproute2 ip xfrm	offload packet
libreswan	nic-offload=packet
strongswan	

   We don't have a way to export this macro.
3. Configure an SR-IOV VF normally, and add its OVS/TC rules.

> ⚠ For this feature to work, DMFS steering mode must be enabled.

- The following is a full minimalistic configuration example using iproute, whereas PF0 is the netdevice PF, F0_REP is the VF representor, and NIC is the VF netdevice to configure IPsec over:

```
1.  echo 1 > /sys/class/net/$PF0 /device/sriov_numvfs
2.  echo 0000:08:00.2 > /sys/bus/pci/drivers/mlx5_core/unbind
3.  devlink dev eswitch set pci/0000:08:00.0 mode switchdev
4.  devlink dev param set pci/0000:08:00.0 name flow_steering_mode value dmfs cmode runtime
5.  devlink port function set pci/0000:08:00.0/1 ipsec_packet enable
6.  echo 0000:08:00.2 > /sys/bus/pci/drivers/mlx5_core/bind
7.  tc qdisc add dev $PF0 ingress
tc qdisc add dev $VF0_REP ingress
tc filter add dev $PF0 parent ffff: protocol 802.1q chain 0 flower vlan_id 10 vlan_ethtype 802.1q cvlan_id
5 action vlan pop action vlan pop  action mirred egress redirect dev $VF0_REP

tc filter add dev $VF0_REP parent ffff: protocol all chain 0 flower action vlan push protocol 802.1q id 5
 action vlan push protocol 802.1q id 10 action mirred egress redirect dev $PF0

8.  ifconfig $PF0 $PF_IP/24 up
ifconfig $NIC $LOC_IP/$SUB_NET up
ip link set dev $VF_REP up
9.  ip  xfrm state flush
ip  xfrm policy flush
```

- Configure IPsec states and policies:

```
#states
ip  -4 xfrm state add src $LOC_IP/$SUB_NET dst $REMOTE_IP/$SUB_NET proto esp spi 1000 reqid 10000 aead
'rfc4106(gcm(aes))' 0x010203047aeaca3f87d060a12f4a4487d5a5c335 128 mode transport sel src $LOC_IP dst
$REMOTE_IP offload packet dev $NIC dir out
ip  -4 xfrm state add src $REMOTE_IP/$SUB_NET dst $LOC_IP/$SUB_NET proto esp spi 1001 reqid 10001 aead
'rfc4106(gcm(aes))' 0x010203047aeaca3f87d060a12f4a4487d5a5c335 128 mode transport sel src $REMOTE_IP dst
$LOC_IP offload packet dev $NIC dir in
#policies
ip  -4 xfrm policy add src $LOC_IP dst $REMOTE_IP offload packet dev $NIC dir out tmpl src $LOC_IP/$SUB_NET
dst $REMOTE_IP/$SUB_NET proto esp reqid 10000 mode transport
ip  -4 xfrm policy add src $REMOTE_IP dst $LOC_IP offload packet dev $NIC dir in tmpl src $REMOTE_IP/
$SUB_NET dst $LOC_IP/$SUB_NET proto esp reqid 10001 mode transport
ip  -4 xfrm policy add src $REMOTE_IP dst $LOC_IP dir fwd tmpl src $REMOTE_IP/$SUB_NET dst $LOC_IP/$SUB_NET
proto esp reqid 10001 mode transport
```

Note that the configuration above is for one side only, yet IPsec must be configured for both sides in order for them to communicate properly. The configuration for the other side should be almost identical, but Step 9 would be configured in an asymmetrical way, meaning the first policy would look the following, and all other states/policies would be adjusted accordingly:

```
ip -4 xfrm state add src $LOC_IP/$SUB_NET dst $REMOTE_IP/$SUB_NET proto esp spi 1001 reqid 10001 aead
'rfc4106(gcm(aes))' 0x010203047aeaca3f87d060a12f4a4487d5a5c335 128 mode transport sel src $LOC_IP dst $REMOTE_IP
offload packet dev $NIC dir out
```

Once this step is completed, you can send any RoCE traffic of your choice between the two machines with configured IPsec. For example, **`ibv_rc_pingpong -g 3 -d VF_device :`** on one side, and **`ibv_rc_pingpong -g 3 -d VF_device $IP_OF_OTHER_SIDE :`** on the other side.

Finally, you can verify that the traffic was encrypted using IPsec by using the ipsec counters:

```
ethtool -S VF_NETDEV | grep ipsec
```

5.6.2.2.20 MACsec Full Offload

> ⚠ MACsec full offload is supported at alpha level and is tested only on NVIDIA® ConnectX®-7.

MACsec Full offload feature, also known as MACsec inline Full offload, enables the user to offload MACsec crypto encryption and decryption, MACsec headers encapsulation and decapsulation, and Anti replay operations to the hardware.

Hardware implementation supports GCM-AES & GCM-AES-XPN encryption schemes.

MACsec introduced in MOFED v5.9 requires a minimal Kernel version of 6.1.

To enable the feature, support in both kernel and adapter firmware is required.

For support in the kernel, make sure the following flags are set as follows:

- **CONFIG_MACSEC=y**
- **CONFIG_MLX5_EN_MACSEC=y**

For support in firmware, use version xx.34.0364 and up.

5.6.2.2.20.1 Configurations

IProute2 Configuration

Configuring Physical Interface

- Client side:

```
ip address flush <physical_device>
ip address add <client_physical_device_ip> dev <physical interface>
ip link set dev <physical_device> up
```

- Server side:

```
ip address flush <physical_device>
ip address add <server_physical_device_ip> dev <physical interface>
ip link set dev <physical_device> up
```

Add MACsec Device

- Client side:

```
ip link add link <physical_device> <macsec_device> type macsec sci <client_sci> client on
```

- Server side:

```
ip link add link <physical_device> <macsec_device> type macsec sci <server_sci> client on
```

Offload MACsec Device

- Client side:

```
ip macsec offload <macsec_device> mac
```

- Server side:

```
ip macsec offload <macsec_device> mac
```

Add MACsec rules:

- Client side:

```
ip macsec add <macsec_device> tx sa <sa_num>pn <inital_packet_number>on key <client_key_id> <client_key>
ip macsec add <macsec_device> rx sci <server_sci> on
ip macsec add <macsec_device> rx sci <server_sci>sa <sa_num> pn <inital_packet_number> on key
<server_key_id> <server_key>
```

- Server side:

```
ip macsec add <macsec_device> tx sa <sa_num>pn <inital_packet_number>on key <server_key_id> <server_key>
ip macsec add <macsec_device> rx sci <client_sci> on
ip macsec add <macsec_device> rx sci <client_sci>sa <sa_num> pn <inital_packet_number> on key
<client_key_id> <client_key>
```

Configure MACsec device IPs:

- Client side:

```
ip address flush <macsec_device>
ip address add <client_macsec_device_ip> dev <macsec_device>
ip link set dev <macsec_device> up
```

- Server side:

```
ip address flush <macsec_device>
ip address add <server_macsec_device_ip> dev <macsec_device>
ip link set dev <macsec_device> up
```

Configuration Example

Client side:

```
ip address flush enp8s0f0
ip address add 1.1.1.1/24 dev enp8s0f0
ip link set dev enp8s0f0 up
ip link add link enp8s0f0 macsec0 type macsec sci 1 encrypt on
ip macsec offload macsec0 mac
ip macsec add macsec0 tx sa 0 pn 1 on key 00 dffafc8d7b9a43d5b9a3dfbbf6a30c16
ip macsec add macsec0 rx sci 2 on
ip macsec add macsec0 rx sci 2 sa 0 pn 1 on key 00 ead3664f508eb06c40ac7104cdae4ce5
ip address flush macsec0
ip address add 2.2.2.1/24 dev macsec0
ip link set dev macsec0 up
```

Server side:

```
ip link del macsec0
ip address flush enp8s0f0
ip address add 1.1.1.2/24 dev enp8s0f0
ip link set dev enp8s0f0 up
ip link add link enp8s0f0 macsec0 type macsec sci 2 encrypt on
ip macsec offload macsec0 mac
ip macsec add macsec0 tx sa 0 pn 1 on key 00 ead3664f508eb06c40ac7104cdae4ce5
ip macsec add macsec0 rx sci 1 on
ip macsec add macsec0 rx sci 1 sa 0 pn 1 on key 00 dffafc8d7b9a43d5b9a3dfbbf6a30c16
ip address flush macsec0
ip address add 2.2.2.2/24 dev macsec0
ip link set dev macsec0 up
```

Use **ip macsec show** command to check configuration.

To verify traffic is offloaded, check MACsec counters by running **ethtool -S <physical_device> | grep macsec** .

Refer to the Linux Manual page for more information.

5.6.2.3 RDMA

The chapter contains the following sections:

- RDMA-aware Networks Programming Guide
- RDMA over Converged Ethernet
- Adaptive Retransmission: Parameters Control
- Migration to RDMA-Core
- Congestion Control Infrastructure

5.6.2.3.1 RDMA-aware Networks Programming Guide

5.6.2.3.1.1 1. Introduction

Remote Direct Memory Access (RDMA) provides direct memory access from the memory of one host (storage or compute) to the memory of another host without involving the remote Operating System and CPU, boosting network and host performance with lower latency, lower CPU load and higher bandwidth. In contrast, TCP/IP communications typically require copy operations, which add latency and consume significant CPU and memory resources.

For further information, please refer to RDMA Core GitHub.

5.6.2.3.1.2 2. Glossary

Term	Description
Access Layer	Low level operating system infrastructure (plumbing) used for accessing the interconnect fabric (VPI™, InfiniBand®, Ethernet, FCoE). It includes all basic transport services needed to support upper level network protocols, middleware, and management agents.
AH (Address Handle)	An object which describes the path to the remote side used in UD QP
CA (Channel Adapter)	A device which terminates an InfiniBand link, and executes transport level functions
CI (Channel Interface)	Presentation of the channel to the Verbs Consumer as implemented through the combination of the network adapter, associated firmware, and device driver software
CM (Communication Manager)	An entity responsible to establish, maintain, and release communication for RC and UC QP service types The Service ID Resolution Protocol enables users of UD service to locate QPs supporting their desired service. There is a CM in every IB port of the end nodes.

Term	Description
Compare & Swap	Instructs the remote QP to read a 64-bit value, compare it with the compare data provided, and if equal, replace it with the swap data, provided in the QP.
CQ (Completion Queue)	A queue (FIFO) which contains CQEs
CQE (Completion Queue Entry)	An entry in the CQ that describes the information about the completed WR (status size etc.)
DMA (Direct Memory Access)	Allowing Hardware to move data blocks directly to and from the memory, bypassing the CPU
Fetch & Add	Instructs the remote QP to read a 64-bit value and replace it with the sum of the 64-bit value and the added data value, provided in the QP.
GUID (Globally Unique IDentifier)	A 64 bit number that uniquely identifies a device or component in a subnet
GID (Global IDentifier)	A 128-bit identifier used to identify a Port on a network adapter, a port on a Router, or a Multicast Group. A GID is a valid 128-bit IPv6 address (per RFC 2373) with additional properties / restrictions defined within IBA to facilitate efficient discovery, communication, and routing.
GRH (Global Routing Header)	A packet header used to deliver packets across a subnet boundary and also used to deliver Multicast messagesThis Packet header is based on IPv6 protocol.
Network Adapter	A hardware device that allows for communication between computers in a network.
Host	A computer platform executing an Operating System which may control one or more network adapters
IB	InfiniBand
Join operation	An IB port must explicitly join a multicast group by sending a request to the SA to receive multicast packets.
lkey	A number that is received upon registration of MR is used locally by the WR to identify the memory region and its associated permissions.
LID (Local IDentifier)	A 16 bit address assigned to end nodes by the subnet manager.Each LID is unique within its subnet.
LLE (Low LatencyEthernet)	RDMA service over CEE (Converged Enhanced Ethernet) allowing IB transport over Ethernet.
NA (Network Adapter)	A device which terminates a link and executes transport level functions.
MGID (Multicast Group ID)	IB multicast groups, identified by MGIDs, are managed by the SM. The SM associates a MLID with each MGID and explicitly programs the IB switches in the fabric to ensure that the packets are received by all the ports that joined the multicast group.
MR (Memory Region)	A contiguous set of memory buffers which have already been registered with access permissions. These buffers need to be registered for the network adapter to make use of them. During registration an L_Key and R_Key are created and associated with the created memory region

Term	Description
MTU (MaximumTransfer Unit)	The maximum size of a packet payload (not including headers) that can be sent / received from a port
MW (Memory Window)	An allocated resource that enables remote access after being bound to a specified area within an existing Memory Registration. Each Memory Window has an associated Window Handle, set of access privileges, and current R_Key.
Outstanding Work Request	WR which was posted to a work queue and its completion was not polled
pkey (Partition key)	The pkey identifies a partition that the port belongs to. A pkey is roughly analogous to a VLAN ID in ethernet networking. It is used to point to an entry within the port's partition key (pkey) table. Each port is assigned at least one pkey by the subnet manager (SM).
PD (Protection Domain)	Object whose components can interact with only each other.AHs interact with QPs, and MRs interact with WQs.
QP (Queue Pair)	The pair (send queue and receive queue) of independent WQs packed together in one object for the purpose of transferring data between nodes of a network. Posts are used to initiate the sending or receiving of data. There are three types of QP: UD Unreliable Datagram, Unreliable Connection, and Reliable Connection.
RC (Reliable Connection)	A QP Transport service type based on a connection oriented protocol.A QP (Queue pair) is associated with another single QP. The messages are sent in a reliable way (in terms of the correctness and order of the information.)
RDMA (Remote Direct Memory Access)	Accessing memory in a remote side without involvement of the remote CPU
RDMA_CM (Remote Direct Memory Access Communication Manager)	API used to setup reliable, connected and unreliable datagram data transfers. It provides an RDMA transport neutral interface for establishing connections. The API is based on sockets, but adapted for queue pair (QP) based semantics: communication must be over a specific RDMA device, and data transfers are message based.
Requestor	The side of the connection that will initiate a data transfer (by posting a send request)
Responder	The side of the connection that will respond to commands from the requestor which may include a request to write to the responder memory or read from the responder memory and finally a command requesting the responder to receive a message.
rkey	A number that is received upon registration of MR is used to enforce permissions on incoming RDMA operations
RNR (Receiver Not Ready)	The flow in an RC QP where there is a connection between the sides but a RR is not present in the Receive side
RQ (Receive Queue)	A Work Queue which holds RRs posted by the user
RR (Receive Request)	A WR which was posted to an RQ which describes where incoming data using a send opcode is going to be written. Also note that a RDMA Write with immediate will consume a RR.
RTR (Ready To Receive)	A QP state in which an RR can be posted and be processed
RTS (Ready To Send)	A QP state in which an SR can be posted and be processed

Term	Description
SA (SubnetAdministrator)	The interface for querying and manipulating subnet management data
SGE (Scatter / GatherElements)	An entry to a pointer to a full or a part of a local registered memory block.The element hold the start address of the block, size, and lkey (with its associated permissions).
S/G Array	An array of S/G elements which exists in a WR that according to the used opcode either collects data from multiple buffers and sends them as a single stream or takes a single stream and breaks it down to numerous buffers
SM (Subnet Manager)	An entity that configures and manages the subnet Discovers the network topology Assign LIDs Determines the routing schemes and sets the routing tables One master SM and possible several slaves (Standby mode) Administers switch routing tables thereby establishing paths through the fabric
SQ (Send Queue)	A Work Queue which holds SRs posted by the user
SR (Send Request)	A WR which was posted to an SQ which describes how much data is going to be transferred, its direction, and the way (the opcode will specify the transfer)
SRQ (Shared Receive Queue)	A queue which holds WQEs for incoming messages from any RC/UC/UD QP which is associated with it.More than one QPs can be associated with one SRQ.
TCA (Target Channel Adapter)	A Channel Adapter that is not required to support verbs, usually used in I/O devices
UC (UnreliableConnection)	A QP transport service type based on a connection oriented protocol,where a QP (Queue pair) is associated with another single QP. The QPs do not execute a reliable Protocol and messages can be lost.
UD (UnreliableDatagram)	A QP transport service type in which messages can be one packet length and every UD QP can send/receive messages from another UD QP in the subnetMessages can be lost and the order is not guaranteed. UD QP is the only type which supports multicast messages. The message size of a UD packet is limited to the Path MTU
Verbs	An abstract description of the functionality of a network adapter.Using the verbs, any application can create / manage objects that are needed in order to use RDMA for data transfer.
VPI (Virtual Protocol Interface)	Allows the user to change the layer 2 protocol of the port.
WQ (Work Queue)	One of Send Queue or Receive Queue.
WQE (Work QueueElement)	A WQE, pronounced "wookie", is an element in a work queue.
WR (Work Request)	A request which was posted by a user to a work queue.

5.6.2.3.1.3 3. RDMA Architecture

3.1. InfiniBand

InfiniBand (IB) is a high-speed, low latency, low CPU overhead, highly efficient and scalable server and storage interconnect technology. One of the key capabilities of InfiniBand is its support for native Remote Direct Memory Access (RDMA). InfiniBand enables data transfer between servers and between server and storage without the involvement of the host CPU in the data path. InfiniBand uses I/O channels for data communication (up to 16 million per host), where each channel provides the semantics of a virtualized NIC or HCA (security, isolations, etc). InfiniBand provides various technology or solution speeds ranging from 10Gb/s (SDR) up to 56Gb/s (FDR) per port, using copper and optical fiber connections. InfiniBand efficiency and scalability have made it the optimal performance and cost/performance interconnect solution for the world's leading high-performance computing, cloud, Web 2.0, storage, database and financial data centers and applications. InfiniBand is a standard technology, defined and specified by the IBTA organization.

3.2. Virtual Protocol Interconnect® (VPI)

Virtual Protocol Interconnect (VPI) architecture provides a high performance, low latency and reliable means for communication among network adapters and switches supporting both InfiniBand and Ethernet semantics. A VPI adapter or switch can be set to deliver either InfiniBand or Ethernet semantics per port. A dual-port VPI adapter, for example, can be configured to one of the following options:

- An adapter (HCA) with two InfiniBand ports
- A NIC with two Ethernet ports
- An adapter with one InfiniBand port and one Ethernet port at the same time

Similarly, a VPI switch can have InfiniBand-only ports, Ethernet-only ports, or a mix of both InfiniBand and Ethernet ports working at the same time. VPI adapters and switches support both the InfiniBand RDMA and the Ethernet RoCE solutions.

3.3. RDMA over Converged Ethernet (RoCE)

RoCE is a standard for RDMA over Ethernet that is also defined and specified by the IBTA organization. RoCE provides true RDMA semantics for Ethernet as it does not require the complex and low performance TCP transport (needed for iWARP, for example).RoCE is the most efficient low latency Ethernet solution today. It requires a very low CPU overhead and takes advantage of Priority Flow Control in Data Center Bridging Ethernet for lossless connectivity. RoCE has been fully supported by the Open Fabrics Software since the release of OFED 1.5.1.

3.4. Comparison of RDMA Technologies

Currently, there are three technologies that support RDMA: InfiniBand, Ethernet RoCE and Ethernet iWARP. All three technologies share a common user API which is defined in this document, but have different physical and link layers.

When it comes to the Ethernet solutions, RoCE has clear performance advantages over iWARP—both for latency, throughput and CPU overhead. RoCE is supported by many leading solutions, and is incorporated within Windows Server software (as well as InfiniBand).

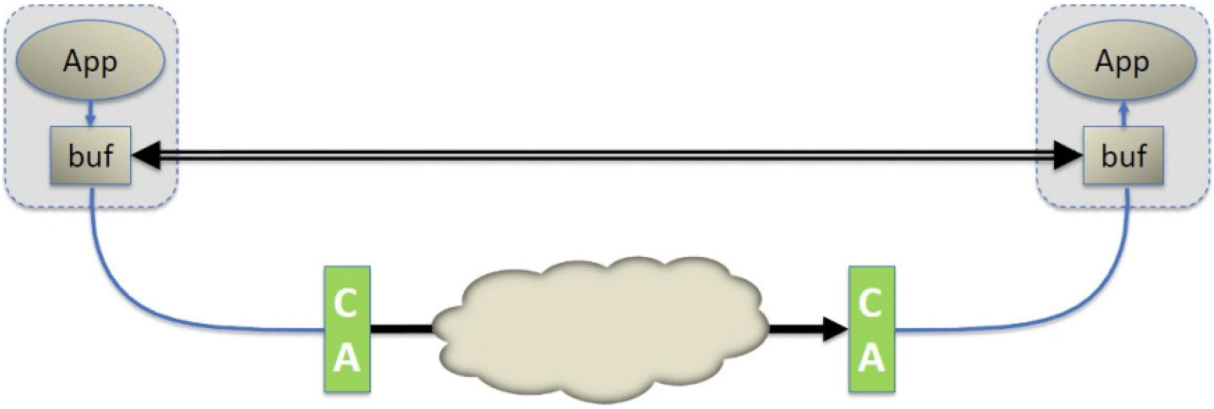
RDMA technologies are based on networking concepts found in a traditional network but there are differences them and their counterparts in IP networks. The key difference is that RDMA provides a messaging service which applications can use to directly access the virtual memory on remote computers. The messaging service can be used for Inter Process Communication (IPC), communication with remote servers and to communicate with storage devices using Upper Layer Protocols (ULPs) such as iSCSI Extensions for RDMA (ISER) and SCSI RDMA Protocol (SRP), Storage Message Block (SMB), Samba, Lustre, ZFS and many more.

RDMA provides low latency through stack bypass and copy avoidance, reduces CPU utilization, reduces memory bandwidth bottlenecks and provides high bandwidth utilization. The key benefits that RDMA delivers accrue from the way that the RDMA messaging service is presented to the application and the underlying technologies used to transport and deliver those messages. RDMA provides Channel based IO. This channel allows an application using an RDMA device to directly read and write remote virtual memory.

In traditional sockets networks, applications request network resources from the operating system through an API which conducts the transaction on their behalf. However RDMA use the OS to establish a channel and then allows applications to directly exchange messages without further OS intervention. A message can be an RDMA Read, an RDMA Write operation or a Send/ Receive operation. InfiniBand and RoCE also support Multicast transmission.

The InfiniBand Link layer offers features such as a credit based flow control mechanism for congestion control. It also allows the use of Virtual Lanes (VLs) which allow simplification of the higher layer level protocols and advanced Quality of Service. It guarantees strong ordering within the VL along a given path. The InfiniBand Transport layer provides reliability and delivery guarantees.

The Network Layer used by InfiniBand has features which make it simple to transport messages directly between applications' virtual memory even if the applications are physically located on different servers. Thus the combination of InfiniBand Transport layer with the Software Transport Interface is better thought of as a RDMA message transport service. The entire stack, including the Software Transport Interface comprises the InfiniBand messaging service.



The most important point is that every application has direct access to the virtual memory of devices in the fabric. This means that applications do not need to make requests to an operating system to transfer messages. Contrast this with the traditional network environment where the shared network resources are owned by the operating system and cannot be accessed by a user application. Thus, an application must rely on the involvement of the operating system to move data from the application's virtual buffer space, through the network stack and out onto the wire. Similarly, at the other end, an application must rely on the operating system to retrieve the data on the wire on its behalf and place it in its virtual buffer space.

TCP/IP/Ethernet is a byte-stream oriented transport for passing bytes of information between sockets applications. TCP/IP is lossy by design but implements a reliability scheme using the Transmission Control Protocol (TCP). TCP/IP requires Operating System (OS) intervention for every operation which includes buffer copying on both ends of the wire. In a byte stream-oriented network, the idea of a message boundary is lost. When an application wants to send a packet, the OS places the bytes into an anonymous buffer in main memory belonging to the operating system and when the byte transfer is complete, the OS copies the data in its buffer into the receive buffer of the application. This process is repeated each time a packet arrives until the entire byte stream is received. TCP is responsible for retransmitting any lost packets due to congestion.

In InfiniBand, a complete message is delivered directly to an application. Once an application has requested transport of an RDMA Read or Write, the InfiniBand hardware segments the outbound message as needed into packets whose size is determined by the fabric path maximum transfer unit. These packets are transmitted through the InfiniBand network and delivered directly into the receiving application's virtual buffer where they are re-assembled into a complete message. The receiving application is notified once the entire message has been received. Thus neither the sending nor the receiving application is involved until the entire message is delivered into the receiving application's buffer.

3.5. Key Components

These are being presented only in the context of the advantages of deploying IB and RoCE. We do not discuss cables and connectors.

3.5.1. Host Channel Adapter

HCAs provide the point at which an IB end node (for example, a server) connects to an IB network. These are the equivalent of the Ethernet (NIC) card but they do much more. HCAs provide address translation mechanism under the control of the operating system which allows an application to access the HCA directly. The same address translation mechanism is the means by which an HCA accesses memory on behalf of a user level application. The application refers to virtual addresses while the HCA has the ability to translate these addresses into physical addresses in order to affect the actual message transfer.

3.5.2. Range Extenders

InfiniBand range extension is accomplished by encapsulating the InfiniBand traffic onto the WAN link and extending sufficient buffer credits to ensure full bandwidth across the WAN.

3.5.3. Subnet Manager

The InfiniBand subnet manager assigns Local Identifiers (LIDs) to each port connected to the InfiniBand fabric and develops a routing table based on the assigned LIDs. The IB Subnet Manager is a concept of Software Defined Networking (SDN) which eliminates the interconnect complexity and enables the creation of very large scale compute and storage infrastructures.

3.5.4. Switches

IB switches are conceptually similar to standard networking switches but are designed to meet IB performance requirements. They implement flow control of the IB Link Layer to prevent packet dropping, and to support congestion avoidance and adaptive routing capabilities, and advanced Quality of Service. Many switches include a Subnet Manager. At least one Subnet Manager is required to configure an IB fabric.

3.6. Support for Existing Applications and ULPs

IP applications are enabled to run over an InfiniBand fabric using IP over IB (IPoIB) or Ethernet over IB (EoIB) or RDS ULPs. Storage applications are supported via iSER, SRP, RDS, NFS, ZFS, SMB and others. MPI and Network Direct are all supported ULPs as well, but are outside the scope of this document.

5.6.2.3.1.4 4. RDMA-Aware Programming

The VPI architecture permits direct user mode access to the hardware. Mellanox provides a dynamically loaded library, creating access to the hardware via the verbs API. This document contains verbs and their related inputs, outputs, descriptions, and functionality as exposed through the operating system programming interface.

> ⚠ This section and its verbs are valid only for user space. See header files for the kernel space verbs.

Programming with verbs allows for customizing and optimizing the RDMA-Aware network. This customizing and optimizing should be done only by programmers with advanced knowledge and experience in the VPI systems.

To perform RDMA operations, establishment of a connection to the remote host, as well as appropriate permissions need to be set up first. The mechanism for accomplishing this is the Queue Pair (QP). For those familiar with a standard IP stack, a QP is roughly equivalent to a socket. The QP needs to be initialized on both sides of the connection. Communication Manager (CM) can be used to exchange information about the QP prior to actual QP setup.

Once a QP is established, the verbs API can be used to perform RDMA reads, RDMA writes, and atomic operations. Serialized send/receive operations, which are similar to socket reads/writes, can be performed as well.

4.1. Available Communication Operations

4.1.1. Send/Send With Immediate

The send operation allows you to send data to a remote QP's receive queue. The receiver must have previously posted a receive buffer to receive the data. The sender does not have any control over where the data will reside in the remote host. Optionally, an immediate 4 byte value may be transmitted with the data buffer. This immediate value is presented to the receiver as part of the receive notification, and is not contained in the data buffer.

4.1.2. Receive

This is the corresponding operation to a send operation. The receiving host is notified that a data buffer has been received, possibly with an inline immediate value. The receiving application is responsible for receive buffer maintenance and posting.

4.1.3. RDMA Read

A section of memory is read from the remote host. The caller specifies the remote virtual address as well as a local memory address to be copied to. Prior to performing RDMA operations, the remote host must provide appropriate permissions to access its memory. Once these permissions are set, RDMA read operations are conducted with no notification whatsoever to the remote host. For both RDMA read and write, the remote side isn't aware that this operation being done (other than the preparation of the permissions and resources).

4.1.4. RDMA Write/RDMA Write With Immediate

Similar to RDMA read, but the data is written to the remote host. RDMA write operations are performed with no notification to the remote host. RDMA write with immediate operations, however, do notify the remote host of the immediate value.

4.1.5. Atomic Fetch and Add/Atomic Compare and Swap

These are atomic extensions to the RDMA operations. The atomic fetch and add operation atomically increments the value at a specified virtual address by a specified amount. The value prior to being incremented is returned to the caller. The atomic compare and swap will atomically compare the value at a specified virtual address with a specified value and if they are equal, a specified value will be stored at the address.

4.2. Transport Modes

There are several different transport modes you may select from when establishing a QP. Operations available in each mode are shown in the following table.

Operation	UD	UC	RC	RD
Send (with immediate)	+	+	+	+
Receive	+	+	+	+
RDMA Write (with immediate)		+	+	+
RDMA Read			+	+
Atomic: Fetch and Add/ Cmp and Swap			+	+
Max message size	MTU	1GB	1GB	1GB

RD is not supported by this API.

4.2.1. Reliable Connection (RC)

Queue Pair is associated with only one other QP. Messages transmitted by the send queue of one QP are reliably delivered to receive queue of the other QP. Packets are delivered in order. A RC connection is very similar to a TCP connection.

4.2.2. Unreliable Connection (UC)

A Queue Pair is associated with only one other QP. The connection is not reliable so packets may be lost. Messages with errors are not retried by the transport, and error handling must be provided by a higher level protocol.

4.2.3. Unreliable Datagram (UD)

A Queue Pair may transmit and receive single-packet messages to/from any other UD QP. Ordering and delivery are not guaranteed, and delivered packets may be dropped by the receiver. Multicast messages are supported (one to many). A UD connection is very similar to a UDP connection.

4.3. Key Concepts

4.3.1. Send Request (SR)

An SR defines how much data will be sent, from where, how and, with RDMA, to where. struct **ibv_send_wr** is used to implement SRs.

4.3.2. Receive Request (RR)

An RR defines buffers where data is to be received for non-RDMA operations. If no buffers are defined and a transmitter attempts a send operation or a RDMA Write with immediate, a receive not ready (RNR) error will be sent. struct **ibv_recv_wr** is used to implement RRs.

4.3.3. Completion Queue

A Completion Queue is an object which contains the completed work requests which were posted to the Work Queues (WQ). Every completion says that a specific WR was completed (both successfully completed WRs and unsuccessfully completed WRs).A Completion Queue is a mechanism to notify the application about information of ended Work Requests (status, opcode, size, source). CQs have n Completion Queue Entries (CQE). The number of CQEs is specified when the CQ is created. When a CQE is polled it is removed from the CQ. CQ is a FIFO of CQEs. CQ can service send queues, receive queues, or both. Work queues from multiple QPs can be associated with a single CQ. struct **ibv_cq** is used to implement a CQ.

4.3.4. Memory Registration

Memory Registration is a mechanism that allows an application to describe a set of virtually contiguous memory locations or a set of physically contiguous memory locations to the network adapter as a virtually contiguous buffer using Virtual Addresses.

The registration process pins the memory pages (to prevent the pages from being swapped out and to keep physical <-> virtual mapping). During the registration, the OS checks the permissions of the registered block. The registration process writes the virtual to physical address table to the network adapter. When registering memory, permissions are set for the region. Permissions are local write, remote read, remote write, atomic, and bind. Every MR has a remote and a local key (**r_key** , **l_key**). Local keys are used by the local HCA to access local memory, such as during a receive operation. Remote keys are given to the remote HCA to allow a remote process access to system memory during RDMA operations. The same memory buffer can be registered several times (even with different access permissions) and every registration results in a different set of keys.

Struct **ibv_mr** is used to implement memory registration.

4.3.5. Memory Window

An MW allows the application to have more flexible control over remote access to its memory. Memory Windows are intended for situations where the application:

- wants to grant and revoke remote access rights to a registered Region in a dynamic fashion with less of a performance penalty than using deregistration/registration or reregistration.
- wants to grant different remote access rights to different remote agents and/or grant those rights over different ranges within a registered Region.

The operation of associating an MW with an MR is called Binding. Different MWs can overlap the same MR (event with different access permissions).

4.3.6. Address Vector

An Address Vector is an object that describes the route from the local node to the remote node. In every UC/RC QP there is an address vector in the QP context. In UD QP the address vector should be defined in every post SR. struct `ibv_ah` is used to implement address vectors.

4.3.7. Global Routing Header (GRH)

The GRH is used for routing between subnets. When using RoCE, the GRH is used for routing inside the subnet and therefore is a mandatory. The use of the GRH is mandatory in order for an application to support both IB and RoCE.

When global routing is used on UD QPs, there will be a GRH contained in the first 40 bytes of the receive buffer. This area is used to store global routing information, so an appropriate address vector can be generated to respond to the received packet. If GRH is used with UD, the RR should always have extra 40 bytes available for this GRH. struct `ibv_grh` is used to implement GRHs.

4.3.8. Protection Domain

Object whose components can interact with only each other. These components can be AH, QP, MR, and SRQ. A protection domain is used to associate Queue Pairs with Memory Regions and Memory Windows, as a means for enabling and controlling network adapter access to Host System memory. PDs are also used to associate Unreliable Datagram queue pairs with Address Handles, as a means of controlling access to UD destinations. struct `ibv_pd` is used to implement protection domains.

4.3.9. Asynchronous Events

The network adapter may send async events to inform the SW about events that occurred in the system.

There are two types of async events:

- Affiliated events: events that occurred to personal objects (CQ, QP, SRQ). Those events will be sent to a specific process.
- Unaffiliated events: events that occurred to global objects (network adapter, port error). Those events will be sent to all processes.

4.3.10. Scatter Gather

Data is being gathered/scattered using scatter gather elements, which include:

- Address: address of the local data buffer that the data will be gathered from or scattered to. Size: the size of the data that will be read from / written to this address.

- L_key: the local key of the MR that was registered to this buffer. struct **ibv_sge** implements scatter gather elements.

4.3.11. Polling

Polling the CQ for completion is getting the details about a WR (Send or Receive) that was posted. If we have completion with bad status in a WR, the rest of the completions will be all be bad (and the Work Queue will be moved to error state). Every WR that does not have a completion (that was polled) is still outstanding. Only after a WR has a completion, the send / receive buffer may be used/reused/freed. The completion status should always be checked. When a CQE is polled it is removed from the CQ. Polling is accomplished with the **ibv_poll_cq** operation.

4.4. Typical Application

This documents provides two program examples:

- The first code, **RDMA_RC_example**, uses the VPI verbs API, demonstrating how to perform RC: Send, Receive, RDMA Read and RDMA Write operations.
- The second code, multicast example, uses **RDMA_CM** verbs API, demonstrating Multicast UD.

The structure of a typical application is as follows:

1. Get the device list (implemented in programming example by **resources_create**):
   First you must retrieve the list of available IB devices on the local host. Every device in this list contains both a name and a GUID. For example the device names can be: mthca0, mlx4_1.
2. Open the requested device (implemented in programming example by **resources_create**):
   Iterate over the device list, choose a device according to its GUID or name and open it.
3. Query the device capabilities (implemented in programming example by **resources_create**):
   The device capabilities allow the user to understand the supported features (APM, SRQ) and capabilities of the opened device.
4. Allocate a Protection Domain to contain your resources (implemented in programming example by **resources_create**):
   A Protection Domain (PD) allows the user to restrict which components can interact with only each other. These components can be AH, QP, MR, MW, and SRQ.
5. Register a memory region (implemented in programming example by **resources_create**):
   VPI only works with registered memory. Any memory buffer which is valid in the process's virtual space can be registered. During the registration process the user sets memory permissions and receives local and remote keys (lkey/rkey) which will later be used to refer to this memory buffer.
6. Create a Completion Queue (CQ) (implemented in programming example by **resources_create**):
   A CQ contains completed work requests (WR). Each WR will generate a completion queue entry (CQE) that is placed on the CQ. The CQE will specify if the WR was completed successfully or not.
7. Create a Queue Pair (QP) (implemented in programming example by **resources_create**):
   Creating a QP will also create an associated send queue and receive queue.

8. Bring up a QP (implemented in programming example by `connect_qp`, `modify_qp_to_init`, `post_receive`, `modify_qp_to_rtr`, and `modify_qp_to_rts`):
   A created QP still cannot be used until it is transitioned through several states, eventually getting to Ready To Send (RTS). This provides needed information used by the QP to be able send/receive data.
9. Post work requests and poll for completion (implemented in programming example by `post_send` and `poll_completion`):
   Use the created QP for communication operations.
10. Cleanup (implemented in programming example by `resources_destroy`):
    Destroy objects in the reverse order you created them:
    a. Delete QP.
    b. Delete CQ.
    c. Deregister MR.
    d. Deallocate PD.
    e. Close device.

5.6.2.3.1.5 5. VPI Verbs API

This chapter describes the details of the VPI verbs API.

5.1. Initialization

- Template: `int ibv_fork_init(void)`
- Input parameters: None
- Output parameters: None
- Return value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

`ibv_fork_init` initializes libibverbs' data structures to handle the `fork()` function safely and avoid data corruption, whether `fork()` is called explicitly or implicitly such as in `system()` calls.

It is not necessary to call `ibv_fork_init` if all parent process threads are always blocked until all child processes end or change address space via an exec() operation.

This function works on Linux kernels supporting the `MADV_DONTFORK` flag for `madvise()` (2.6.17 and higher).

Setting the environment variable `RDMAV_FORK_SAFE` or `IBV_FORK_SAFE` to any value has the same effect as calling `ibv_fork_init()`.

Setting the environment variable `RDMAV_HUGEPAGES_SAFE` to any value tells the library to check the underlying page size used by the kernel for memory regions. This is required if an application uses huge pages either directly or indirectly via a library such as libhugetlbfs.

Calling `ibv_fork_init()` will reduce performance due to an extra system call for every memory registration, and the additional memory allocated to track memory regions. The precise performance impact depends on the workload and usually will not be significant.

Setting `RDMAV_HUGEPAGES_SAFE` adds further overhead to all memory registrations.

5.2. Verb Context Operations

The following commands are used once a device has been opened. These commands allow you to get more specific information about a device or one of its ports, create completion queues (CQ), completion channels (CC), and protection domains (PD) which can be used for further operations.

5.2.1. ibv_query_device

- Template: **int ibv_query_device(struct ibv_context *context, struct ibv_device_attr *device_attr)**
- Input Parameters:
  - **context** struct **ibv_context** from **ibv_open_device**
- Output Parameters: **struct ibv_device_attr** containing device attributes
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_query_device retrieves the various attributes associated with a device. The user should malloc a struct ibv_device_attr, pass it to the command, and it will be filled in upon successful return. The user is responsible to free this struct.

struct **ibv_device_attr** is defined as follows:

```
struct ibv_device_attr

{
       char                                    fw_ver[64];
       uint64_t                                node_guid;
       uint64_t                                sys_image_guid;
       uint64_t                                max_mr_size;
       uint64_t                                page_size_cap;
       uint32_t                                vendor_id;
       uint32_t                                vendor_part_id;
       uint32_t                                hw_ver;
       int                                     max_qp;
       int                                     max_qp_wr;
       int                                     device_cap_flags;
       int                                     max_sge;
       int                                     max_sge_rd;
       int                                     max_cq;
       int                                     max_cqe;
       int                                     max_mr;
       int                                     max_pd;
       int                                     max_qp_rd_atom;
       int                                     max_ee_rd_atom;
       int                                     max_res_rd_atom;
       int                                     max_qp_init_rd_atom;
       int                                     max_ee_init_rd_atom;
       enum ibv_atomic_cap                     atomic_cap;
       int                                     max_ee;
       int                                     max_rdd;
       int                                     max_mw;
       int                                     max_raw_ipv6_qp;
       int                                     max_raw_ethy_qp;
       int                                     max_mcast_grp;
       int                                     max_mcast_qp_attach;
       int                                     max_total_mcast_qp_attach;
       int                                     max_ah;
       int                                     max_fmr;
       int                                     max_map_per_fmr;
       int                                     max_srq;
       int                                     max_srq_wr;
       int                                     max_srq_sge;
       uint16_t                                max_pkeys;
       uint8_t                                 local_ca_ack_delay;
       uint8_t                                 phys_port_cnt;
}

fw_ver              Firmware version
node_guid           Node global unique identifier (GUID)
sys_image_guid      System image GUID
max_mr_size         Largest contiguous block that can be registered
page_size_cap       Supported page sizes
vendor_id           Vendor ID, per IEEE
vendor_part_id      Vendor supplied part ID
hw_ver              Hardware version
max_qp              Maximum number of Queue Pairs (QP)
max_qp_wr           Maximum outstanding work requests (WR) on any queue
device_cap_flags    IBV_DEVICE_RESIZE_MAX_WR
                     IBV_DEVICE_BAD_PKEY_CNTR
                     IBV_DEVICE_BAD_QKEY_CNTR
                     IBV_DEVICE_RAW_MULTI
```

```
                      IBV_DEVICE_AUTO_PATH_MIG
                      IBV_DEVICE_CHANGE_PHY_PORT
                      IBV_DEVICE_UD_AV_PORT_ENFORCE
                      IBV_DEVICE_CURR_QP_STATE_MOD
                      IBV_DEVICE_SHUTDOWN_PORT
                      IBV_DEVICE_INIT_TYPE
                      IBV_DEVICE_PORT_ACTIVE_EVENT
                      IBV_DEVICE_SYS_IMAGE_GUID
                      IBV_DEVICE_RC_RNR_NAK_GEN
                      IBV_DEVICE_SRQ_RESIZE
                      IBV_DEVICE_N_NOTIFY_CQ
                      IBV_DEVICE_XRC
max_sge               Maximum scatter/gather entries (SGE) per WR for non-RD QPs
max_sge_rd            Maximum SGEs per WR for RD QPs
max_cq                Maximum supported completion queues (CQ)
max_cqe               Maximum completion queue entries (CQE) per CQ
max_mr                Maximum supported memory regions (MR)
max_pd                Maximum supported protection domains (PD)
max_qp_rd_atom        Maximum outstanding RDMA read and atomic operations per QP
max_ee_rd_atom        Maximum outstanding RDMA read and atomic operations per End to End (EE) context (RD
connections)
max_res_rd_atom       Maximum resources used for incoming RDMA read and atomic operations
max_qp_init_rd_atom Maximium RDMA read and atomic operations that may be initiated per QP
max_ee_init_atom      Maximum RDMA read and atomic operations that may be initiated per EE
atomic_cap            IBV_ATOMIC_NONE - no atomic guarantees
                       IBV_ATOMIC_HCA - atomic guarantees within this device
                       IBV_ATOMIC_GLOB - global atomic guarantees
max_ee                Maximum supported EE contexts
max_rdd               Maximum supported RD domains
max_mw                Maximum supported memory windows (MW)
max_raw_ipv6_qp       Maximum supported raw IPv6 datagram QPs
max_raw_ethy_qp       Maximum supported ethertype datagram QPs
max_mcast_grp         Maximum supported multicast groups
max_mcast_qp_attach Maximum QPs per multicast group that can be attached

max_total_mcast_qp_attach
                      Maximum total QPs that can be attached to multicast groups
max_ah                Maximum supported address handles (AH)
max_fmr               Maximum supported fast memory regions (FMR)
max_map_per_fmr       Maximum number of remaps per FMR before an unmap operation is required
max_srq               Maximum supported shared receive queues (SRCQ)
max_srq_wr            Maximum work requests (WR) per SRQ
max_srq_sge           Maximum SGEs per SRQ
max_pkeys             Maximum number of partitions
local_ca_ack_delay    Local CA ack delay
phys_port_cnt         Number of physical ports
```

5.2.2. ibv_query_port

- Template: **int ibv_query_port(struct ibv_context *context, uint8_t port_num, struct ibv_port_attr *port_attr)**
- Input Parameters:
  - **context** struct **ibv_context** from **ibv_open_device** **port_num** physical port number (1 is first port).
- Output Parameters: **port_attr** struct **ibv_port_attr** containing port attributes.
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_query_port retrieves the various attributes associated with a port. The user should allocate a struct **ibv_port_attr**, pass it to the command, and it will be filled in upon successful return. The user is responsible to free this struct.

struct **ibv_port_attr** is defined as follows:

```
struct ibv_port_attr
{
    enum ibv_port_state state;
    enum ibv_mtu    max_mtu;
    enum ibv_mtu    active_mtu;
    int gid_tbl_len;
    uint32_t    port_cap_flags;
    uint32_t    max_msg_sz;
    uint32_t    bad_pkey_cntr;
    uint32_t    qkey_viol_cntr;
    uint16_t    pkey_tbl_len;
    uint16_t    lid;
    uint16_t    sm_lid;
    uint8_t lmc;
    uint8_t max_vl_num;
    uint8_t sm_sl;
```

```
    uint8_t subnet_timeout;
    uint8_t init_type_reply;
    uint8_t active_width;
    uint8_t active_speed;
    uint8_t phys_state;
};

state   IBV_PORT_NOP
    IBV_PORT_DOWN
    IBV_PORT_INIT
    IBV_PORT_ARMED
    IBV_PORT_ACTIVE
    IBV_PORT_ACTIVE_DEFER
max_mtu Maximum Transmission Unit (MTU) supported by port. Can be:
    IBV_MTU_256
    IBV_MTU_512
    IBV_MTU_1024
    IBV_MTU_2048
    IBV_MTU_4096
active_mtu  Actual MTU in use
gid_tbl_len Length of source global ID (GID) table
port_cap_flags  Supported capabilities of this port. There are currently no enumerations/defines declared in
verbs.h
max_msg_sz  Maximum message size
bad_pkey_cntr   Bad P_Key counter
qkey_viol_cntr  Q_Key violation counter
pkey_tbl_len    Length of partition table
lid First local identifier (LID) assigned to this port
sm_lid  LID of subnet manager (SM)
lmc LID Mask control (used when multiple LIDs are assigned to port)
max_vl_num  Maximum virtual lanes (VL)
sm_sl   SM service level (SL)
subnet_timeout  Subnet propagation delay
init_type_reply Type of initialization performed by SM
active_width    Currently active link width
active_speed    Currently active link speed
phys_state  Physical port state
```

5.2.3. ibv_query_gid

- Template: **int ibv_query_gid(struct ibv_context *context, uint8_t port_num, int index, union ibv_gid *gid)**
- Input Parameters:
  - **context** struct **ibv_context** from **ibv_open_device**
  - **port_num** physical port number (1 is first port)
  - **index** which entry in the GID table to return (0 is first)
- Output Parameters:
  - **gid** union **ibv_gid** containing gid information
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_query_gid retrieves an entry in the port's global identifier (GID) table. Each port is assigned at least one GID by the subnet manager (SM). The GID is a valid IPv6 address composed of the globally unique identifier (GUID) and a prefix assigned by the SM. **GID[0]** is unique and contains the port's GUID.

The user should allocate a union **ibv_gid**, pass it to the command, and it will be filled in upon successful return. The user is responsible to free this union.

Union **ibv_gid** is defined as follows:

```
union ibv_gid
{
       uint8_t                                 raw[16];
       struct
       {
           uint64_t                            subnet_prefix;
           uint64_t                            interface_id;
       } global;
};
```

5.2.4. ibv_query_pkey

- Template: `int ibv_query_pkey(struct ibv_context *context, uint8_t port_num, int index, uint16_t *pkey)`
- Input Parameters:
  - `context` struct `ibv_context` from `ibv_open_device`
  - `port_num` physical port number (1 is first port)
  - `index` which entry in the pkey table to return (0 is first)
- Output Parameters:
  - `pkey` desired pkey
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

`ibv_query_pkey` retrieves an entry in the port's partition key (pkey) table. Each port is assigned at least one pkey by the subnet manager (SM). The pkey identifies a partition that the port belongs to. A pkey is roughly analogous to a VLAN ID in Ethernet networking.The user passes in a pointer to a uint16 that will be filled in with the requested pkey. The user is responsible to free this uint16.

5.2.5. ibv_alloc_pd

- Template: `struct ibv_pd {}ibv_alloc_pd{*}(struct ibv_context *context)`
- Input Parameters:
  - `context` struct `ibv_context` from `ibv_open_device`
- Output Parameters: None
- Return Value: Pointer to created protection domain or NULL on failure.

Description:

`ibv_alloc_pd` creates a protection domain (PD). PDs limit which memory regions can be accessed by which queue pairs (QP) providing a degree of protection from unauthorized access. The user must create at least one PD to use VPI verbs.

5.2.6. ibv_dealloc_pd

- Template: `int ibv_dealloc_pd(struct ibv_pd *pd)`
- Input Parameters:
  - `pd` struct `ibv_pd` from `ibv_alloc_pd`
- Output Parameters: none
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

`ibv_dealloc_pd` frees a protection domain (PD). This command will fail if any other objects are currently associated with the indicated PD.

5.2.7. ibv_create_cq

- Template: `struct ibv_cq {}ibv_create_cq{*}(struct ibv_context *context, int cqe, void *cq_context, struct ibv_comp_channel *channel, int comp_vector)`
- Input Parameters:
  - `context` – struct `ibv_context` from `ibv_open_device`
  - `cqe` – Minimum number of entries CQ will support
  - `cq_context` – (Optional) User defined value returned with completion events
  - `channel` – (Optional) Completion channel
  - `comp_vector` – (Optional) Completion vector
- Output Parameters: none
- Return Value: pointer to created CQ or NULL on failure.

Description:

`ibv_create_cq` creates a completion queue (CQ). A completion queue holds completion queue entries (CQE). Each Queue Pair (QP) has an associated send and receive CQ. A single CQ can be shared for sending and receiving as well as be shared across multiple QPs.

The parameter cqe defines the minimum size of the queue. The actual size of the queue may be larger than the specified value.

The parameter `cq_context` is a user defined value. If specified during CQ creation, this value will be returned as a parameter in `ibv_get_cq_event` when using a completion channel (CC).

The parameter channel is used to specify a CC. A CQ is merely a queue that does not have a built in notification mechanism. When using a polling paradigm for CQ processing, a CC is unnecessary. The user simply polls the CQ at regular intervals. If, however, you wish to use a pend paradigm, a CC is required. The CC is the mechanism that allows the user to be notified that a new CQE is on the CQ.

The parameter `comp_vector` is used to specify the completion vector used to signal completion events. It must be >=0 and < context→num_comp_vectors.

5.2.8. ibv_resize_cq

- Template: `int ibv_resize_cq(struct ibv_cq *cq, int cqe)`
- Input Parameters:
  - `cq` CQ to resize cqe Minimum number of entries CQ will support
- Output Parameters: none
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

`ibv_resize_cq` resizes a completion queue (CQ). The parameter cqe must be at least the number of outstanding entries on the queue. The actual size of the queue may be larger than the specified value. The CQ may (or may not) contain completions when it is being resized thus, it can be resized during work with the CQ.

5.2.9. ibv_destroy_cq

- Template: `int ibv_destroy_cq(struct ibv_cq *cq)`

- Input Parameters:
  - **cq** CQ to destroy
- Output Parameters: none
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_destroy_cq frees a completion queue (CQ). This command will fail if there is any queue pair (QP) that still has the specified CQ associated with it.

5.2.10. ibv_create_comp_channel

- Template: **struct ibv_comp_channel {}ibv_create_comp_channel{*}(struct ibv_context *context)**
- Input Parameters:
  - **context** struct **ibv_context** from **ibv_open_device**
- Output Parameters: none
- Return Value: pointer to created CC or NULL on failure.

Description:

ibv_create_comp_channel creates a completion channel. A completion channel is a mechanism for the user to receive notifications when new completion queue event (CQE) has been placed on a completion queue (CQ).

5.2.11. ibv_destroy_comp_channel

- Template: **int ibv_destroy_comp_channel(struct ibv_comp_channel *channel)**
- Input Parameters:
  - **channel** struct **ibv_comp_channel** from **ibv_create_comp_channel**
- Output Parameters: none
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_destroy_comp_channel frees a completion channel. This command will fail if there are any completion queues (CQ) still associated with this completion channel.

5.3. Protection Domain Operations

Once you have established a protection domain (PD), you may create objects within that domain. This section describes operations available on a PD. These include registering memory regions (MR), creating queue pairs (QP) or shared receive queues (SRQ) and address handles (AH).

5.3.1. ibv_reg_mr

- Template: **struct ibv_mr {}ibv_reg_mr{*}(struct ibv_pd *pd, void *addr, size_t length, enum ibv_access_flags access)**
- Input Parameters:
  - **pd** protection domain, struct ibv_pd from ibv_alloc_pd
  - **addr** memory base address

    - **length** length of memory region in bytes
    - **access** access flags
- Output Parameters: none
- Return Value: pointer to created memory region (MR) or NULL on failure.
- Description:

ibv_reg_mr registers a memory region (MR), associates it with a protection domain (PD), and assigns it local and remote keys (lkey, rkey). All VPI commands that use memory require the memory to be registered via this command. The same physical memory may be mapped to different MRs even allowing different permissions or PDs to be assigned to the same memory, depending on user requirements.

Access flags may be bitwise or one of the following enumerations:

- **IBV_ACCESS_LOCAL_WRITE** Allow local host write access
- **IBV_ACCESS_REMOTE_WRITE** Allow remote hosts write access
- **IBV_ACCESS_REMOTE_READ** Allow remote hosts read access
- **IBV_ACCESS_REMOTE_ATOMIC** Allow remote hosts atomic access
- **IBV_ACCESS_MW_BIND** Allow memory windows on this MR

Local read access is implied and automatic.

Any VPI operation that violates the access permissions of the given memory operation will fail. Note that the queue pair (QP) attributes must also have the correct permissions or the operation will fail.

If **IBV_ACCESS_REMOTE_WRITE** or **IBV_ACCESS_REMOTE_ATOMIC** is set, then **IBV_ACCESS_LOCAL_WRITE** must be set as well.

Struct **ibv_mr** is defined as follows:

```
struct ibv_mr
{
struct ibv_context*context;
struct ibv_pd *pd;
void*addr;
size_tlength;
uint32_thandle;
uint32_tlkey;
uint32_trkey;
};
```

5.3.2. ibv_dereg_mr

- Template: **int ibv_dereg_mr(struct ibv_mr *mr)**
- Input Parameters:
    - **mr** struct **ibv_mr** from **ibv_reg_mr**
- Output Parameters: none
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_dereg_mr frees a memory region (MR). The operation will fail if any memory windows (MW) are still bound to the MR.

5.3.3. ibv_create_qp

- Template: **struct ibv_qp {}ibv_create_qp{*}(struct ibv_pd *pd, struct ibv_qp_init_attr *qp_init_attr)**
- Input Parameters:
  - **pd** struct **ibv_pd** from **ibv_alloc_pd**
  - **qp_init_attr** initial attributes of queue pair
- Output Parameters:
  - **qp_init_attractual** values are filled in
- Return Value: pointer to created queue pair (QP) or NULL on failure.

Description:

ibv_create_qp creates a QP. When a QP is created, it is put into the RESET state.

struct qp_init_attr is defined as follows:

```
struct ibv_qp_init_attr
{
    void    *qp_context;
    struct ibv_cq   *send_cq;
    struct ibv_cq   *recv_cq;
    struct ibv_srq  *srq;
    struct ibv_qp_cap   cap;
    enum ibv_qp_type    qp_type;
    int sq_sig_all;
    struct ibv_xrc_domain   *xrc_domain;
};

qp_context  (optional) user defined value associated with QP.
send_cq send CQ. This must be created by the user prior to calling ibv_create_qp.
recv_cq receive CQ. This must be created by the user prior to calling ibv_create_qp. It may be the same as send_cq.
srq (optional) shared receive queue. Only used for SRQ QP's.
cap defined below.
qp_type must be one of the following:
        IBV_QPT_RC = 2,
    IBV_QPT_UC,
    IBV_QPT_UD,
    IBV_QPT_XRC,
    IBV_QPT_RAW_PACKET = 8,
    IBV_QPT_RAW_ETH = 8
sq_sig_all  If this value is set to 1, all send requests (WR) will generate completion queue events (CQE). If this
 value is set to 0, only WRs that are flagged will generate CQE's (see ibv_post_send).
xrc_domain  (Optional) Only used for XRC operations.

struct ibv_qp_cap is defined as follows:

struct ibv_qp_cap
{
    uint32_t    max_send_wr;
    uint32_t    max_recv_wr;
    uint32_t    max_send_sge;
    uint32_t    max_recv_sge;
    uint32_t    max_inline_data;
};

max_send_wr Maximum number of outstanding send requests in the send queue.
max_recv_wr Maximum number of outstanding receive requests (buffers) in the receive queue.
max_send_sge    Maximum number of scatter/gather elements (SGE) in a WR on the send queue.
max_recv_sge    Maximum number of SGEs in a WR on the receive queue.
max_inline_data Maximum size in bytes of inline data on the send queue.
```

5.3.4. ibv_destroy_qp

- Template: **int ibv_destroy_qp(struct ibv_qp *qp)**
- Input Parameters:
  - **qp** struct ibv_qp from **ibv_create_qp**

- Output Parameters: none
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_destroy_qp frees a queue pair (QP).

5.3.5. ibv_create_srq

- Template: **struct ibv_srq *ibv_create_srq(struct ibv_pd *pd, struct ibv_srq_init_attr *srq_init_attr)**
- Input Parameters:
  - **pd** The protection domain associated with the shared receive queue (SRQ)
  - **srq_init_attr** A list of initial attributes required to create the SRQ
- Output Parameters:
  - **ibv_srq_attr** Actual values of the struct are set
- Return Value: A pointer to the created SRQ or NULL on failure

Description:

ibv_create_srq creates a shared receive queue (SRQ). **srq_attr** -> **max_wr** and **srq_attr** -> **max_sge** are read to determine the requested size of the SRQ, and set to the actual values allocated on return. If **ibv_create_srq** succeeds, then max_wr and **max_sge** will be at least as large as the requested values.

struct **ibv_srq** is defined as follows:

```
struct ibv_srq {
        struct ibv_context     *context;    struct ibv_context from ibv_open_device
        void                   *srq_context;
        struct ibv_pd          *pd;     Protection domain
        uint32_t                handle;
        pthread_mutex_t         mutex;
        pthread_cond_t          cond;
        uint32_t                events_completed;
}
struct ibv_srq_init_attr is defined as follows:
struct ibv_srq_init_attr
{
    void            *srq_context;
    struct ibv_srq_attr attr;
};

srq_context struct ibv_context from ibv_open_device
attr        An ibv_srq_attr struct defined as follows:


struct ibv_srq_attr is defined as follows:

struct ibv_srq_attr
{
    uint32_t        max_wr;
    uint32_t        max_sge;
    uint32_t        srq_limit;
};

max_wr      Requested maximum number of outstanding WRs in the SRQ
max_sge Requested number of scatter elements per WR
srq_limit;  The limit value of the SRQ (irrelevant for ibv_create_srq)
```

5.3.6. ibv_modify_srq

- Template: `int ibv_modify_srq (struct ibv_srq *srq, struct ibv_srq_attr *srq_attr, int srq_attr_mask)`
- Input Parameters:
  - `srq` The SRQ to modify
  - `srq_attr` Specifies the SRQ to modify (input)/the current values of the selected SRQ attributes are returned (output)
  - `srq_attr_mask` A bit-mask used to specify which SRQ attributes are being modified
- Output Parameters:
  - `srq_attr` The struct `ibv_srq_attr` is returned with the updated values
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

`ibv_modify_srq` modifies the attributes of the SRQ `srq` using the attribute values in `srq_attr` based on the mask `srq_attr_mask`. `srq_attr` is an `ibv_srq_attr` struct as defined above under the verb `ibv_create_srq`. The argument `srq_attr_mask` specifies the SRQ attributes to be modified. It is either 0 or the bitwise OR of one or more of the flags:

- `IBV_SRQ_MAX_WR` Resize the SRQ
- `IBV_SRQ_LIMIT` Set the SRQ limit

If any of the attributes to be modified is invalid, none of the attributes will be modified. Also, not all devices support resizing SRQs. To check if a device supports resizing, check if the `IBV_DEVICE_SRQ_RESIZE` bit is set in the device capabilities flags.

Modifying the SRQ limit arms the SRQ to produce an `IBV_EVENT_SRQ_LIMIT_REACHED` 'low watermark' async event once the number of WRs in the SRQ drops below the SRQ limit.

5.3.7. ibv_destroy_srq

- Template: `int ibv_destroy_srq(struct ibv_srq *srq)`
- Input Parameters:
  - `srq` The SRQ to destroy
- Output Parameters: none
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

`ibv_destroy_srq` destroys the specified SRQ. It will fail if any queue pair is still associated with this SRQ.

5.3.8. ibv_open_xrc_domain

- Template: `struct ibv_xrc_domain *ibv_open_xrc_domain(struct ibv_context *context, int fd, int oflag)`
- Input Parameters:
  - `context` struct `ibv_context` from `ibv_open_device`
  - `fd` The file descriptor to be associated with the XRC domain

    - **oflag** The desired file creation attributes
- Output Parameters: A file descriptor associated with the opened XRC domain
- Return Value: A reference to an opened XRC domain or NULL

Description: **ibv_open_xrc_domain** opens an eXtended Reliable Connection (XRC) domain for the RDMA device context. The desired file creation attributes **oflag** can either be 0 or the bitwise OR of **O_CREAT** and **O_EXCL**. If a domain belonging to the device named by the context is already associated with the inode, then the **O_CREAT** flag has no effect. If both **O_CREAT** and **O_EXCL** are set, open will fail if a domain associated with the inode already exists. Otherwise a new XRC domain will be created and associated with the inode specified by **fd**.

Please note that the check for the existence of the domain and creation of the domain if it does not exist is atomic with respect to other processes executing open with **fd** naming the same inode.

If **fd** equals -1, then no inode is associated with the domain, and the only valid value for **oflag** is **O_CREAT**.

Since each **ibv_open_xrc_domain** call increments the **xrc_domain** object's reference count, each such call must have a corresponding **ibv_close_xrc_domain** call to decrement the **xrc_domain** object's reference count.

5.3.9. ibv_create_xrc_srq

- Template:

```
struct ibv_srq {}ibv_create_xrc_srq{*}(struct ibv_pd *pd,
              struct ibv_xrc_domain *xrc_domain,
              struct ibv_cq *xrc_cq,
              struct ibv_srq_init_attr *srq_init_attr)
```

- Input Parameters:
    - **pd** The protection domain associated with the shared receive queue
    - **xrc_domain** The XRC domain
    - **xrc_cq** The CQ which will hold the XRC completion
    - **srq_init_attr** A list of initial attributes required to create the SRQ (described above)
- Output Parameters:
    - **ibv_srq_attr** Actual values of the struct are set
- Return Value: A pointer to the created SRQ or NULL on failure

Description:

ibv_create_xrc_srq creates an XRC shared receive queue (SRQ) associated with the protection domain pd, the XRC domain **domain_xrc** and the CQ which will hold the completion **xrc_cq**.

Struct **ibv_xrc_domain** is defined as follows:

```
struct ibv_xrc_domain {
        struct ibv_context     *context;    struct ibv_context from ibv_open_device
        uint64_t                handle;
}
```

5.3.10. ibv_close_xrc_domain

- Template: `int ibv_close_xrc_domain(struct ibv_xrc_domain *d)`
- Input Parameters:
- `d` A pointer to the XRC domain the user wishes to close
- Output Parameters: none
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

`ibv_close_xrc_domain` closes the XRC domain, d. If this happens to be the last reference, then the XRC domain will be destroyed. This function decrements a reference count and may fail if any QP or SRQ are still associated with the XRC domain being closed.

5.3.11. ibv_create_xrc_rcv_qp

- Template: `int ibv_create_xrc_rcv_qp(struct ibv_qp_init_attr *init_attr, uint32_t *xrc_rcv_qpn)`
- Input Parameters:
    - `init_attr` The structure to be populated with QP information
    - `xrc_rcv_qpn` The QP number associated with the receive QP to be created
- Output Parameters:
    - `init_attr` Populated with the XRC domain information the QP will be associated with
    - `xrc_rcv_qpn` The QP number associated with the receive QP being created
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

`ibv_create_xrc_rcv_qp` creates an XRC queue pair (QP) to serve as a receive side only QP and returns the QP number through `xrc_rcv_qpn`. This number must be passed to the remote (sender) node. The remote node will use `xrc_rcv_qpn` in `ibv_post_send` when it sends messages to an XRC SRQ on this host in the same xrc domain as the XRC receive QP.

The QP with number `xrc_rcv_qpn` is created in kernel space and persists until the last process registered for the QP called `ibv_unreg_xrc_rcv_qp`, at which point the QP is destroyed. The process which creates this QP is automatically registered for it and should also call `ibv_unreg_xrc_rcv_qp` at some point to unregister.

Any process which wishes to receive on an XRC SRQ via this QP must call `ibv_reg_xrc_rcv_qp` for this QP to ensure that the QP will not be destroyed while they are still using it.

Please note that because the QP `xrc_rcv_qpn` is a receive only QP, the send queue in the `init_attr` struct is ignored.

5.3.12. ibv_modify_xrc_rcv_qp

- Template: `int ibv_modify_xrc_rcv_qp(struct ibv_xrc_domain *xrc_domain, uint32_t xrc_qp_num, struct ibv_qp_attr *attr, int attr_mask)`
- Input Parameters:

- **xrc_domain** The XRC domain associated with this QP **xrc_qp_num** The queue pair number to identify this QP
- **attr** The attributes to use to modify the XRC receive QP **attr_mask** The mask to use for modifying the QP attributes

- Output Parameters: None
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_modify_xrc_rcv_qp modifies the attributes of an XRC receive QP with the number **xrc_qp_num** which is associated with the attributes in the struct attr according to the mask **attr_mask**. It then moves the QP through the following transitions: Reset → Init → RTR.

At least the following masks must be set (the user may add optional attributes as needed)

Next State	Next State Required attributes
Init	**IBV_QP_STATE**, **IBV_QP_PKEY_INDEX**, **IBV_QP_PORT**, **IBV_QP_ACCESS_FLAGS**
RTR	**IBV_QP_STATE**, **IBV_QP_AV**, **IBV_QP_PATH_MTU**, **IBV_QP_DEST_QPN**, **IBV_QP_RQ_PSN**, **IBV_QP_MAX_DEST_RD_ATOMIC**, **IBV_QP_MIN_RNR_TIMER**

Please note that if any attribute to modify is invalid or if the mask as invalid values, then none of the attributes will be modified, including the QP state.

5.3.13. ibv_reg_xrc_rcv_qp

- Template: **int ibv_reg_xrc_rcv_qp(struct ibv_xrc_domain *xrc_domain, uint32_t xrc_qp_num)**
- Input Parameters:
  - **xrc_domain** The XRC domain associated with the receive QP
  - **xrc_qp_num** The number associated with the created QP to which the user process is to be registered
- Output Parameters: None
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_reg_xrc_rcv_qp registers a user process with the XRC receive QP whose number is **xrc_qp_num** associated with the XRC domain **xrc_domain**.This function may fail if the number **xrc_qp_num** is not the number of a valid XRC receive QP (for example if the QP is not allocated or it is the number of a non-XRC QP), or the XRC receive QP was created with an XRC domain other than **xrc_domain**.

5.3.14. ibv_unreg_xrc_rcv_qp

- Template: **int ibv_unreg_xrc_rcv_qp(struct ibv_xrc_domain *xrc_domain, uint32_t xrc_qp_num)**
- Input Parameters:

    - **xrc_domain** The XRC domain associated with the XRC receive QP from which the user wishes to unregister
    - **xrc_qp_num** The QP number from which the user process is to be unregistered
- Output Parameters: None
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_unreg_xrc_rcv_qp unregisters a user process from the XRC receive QP number **xrc_qp_num** which is associated with the XRC domain **xrc_domain**. When the number of user processes registered with this XRC receive QP drops to zero, the QP is destroyed.

5.3.15. ibv_create_ah

- Template: **struct ibv_ah {}ibv_create_ah{*}(struct ibv_pd *pd, struct ibv_ah_attr *attr)**
- Input Parameters:
    - **pd** struct **ibv_pd** from **ibv_alloc_pd**
    - **attr** attributes of address
- Output Parameters: none
- Return Value: pointer to created address handle (AH) or NULL on failure.

Description:

ibv_create_ah creates an AH. An AH contains all of the necessary data to reach a remote destination. In connected transport modes (RC, UC) the AH is associated with a queue pair (QP). In the datagram transport modes (UD), the AH is associated with a work request (WR).

Struct **ibv_ah_attr** is defined as follows:

```
struct ibv_ah_attr
{
    struct ibv_global_route grh;
    uint16_t    dlid;
    uint8_t sl;
    uint8_t src_path_bits;
    uint8_t static_rate;
    uint8_t is_global;
    uint8_t port_num;
};

grh defined below
dlid    destination lid
sl  service level
src_path_bits   source path bits
static_rate static rate
is_global   this is a global address, use grh.
port_num    physical port number to use to reach this destination
struct ibv_global_route is defined as follows:

struct ibv_global_route
{
    union ibv_gid   dgid;
    uint32_t    flow_label;
    uint8_t sgid_index;
    uint8_t hop_limit;
    uint8_t traffic_class;
};

dgid    destination GID (see ibv_query_gid for definition)
flow_label  flow label
sgid_index  index of source GID (see ibv_query_gid)
hop_limit   hop limit
traffic_class   traffic class
```

5.3.16. ibv_destroy_ah

- Template: **int ibv_destroy_ah(struct ibv_ah *ah)**

- Input Parameters:
    - **ah** struct **ibv_ah** from **ibv_create_ah**
- Output Parameters: none
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_destroy_ah frees an address handle (AH). Once an AH is destroyed, it cannot be used anymore in UD QPs.

5.4. Queue Pair Bringup (ibv_modify_qp)

Queue pairs (QP) must be transitioned through an incremental sequence of states prior to being able to be used for communication.

QP states:

```
RESET   Newly created, queues empty.
INIT    Basic information set. Ready for posting to receive queue.
RTR Ready to Receive. Remote address info set for connected QPs, QP may now receive packets.
RTS Ready to Send. Timeout and retry parameters set, QP may now send packets.
```

5.4.1. ibv_modify_qp

- Template: **int ibv_modify_qp(struct ibv_qp *qp, struct ibv_qp_attr *attr, enum ibv_qp_attr_mask attr_mask)**
- Input Parameters:
    - **qp** struct **ibv_qp** from **ibv_create_qp**
    - **attr** QP attributes
    - **attr_mask** Bit mask that defines which attributes within attr have been set for this call
- Output Parameters: none
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_modify_qp this verb changes QP attributes and one of those attributes may be the QP state. Its name is a bit of a misnomer, since you cannot use this command to modify qp attributes at will. There is a very strict set of attributes that may be modified during each transition, and transitions must occur in the proper order. The following subsections describe each transition in more detail.

Struct **ibv_qp_attr** is defined as follows:

```
struct ibv_qp_attr
{
    enum ibv_qp_state   qp_state;
    enum ibv_qp_state   cur_qp_state;
    enum ibv_mtu    path_mtu;
    enum ibv_mig_state  path_mig_state;
    uint32_t    qkey;
    uint32_t    rq_psn;
    uint32_t    sq_psn;
    uint32_t    dest_qp_num;
    int qp_access_flags;
    struct ibv_qp_cap   cap;
    struct ibv_ah_attr  ah_attr;
```

```
    struct ibv_ah_attr  alt_ah_attr;
    uint16_t    pkey_index;
    uint16_t    alt_pkey_index;
    uint8_t en_sqd_async_notify;
    uint8_t sq_draining;
    uint8_t max_rd_atomic;
    uint8_t max_dest_rd_atomic;
    uint8_t min_rnr_timer;
    uint8_t port_num;
    uint8_t timeout;
    uint8_t retry_cnt;
    uint8_t rnr_retry;
    uint8_t alt_port_num;
    uint8_t alt_timeout;
};
```

The following values select one of the above attributes and should be OR'd into the **attr_mask** field:

```
IBV_QP_STATE
    IBV_QP_CUR_STATE
    IBV_QP_EN_SQD_ASYNC_NOTIFY
    IBV_QP_ACCESS_FLAGS
    IBV_QP_PKEY_INDEX
    IBV_QP_PORT
    IBV_QP_QKEY
    IBV_QP_AV
    IBV_QP_PATH_MTU
    IBV_QP_TIMEOUT
    IBV_QP_RETRY_CNT
    IBV_QP_RNR_RETRY
    IBV_QP_RQ_PSN
    IBV_QP_MAX_QP_RD_ATOMIC
    IBV_QP_ALT_PATH
    IBV_QP_MIN_RNR_TIMER
    IBV_QP_SQ_PSN
    IBV_QP_MAX_DEST_RD_ATOMIC
    IBV_QP_PATH_MIG_STATE
    IBV_QP_CAP
    IBV_QP_DEST_QPN
```

5.4.2. RESET to INIT

When a queue pair (QP) is newly created, it is in the RESET state. The first state transition that needs to happen is to bring the QP in the INIT state.

- Required Attributes:

```
*** All QPs ***
qp_state / IBV_QP_STATE IBV_QPS_INIT
pkey_index / IBV_QP_PKEY_INDEX  pkey index, normally 0
port_num / IBV_QP_PORT  physical port number (1...n)
qp_access_flags /
   IBV_QP_ACCESS_FLAGS  access flags (see ibv_reg_mr)

*** Unconnected QPs only ***
qkey / IBV_QP_QKEY  qkey (see ibv_post_send)
```

- Optional Attributes: none
- Effect of Transition: Once the QP is transitioned into the INIT state, the user may begin to post receive buffers to the receive queue via the **ibv_post_recv** command. At least one receive buffer should be posted before the QP can be transitioned to the RTR state.

5.4.3. INIT to RTR

Once a queue pair (QP) has receive buffers posted to it, it is now possible to transition the QP into the ready to receive (RTR) state.

- Required Attributes:

```
*** All QPs ***
qp_state / IBV_QP_STATE IBV_QPS_RTR
path_mtu / IBV_QP_PATH_MTU  IB_MTU_256
    IB_MTU_512 (recommended value)
    IB_MTU_1024
    IB_MTU_2048
    IB_MTU_4096

*** Connected QPs only ***
```

```
ah_attr / IBV_QP_AV an address handle (AH) needs to be created and filled in as appropriate. Minimally,
ah_attr.dlid needs to be filled in.
dest_qp_num / IBV_QP_DEST_QPN   QP number of remote QP.
rq_psn / IBV_QP_RQ_PSN  starting receive packet sequence number (should match remote QP's sq_psn)
max_dest_rd_atomic /
    IBV_MAX_DEST_RD_ATOMIC  maximum number of resources for incoming RDMA requests
min_rnr_timer /
    IBV_QP_MIN_RNR_TIMER    minimum RNR NAK timer (recommended value: 12)
```

- Optional Attributes:

```
*** All QPs ***
qp_access_flags /
    IBV_QP_ACCESS_FLAGS access flags (see ibv_reg_mr)
pkey_index / IBV_QP_PKEY_INDEX  pkey index, normally 0

*** Connected QPs only ***
alt_ah_attr / IBV_QP_ALT_PATH   AH with alternate path info filled in

*** Unconnected QPs only ***
qkey / IBV_QP_QKEY  qkey (see ibv_post_send)
```

- Effect of Transition: Once the QP is transitioned into the RTR state, the QP begins receive processing.

5.4.4. RTR to RTS

Once a queue pair (QP) has reached ready to receive (RTR) state, it may then be transitioned to the ready to send (RTS) state.

- Required Attributes:

```
*** All QPs ***
qp_state / IBV_QP_STATE IBV_QPS_RTS

*** Connected QPs only ***
timeout / IBV_QP_TIMEOUT    local ack timeout (recommended value: 14)
retry_cnt / IBV_QP_RETRY_CNT    retry count (recommended value: 7)
rnr_retry / IBV_QP_RNR_RETRY    RNR retry count (recommended value: 7)
sq_psn / IBV_SQ_PSN send queue starting packet sequence number (should match remote QP's rq_psn)
max_rd_atomic
    / IBV_QP_MAX_QP_RD_ATOMIC   number of outstanding RDMA reads and atomic operations allowed.
```

- Optional Attributes:

```
*** All QPs ***
qp_access_flags /
    IBV_QP_ACCESS_FLAGS access flags (see ibv_reg_mr)

*** Connected QPs only ***
alt_ah_attr / IBV_QP_ALT_PATH   AH with alternate path info filled in
min_rnr_timer /
    IBV_QP_MIN_RNR_TIMER    minimum RNR NAK timer

*** Unconnected QPs only ***
qkey / IBV_QP_QKEY  qkey (see ibv_post_send)
```

- Effect of Transition: Once the QP is transitioned into the RTS state, the QP begins send processing and is fully operational. The user may now post send requests with the **ibv_post_send** command.

5.5. Active Queue Pair Operations

A QP can be queried staring at the point it was created and once a queue pair is completely operational, you may query it, be notified of events and conduct send and receive operations on it. This section describes the operations available to perform these actions.

5.5.1. ibv_query_qp

- Template: **int ibv_query_qp(struct ibv_qp *qp, struct ibv_qp_attr *attr, enum ibv_qp_attr_mask attr_mask, struct ibv_qp_init_attr *init_attr)**
- Input Parameters:

    - **qp** struct **ibv_qp** from **ibv_create_qp**
    - **attr_mask** bitmask of items to query (see **ibv_modify_qp**)
- Output Parameters:
    - **attr** struct **ibv_qp_attr** to be filled in with requested attributes
    - **init_attr** struct **ibv_qp_init_attr** to be filled in with initial attributes
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_query_qp retrieves the various attributes of a queue pair (QP) as previously set through **ibv_create_qp** and **ibv_modify_qp**. The user should allocate a struct **ibv_qp_attr** and a struct **ibv_qp_init_attr** and pass them to the command. These structs will be filled in upon successful return. The user is responsible to free these structs.

Struct **ibv_qp_init_attr** is described in **ibv_create_qp** and struct **ibv_qp_attr** is described in **ibv_modify_qp**.

5.5.2. ibv_query_srq

- Template: **int ibv_query_srq(struct ibv_srq *srq, struct ibv_srq_attr *srq_attr)**
- Input Parameters:
    - **srq** The SRQ to query
    - **srq_attr** The attributes of the specified SRQ
- Output Parameters:
    - **srq_attr** The struct **ibv_srq_attr** is returned with the attributes of the specified SRQ
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_query_srq returns the attributes list and current values of the specified SRQ. It returns the attributes through the pointer **srq_attr** which is an **ibv_srq_attr** struct described above under **ibv_create_srq**. If the value of **srq_limit** in **srq_attr** is 0, then the SRQ limit reached ('low watermark') event is not or is no longer armed. No asynchronous events will be generated until the event is re-armed.

5.5.3. ibv_query_xrc_rcv_qp

- Template: **int ibv_query_xrc_rcv_qp(struct ibv_xrc_domain *xrc_domain, uint32_t xrc_qp_num, struct ibv_qp_attr *attr, int attr_mask, struct ibv_qp_init_attr *init_attr)**
- Input Parameters:
    - **xrc_domain** The XRC domain associated with this QP
    - **xrc_qp_num** The queue pair number to identify this QP
    - **attr** The **ibv_qp_attr** struct in which to return the attributes
    - **attr_mask** A mask specifying the minimum list of attributes to retrieve
    - **init_attr** The **ibv_qp_init_attr** struct to return the initial attributes
- Output Parameters:

- **attr** A pointer to the struct containing the QP attributes of interest
- **init_attr** A pointer to the struct containing initial attributes

- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_query_xrc_rcv_qp retrieves the attributes specified in **attr_mask** for the XRC receive QP with the number **xrc_qp_num** and domain **xrc_domain**. It returns them through the pointers **attr** and **init_attr**.

The **attr_mask** specifies a minimal list to retrieve. Some RDMA devices may return extra attributes not requested. Attributes are valid if they have been set using the **ibv_modify_xrc_rcv_qp**.

The exact list of valid attributes depends on the QP state. Multiple **ibv_query_xrc_rcv_qp** calls may yield different returned values for these attributes: **qp_state**, **path_mig_state**, **sq_draining**, **ah_attr** if automatic path migration (APM) is enabled.

5.5.4. ibv_post_recv

- Template: **int ibv_post_recv(struct ibv_qp *qp, struct ibv_recv_wr *wr, struct ibv_recv_wr **bad_wr)**
- Input Parameters:
  - **qp** struct **ibv_qp** from **ibv_create_qp**
  - **wr** first work request (WR) containing receive buffers
- Output Parameters:
  - **bad_wr** pointer to first rejected WR
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_post_recv posts a linked list of WRs to a queue pair's (QP) receive queue. At least one receive buffer should be posted to the receive queue to transition the QP to RTR. Receive buffers are consumed as the remote peer executes Send, Send with Immediate and RDMA Write with Immediate operations. Receive buffers are not used for other RDMA operations. Processing of the WR list is stopped on the first error and a pointer to the offending WR is returned in **bad_wr**.

Struct **ibv_recv_wr** is defined as follows:

```
struct ibv_recv_wr
{
    uint64_t    wr_id;
    struct ibv_recv_wr  *next;
    struct ibv_sge  *sg_list;
    int num_sge;
};

wr_id   user assigned work request ID
next    pointer to next WR, NULL if last one.
sg_list scatter array for this WR
num_sge number of entries in sg_list
struct ibv_sge is defined as follows:
struct ibv_sge
{
    uint64_t    addr;
    uint32_t    length;
    uint32_t    lkey;
```

```
};

addr    address of buffer
length  length of buffer
lkey    local key (lkey) of buffer from ibv_reg_mr
```

5.5.5. ibv_post_send

- Template: **int ibv_post_send(struct ibv_qp *qp, struct ibv_send_wr *wr, struct ibv_send_wr **bad_wr)**
- Input Parameters:
  - **qp** struct **ibv_qp** from **ibv_create_qp**
  - **wr** first work request (WR)
- Output Parameters:
  - **bad_wr** pointer to first rejected WR
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_post_send posts a linked list of WRs to a queue pair's (QP) send queue. This operation is used to initiate all communication, including RDMA operations. Processing of the WR list is stopped on the first error and a pointer to the offending WR is returned in **bad_wr**.

The user should not alter or destroy AHs associated with WRs until the request has been fully executed and a completion queue entry (CQE) has been retrieved from the corresponding completion queue (CQ) to avoid unexpected behaviour.

The buffers used by a WR can only be safely reused after the WR has been fully executed and a WCE has been retrieved from the corresponding CQ. However, if the **IBV_SEND_INLINE** flag was set, the buffer can be reused immediately after the call returns.

Struct **ibv_send_wr** is defined as follows:

```
struct ibv_send_wr
{
    uint64_t    wr_id;
    struct ibv_send_wr  *next;
    struct ibv_sge  *sg_list;
    int num_sge;
    enum ibv_wr_opcode  opcode;
    enum ibv_send_flags send_flags;
    uint32_t    imm_data;/* network byte order */
    union
    {
        struct
        {
            uint64_t                remote_addr;
            uint32_t    rkey;
        } rdma;
        struct
        {
            uint64_t    remote_addr;
            uint64_t    compare_add;
            uint64_t    swap;
            uint32_t    rkey;
        } atomic;
        struct
        {
            struct ibv_ah   *ah;
            uint32_t    remote_qpn;
            uint32_t    remote_qkey;
        } ud;
    } wr;
    uint32_t    xrc_remote_srq_num;
};
wr_id   user assigned work request ID
next    pointer to next WR, NULL if last one.
sg_list scatter/gather array for this WR
num_sge number of entries in sg_list
opcode  IBV_WR_RDMA_WRITE
    IBV_WR_RDMA_WRITE_WITH_IMM
    IBV_WR_SEND
```

```
    IBV_WR_SEND_WITH_IMM
    IBV_WR_RDMA_READ
    IBV_WR_ATOMIC_CMP_AND_SWP
    IBV_WR_ATOMIC_FETCH_AND_ADD
send_flags  (optional) - this is a bitwise OR of the flags. See the details below.
imm_data    immediate data to send in network byte order
remote_addr remote virtual address for RDMA/atomic operations
rkey    remote key (from ibv_reg_mr on remote) for RDMA/atomic operations
compare_add compare value for compare and swap operation
swap    swap value
ah  address handle (AH) for datagram operations
remote_qpn  remote QP number for datagram operations
remote_qkey Qkey for datagram operations
xrc_remote_srq_num  shared receive queue (SRQ) number for the destination extended reliable connection (XRC). Only
used for XRC operations.

send flags:
IBV_SEND_FENCE  set fence indicator
IBV_SEND_SIGNALED   send completion event for this WR. Only meaningful for QPs that had the sq_sig_all set to 0
IBV_SEND_SEND_SOLICITED
    set solicited event indicator
IBV_SEND_INLINE send data in sge_list as inline data.
struct ibv_sge is defined in ibv_post_recv.
```

5.5.6. ibv_post_srq_recv

- Template: **int ibv_post_srq_recv(struct ibv_srq *srq, struct ibv_recv_wr *recv_wr, struct ibv_recv_wr **bad_recv_wr)**
- Input Parameters:
  - **srq** The SRQ to post the work request to
  - **recv_wr** A list of work requests to post on the receive queue
- Output Parameters:
  - **bad_recv_wr** pointer to first rejected WR
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_post_srq_recv posts a list of work requests to the specified SRQ. It stops processing the WRs from this list at the first failure (which can be detected immediately while requests are being posted), and returns this failing WR through the **bad_recv_wr** parameter.

The buffers used by a WR can only be safely reused after WR the request is fully executed and a work completion has been retrieved from the corresponding completion queue (CQ).

If a WR is being posted to a UD QP, the Global Routing Header (GRH) of the incoming message will be placed in the first 40 bytes of the buffer(s) in the scatter list. If no GRH is present in the incoming message, then the first 40 bytes will be undefined. This means that in all cases for UD QPs, the actual data of the incoming message will start at an offset of 40 bytes into the buffer(s) in the scatter list.

5.5.7. ibv_req_notify_cq

- Template: **int ibv_req_notify_cq(struct ibv_cq *cq, int solicited_only)**
- Input Parameters:
  - **cq** struct **ibv_cq** from **ibv_create_cq**
  - **solicited_only** only notify if WR is flagged as solicited
- Output Parameters: none
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_req_notify_cq arms the notification mechanism for the indicated completion queue (CQ). When a completion queue entry (CQE) is placed on the CQ, a completion event will be sent to the completion channel (CC) associated with the CQ. If there is already a CQE in that CQ, an event won't be generated for this event. If the solicited_only flag is set, then only CQEs for WRs that had the solicited flag set will trigger the notification.

The user should use the **ibv_get_cq_event** operation to receive the notification.

The notification mechanism will only be armed for one notification. Once a notification is sent, the mechanism must be re-armed with a new call to **ibv_req_notify_cq** .

5.5.8. ibv_get_cq_event

- Template: **int ibv_get_cq_event(struct ibv_comp_channel *channel, struct ibv_cq **cq, void **cq_context)**
- Input Parameters:
  - **channel** struct **ibv_comp_channel** from **ibv_create_comp_channel**
- Output Parameters:
  - **cq** pointer to completion queue (CQ) associated with event
  - **cq_context** user supplied context set in **ibv_create_cq**
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_get_cq_event waits for a notification to be sent on the indicated completion channel (CC). Note that this is a blocking operation. The user should allocate pointers to a struct **ibv_cq** and a void to be passed into the function. They will be filled in with the appropriate values upon return. It is the user's responsibility to free these pointers.

Each notification sent MUST be acknowledged with the **ibv_ack_cq_events** operation. Since the **ibv_destroy_cq** operation waits for all events to be acknowledged, it will hang if any events are not properly acknowledged.

Once a notification for a completion queue (CQ) is sent on a CC, that CQ is now "disarmed" and will not send any more notifications to the CC until it is rearmed again with a new call to the **ibv_req_notify_cq** operation.

This operation only informs the user that a CQ has completion queue entries (CQE) to be processed, it does not actually process the CQEs. The user should use the **ibv_poll_cq** operation to process the CQEs.

5.5.9. ibv_ack_cq_events

- Template: **void ibv_ack_cq_events(struct ibv_cq *cq, unsigned int nevents)**
- Input Parameters:
  - **cq** struct **ibv_cq** from **ibv_create_cq**
  - **nevents** number of events to acknowledge (1...n)
- Output Parameters: None
- Return Value: None

Description:

ibv_ack_cq_events acknowledges events received from **ibv_get_cq_event**. Although each notification received from **ibv_get_cq_event** counts as only one event, the user may acknowledge multiple events through a single call to **ibv_ack_cq_events**. The number of events to acknowledge is passed in nevents and should be at least 1. Since this operation takes a mutex, it is somewhat expensive and acknowledging multiple events in one call may provide better performance.
See **ibv_get_cq_event** for additional details.

5.5.10. ibv_poll_cq

- Template: **int ibv_poll_cq(struct ibv_cq *cq, int num_entries, struct ibv_wc *wc)**
- Input Parameters:
  - **cq** struct **ibv_cq** from **ibv_create_cq**
  - **num_entries** maximum number of completion queue entries (CQE) to return
- Output Parameters:
  - **wc** CQE array
- Return Value: Number of CQEs in array wc or -1 on error

Description:

ibv_poll_cq retrieves CQEs from a completion queue (CQ). The user should allocate an array of struct **ibv_wc** and pass it to the call in **wc**. The number of entries available in wc should be passed in **num_entries**. It is the user's responsibility to free this memory.

The number of CQEs actually retrieved is given as the return value. CQs must be polled regularly to prevent an overrun. In the event of an overrun, the CQ will be shut down and an async event **IBV_EVENT_CQ_ERR** will be sent.

Struct **ibv_wc** is defined as follows:

```
struct ibv_wc
{
    uint64_t    wr_id;
    enum ibv_wc_status  status;
    enum ibv_wc_opcode  opcode;
    uint32_t    vendor_err;
    uint32_t    byte_len;
    uint32_t    imm_data;/* network byte order */
    uint32_t    qp_num;
    uint32_t    src_qp;
    enum ibv_wc_flags   wc_flags;
    uint16_t    pkey_index;
    uint16_t    slid;
    uint8_t sl;
    uint8_t dlid_path_bits;
};
wr_id   user specified work request id as given in ibv_post_send or ibv_post_recv
status  IBV_WC_SUCCESS
    IBV_WC_LOC_LEN_ERR
    IBV_WC_LOC_QP_OP_ERR
    IBV_WC_LOC_EEC_OP_ERR
    IBV_WC_LOC_PROT_ERR
    IBV_WC_WR_FLUSH_ERR
    IBV_WC_MW_BIND_ERR
    IBV_WC_BAD_RESP_ERR
    IBV_WC_LOC_ACCESS_ERR
    IBV_WC_REM_INV_REQ_ERR
    IBV_WC_REM_ACCESS_ERR
    IBV_WC_REM_OP_ERR
    IBV_WC_RETRY_EXC_ERR
    IBV_WC_RNR_RETRY_EXC_ERR
    IBV_WC_LOC_RDD_VIOL_ERR
    IBV_WC_REM_INV_RD_REQ_ERR
    IBV_WC_REM_ABORT_ERR
    IBV_WC_INV_EECN_ERR
    IBV_WC_INV_EEC_STATE_ERR
    IBV_WC_FATAL_ERR
    IBV_WC_RESP_TIMEOUT_ERR
    IBV_WC_GENERAL_ERR
opcode      IBV_WC_SEND,
```

```
    IBV_WC_RDMA_WRITE,
    IBV_WC_RDMA_READ,
    IBV_WC_COMP_SWAP,
    IBV_WC_FETCH_ADD,
    IBV_WC_BIND_MW,
    IBV_WC_RECV         = 1 << 7,
    IBV_WC_RECV_RDMA_WITH_IMM
vendor_err  vendor specific error
byte_len    number of bytes transferred
imm_data    immediate data
qp_num  local queue pair (QP) number
src_qp  remote QP number
wc_flags    see below
pkey_index  index of pkey (valid only for GSI QPs)
slid    source local identifier (LID)
sl  service level (SL)
dlid_path_bits  destination LID path bits

flags:
IBV_WC_GRH  global route header (GRH) is present in UD packet
IBV_WC_WITH_IMM immediate data value is valid
```

5.5.11. ibv_init_ah_from_wc

- Template: **int ibv_init_ah_from_wc(struct ibv_context *context, uint8_t port_num, struct ibv_wc *wc, struct ibv_grh *grh, struct ibv_ah_attr *ah_attr)**
- Input Parameters:
  - **context** struct **ibv_context** from **ibv_open_device**. This should be the device the completion queue entry (CQE) was received on.
  - **port_num** physical port number (1..n) that CQE was received on wc received CQE from **ibv_poll_cq**
  - **grh** global route header (GRH) from packet (see description)
- Output Parameters:
  - **ah_attr** address handle (AH) attributes
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_init_ah_from_wc initializes an AH with the necessary attributes to generate a response to a received datagram. The user should allocate a struct **ibv_ah_attr** and pass this in. If appropriate, the GRH from the received packet should be passed in as well. On UD connections the first 40 bytes of the received packet may contain a GRH. Whether or not this header is present is indicated by the **IBV_WC_GRH** flag of the CQE. If the GRH is not present on a packet on a UD connection, the first 40 bytes of a packet are undefined.

When the function **ibv_init_ah_from_wc** completes, the **ah_attr** will be filled in and the **ah_attr** may then be used in the **ibv_create_ah** function. The user is responsible for freeing **ah_attr**.

Alternatively, **ibv_create_ah_from_wc** may be used instead of this operation.

5.5.12. ibv_create_ah_from_wc

- Template: **struct ibv_ah {}ibv_create_ah_from_wc{*}(struct ibv_pd *pd, struct ibv_wc *wc, struct ibv_grh *grh, uint8_t port_num)**
- Input Parameters:
  - **pd** protection domain (PD) from **ibv_alloc_pd**
  - **wc** completion queue entry (CQE) from **ibv_poll_cq**
  - **grh** global route header (GRH) from packet
  - **port_num** physical port number (1..n) that CQE was received on

- Output Parameters: none
- Return Value: Created address handle (AH) on success or -1 on error

Description:

ibv_create_ah_from_wc combines the operations **ibv_init_ah_from_wc** and **ibv_create_ah**. See the description of those operations for details.

5.5.13. ibv_attach_mcast

- Template: **int ibv_attach_mcast(struct ibv_qp *qp, const union ibv_gid *gid, uint16_t lid)**
- Input Parameters:
  - **qp** QP to attach to the multicast group
  - **gid** The multicast group GID
  - **lid** The multicast group LID in host byte order
- Output Parameters: none
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_attach_mcast attaches the specified QP, **qp**, to the multicast group whose multicast group GID is gid, and multicast LID is **lid**.

Only QPs of Transport Service Type **IBV_QPT_UD** may be attached to multicast groups.

To receive multicast messages, a join request for the multicast group must be sent to the subnet administrator (SA), so that the fabric's multicast routing is configured to deliver messages to the local port.

If a QP is attached to the same multicast group multiple times, the QP will still receive a single copy of a multicast message.

5.5.14. ibv_detach_mcast

- Template: **int ibv_detach_mcast(struct ibv_qp *qp, const union ibv_gid *gid, uint16_t lid)**
- Input Parameters:
  - **qp** QP to attach to the multicast group
  - **gid** The multicast group GID
  - **lid** The multicast group LID in host byte order
- Output Parameters: none
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_detach_mcast detaches the specified QP, **qp**, from the multicast group whose multicast group GID is **gid**, and multicast LID is **lid**.

5.6. Event Handling Operations

5.6.1. ibv_get_async_event

- Template: **int ibv_get_async_event(struct ibv_context *context, struct ibv_async_event *event)**
- Input Parameters:
  - **context** struct **ibv_context** from **ibv_open_device**
  - **event** A pointer to use to return the async event
- Output Parameters:
  - **event** A pointer to the async event being sought
- Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_get_async_event gets the next asynchronous event of the RDMA device context **context** and returns it through the pointer **event** which is an **ibv_async_event** struct. All async events returned by **ibv_get_async_event** must eventually be acknowledged with **ibv_ack_async_event** . **ibv_get_async_event()** is a blocking function. If multiple threads call this function simultaneously, then when an async event occurs, only one thread will receive it, and it is not possible to predict which thread will receive it.

Struct **ibv_async_event** is defined as follows:

```
struct ibv_async_event {
    union {
    struct ibv_cq *cq;                  The CQ that got the event
    struct ibv_qp *qp;              The QP that got the event
    struct ibv_srq *srq;            The SRQ that got the event
    int port_num;               The port number that got the event
    } element;
    enum ibv_event_type event_type; Type of event
};
```

One member of the element union will be valid, depending on the **event_type** member of the structure. **event_type** will be one of the following events:

- QP events:

Event	Description
IBV_EVENT_QP_FATAL	Error occurred on a QP and it transitioned to error state
IBV_EVENT_QP_REQ_ERR	Invalid Request Local Work Queue Error
IBV_EVENT_QP_ACCESS_ERR	Local access violation error
IBV_EVENT_COMM_EST	Communication was established on a QP
IBV_EVENT_SQ_DRAINED	Send Queue was drained of outstanding messages in progress
IBV_EVENT_PATH_MIG	A connection has migrated to the alternate path
IBV_EVENT_PATH_MIG_ERR	A connection failed to migrate to the alternate path
IBV_EVENT_QP_LAST_WQE_REACHED	Last WQE Reached on a QP associated with an SRQ

- CQ events:

Event	Description
`IBV_EVENT_CQ_ERR`	CQ is in error (CQ overrun)

- SRQ events:

Event	Description
`IBV_EVENT_SRQ_ERR`	Error occurred on an SRQ
`IBV_EVENT_SRQ_LIMIT_REACHED`	SRQ limit was reached

- Port events:

Event	Description
`IBV_EVENT_PORT_ACTIVE`	Link became active on a port
`IBV_EVENT_PORT_ERR`	Link became unavailable on a port
`IBV_EVENT_LID_CHANGE`	LID was changed on a port
`IBV_EVENT_PKEY_CHANGE`	P_Key table was changed on a port
`IBV_EVENT_SM_CHANGE`	SM was changed on a port
`IBV_EVENT_CLIENT_REREGISTER`	SM sent a CLIENT_REREGISTER request to a port
`IBV_EVENT_GID_CHANGE`	GID table was changed on a port

- CA events:

Event	Description
IBV_EVENT_DEVICE_FATAL	CA is in FATAL state

5.6.2. ib_ack_async_event

- Template: `void ibv_ack_async_event(struct ibv_async_event *event)`
- Input Parameters:
  - `event` A pointer to the event to be acknowledged
- Output Parameters: None
- Return Value: None

Description:

All async events that `ibv_get_async_event()` returns must be acknowledged using `ibv_ack_async_event()`. To avoid races, destroying an object (CQ, SRQ or QP) will wait for all affiliated events for the object to be acknowledged; this avoids an application retrieving an affiliated event after the corresponding object has already been destroyed.

5.6.3. ibv_event_type_str

- Template: `const char {}ibv_event_type_str{*}(enum ibv_event_type event_type)`

- Input Parameters:
    - **event_type** – **ibv_event_type** enum value
- Output Parameters: None
- Return Value: A constant string which describes the enum value event_type

Description:

ibv_event_type_str returns a string describing the event type enum value, **event_type**. **event_type** may be any one of the 19 different enum values describing different IB events.

```
ibv_event_type {
    IBV_EVENT_CQ_ERR,
    IBV_EVENT_QP_FATAL,
    IBV_EVENT_QP_REQ_ERR,
    IBV_EVENT_QP_ACCESS_ERR,
    IBV_EVENT_COMM_EST,
    IBV_EVENT_SQ_DRAINED,
    IBV_EVENT_PATH_MIG,
    IBV_EVENT_PATH_MIG_ERR,
    IBV_EVENT_DEVICE_FATAL,
    IBV_EVENT_PORT_ACTIVE,
    IBV_EVENT_PORT_ERR,
    IBV_EVENT_LID_CHANGE,
    IBV_EVENT_PKEY_CHANGE,
    IBV_EVENT_SM_CHANGE,
    IBV_EVENT_SRQ_ERR,
    IBV_EVENT_SRQ_LIMIT_REACHED,
    IBV_EVENT_QP_LAST_WQE_REACHED,
    IBV_EVENT_CLIENT_REREGISTER,
    IBV_EVENT_GID_CHANGE,
};
```

5.7. Experimental APIs

5.7.1. ibv_exp_query_device

- Template: **int ibv_exp_query_device(struct ibv_context *context, struct ibv_exp_device_attr *attr)**
- Input Parameters:
    - context
- Output Parameters:
    - attr
- Return Value: returns 0 on success, or the value of errno on failure (which indicates the failure reason).

Description:

ibv_exp_query_device returns the attributes of the device with context context. The argument attr is a pointer to an ibv_exp_device_attr struct, as defined in **<infiniband/verbs_exp.h>**.

```
struct ibv_exp_device_attr {
  char      fw_ver[64];
  uint64_t    node_guid;
  uint64_t    sys_image_guid;
  uint64_t    max_mr_size;
  uint64_t    page_size_cap;
  uint32_t    vendor_id;
  uint32_t    vendor_part_id;
  uint32_t    hw_ver;
  int     max_qp;
  int     max_qp_wr;
  int     reserved; /* place holder to align with ibv_device_attr */
  int     max_sge;
  int     max_sge_rd;
 int     max_cq;
  int     max_cqe;
  int     max_mr;
  int     max_pd;
  int     max_qp_rd_atom;
  int     max_ee_rd_atom;
  int     max_res_rd_atom;
```

```
  int     max_qp_init_rd_atom;
  int     max_ee_init_rd_atom;
  enum ibv_exp_atomic_cap exp_atomic_cap;
 int     max_ee;
  int     max_rdd;
  int     max_mw;
  int     max_raw_ipv6_qp;
  int     max_raw_ethy_qp;
  int     max_mcast_grp;
  int     max_mcast_qp_attach;
  int     max_total_mcast_qp_attach;
  int     max_ah;
  int     max_fmr;
  int     max_map_per_fmr;
  int     max_srq;
  int     max_srq_wr;
  int     max_srq_sge;
  uint16_t    max_pkeys;
  uint8_t     local_ca_ack_delay;
  uint8_t     phys_port_cnt;
  uint32_t                comp_mask;
  struct ibv_exp_device_calc_cap calc_cap;
  uint64_t    timestamp_mask;
  uint64_t    hca_core_clock;
  uint64_t    exp_device_cap_flags; /* use ibv_exp_device_cap_flags */
  int     max_dc_req_rd_atom;
  int     max_dc_res_rd_atom;
  int     inline_recv_sz;
  uint32_t    max_rss_tbl_sz;
  struct ibv_exp_ext_atomics_params ext_atom;
  uint32_t                max_mkey_klm_list_size;
  uint32_t                max_send_wqe_inline_klms;
  uint32_t                max_umr_recursion_depth;
  uint32_t                max_umr_stride_dimension;
};
```

5.7.2. ibv_exp_create_qp

- Template: **ibv_exp_create_qp(struct ibv_context *context, struct ibv_exp_qp_init_attr *qp_init_attr)**
- Input Parameters:
  - **context**
- Output Parameters:
  - **qp_init_attr**
- Return Value: Returns a pointer to the created QP, or NULL if the request fails. Check the QP number (qp_num) in the returned QP.

Description:

ibv_exp_create_qp creates a queue pair (QP) associated with the protection domain **pd**. The argument **init_attr** is an **ibv_exp_qp_init_attr** struct, as defined in **<infiniband/verbs_exp.h>**.

```
struct ibv_exp_qp_init_attr {
  void            *qp_context;
  struct ibv_cq        *send_cq;
  struct ibv_cq        *recv_cq;
  struct ibv_srq         *srq;
  struct ibv_qp_cap cap;
  enum ibv_qp_type  qp_type;
  int     sq_sig_all;

  uint32_t    comp_mask; /* use ibv_exp_qp_init_attr_comp_mask */
  struct ibv_pd        *pd;
  struct ibv_xrcd        *xrcd;
  uint32_t    exp_create_flags; /* use ibv_exp_qp_create_flags */

  uint32_t    max_inl_recv;
  struct ibv_exp_qpg  qpg;
  uint32_t    max_atomic_arg;
  uint32_t                max_inl_send_klms;
};
```

5.7.3. ibv_exp_post_send

- Template: **static inline int ibv_exp_post_send(struct ibv_qp *qp, struct ibv_exp_send_wr *wr, struct ibv_exp_send_wr **bad_wr)**
- Input Parameters:

    - `qp`
    - `wr`
- Output Parameters:
    - `bad_wr`
- Return Value: returns 0 on success, or the value of errno on failure (which indicates the failure reason).

Description:

`ibv_exp_post_send` posts the linked list of work requests (WRs) starting with wr to the send queue of the queue pair `qp`. It stops processing WRs from this list at the first failure (that can be detected immediately while requests are being posted), and returns this failing WR through `bad_wr`.

```
struct ibv_exp_send_wr {
  uint64_t    wr_id;
  struct ibv_exp_send_wr *next;
  struct ibv_sge         *sg_list;
  int     num_sge;
  enum ibv_exp_wr_opcode  exp_opcode; /* use ibv_exp_wr_opcode */
  int     reserved; /* place holder to align with ibv_send_wr */
  union {
    uint32_t  imm_data; /* in network byte order */
    uint32_t  invalidate_rkey;
  } ex;
  union {
    struct {
      uint64_t  remote_addr;
      uint32_t  rkey;
    } rdma;
    struct {
      uint64_t  remote_addr;
      uint64_t  compare_add;
      uint64_t  swap;
      uint32_t  rkey;
    } atomic;
    struct {
      struct ibv_ah  *ah;
      uint32_t  remote_qpn;
      uint32_t  remote_qkey;
    } ud;
  } wr;
  union {
    union {
      struct {
        uint32_t    remote_srqn;
      } xrc;
    } qp_type;

    uint32_t    xrc_remote_srq_num;
  };
  union {
    struct {
      uint64_t    remote_addr;
      uint32_t    rkey;
    } rdma;
    struct {
      uint64_t    remote_addr;
      uint64_t    compare_add;
      uint64_t    swap;
      uint32_t    rkey;
    } atomic;
    struct {
      struct ibv_cq *cq;
      int32_t  cq_count;
    } cqe_wait;
    struct {
      struct ibv_qp *qp;
      int32_t  wqe_count;
    } wqe_enable;
  } task;
  union {
    struct {
      enum ibv_exp_calc_op        calc_op;
      enum ibv_exp_calc_data_type data_type;
      enum ibv_exp_calc_data_size data_size;
    } calc;
  } op;
  struct {
    struct ibv_ah   *ah;
    uint64_t        dct_access_key;
    uint32_t        dct_number;
  } dc;
  struct {
    struct ibv_mw     *mw;
    uint32_t      rkey;
    struct ibv_exp_mw_bind_info bind_info;
  } bind_mw;
```

```
  uint64_t  exp_send_flags; /* use ibv_exp_send_flags */
  uint32_t  comp_mask; /* reserved for future growth (must be 0) */
  union {
    struct {
      struct {
        enum mem_layout_type mkey_type;
        union {
          struct ibv_exp_mem_region *mem_reg_list; /* array, size corresponds to wr->num_sge */
          struct {
            struct ibv_exp_mem_repeat_block *mem_repeat_block_list; /* array,  size corresponds to wr->num_sge */
            size_t *repeat_count; /* array size corresponds to ndim */
            uint32_t ndim;
          } rb;
        } mem_list;
        struct non_inline_data *memory_objects; /* used when IBV_EXP_SEND_INLINE is not set */
        int access;
        struct ibv_mr *modified_mr;
        void *region_base_addr;
      } memory_key;
    } umr;
    struct {
      uint32_t        log_arg_sz;
      uint64_t  remote_addr;
      uint32_t  rkey;
      union {
        struct {
          /* For the next four fields:
          * If operand_size <= 8 then inline data is immediate
           * from the corresponding field; for small opernands,
           * ls bits are used.
           * Else the fields are pointers in the process's address space
           * where arguments are stored
           */
          union {
            struct ibv_exp_cmp_swap cmp_swap;
            struct ibv_exp_fetch_add fetch_add;
          } op;
        } inline_data;       /* IBV_EXP_SEND_EXT_ATOMIC_INLINE is set */
        /* in the future add support for non-inline argument provisioning */
      } wr_data;
    } masked_atomics;
  } ext_op;
};
```

For atomic operations, to support atomic responses in big-endian format (the only way to use atomics on NVIDIA® Connect-IB® on little-endian machines) is:

1. Use experimental verbs.
2. Check to see if the atomics capabilities flag **IBV_EXP_ATOMIC_HCA_REPLY_BE** in the exp_atomic_cap field of the struct **ibv_exp_device_attr** returned by **ibv_exp_query_device()**.
3. Set the flag **IBV_EXP_QP_CREATE_ATOMIC_BE_REPLY** when opening the QP. This is what enables the use of atomic ops on Connect-IB.
4. Use the experimental post send verb.

5.6.2.3.1.6 6. RDMA_CM API

This chapter describes the details of the RDMA_CM verbs API.

6.1. Event Channel Operations

6.1.1. rdma_create_event_channel

- Template: **struct rdma_event_channel * rdma_create_event_channel (void)**
- Input Parameters:
    - void no arguments
- Output Parameters:
    - none
- Return Value:
    - A pointer to the created event channel, or NULL if the request fails. On failure, errno will be set to indicate the failure reason.

Description:

Opens an event channel used to report communication events. Asynchronous events are reported to users through event channels.

Event channels are used to direct all events on an `rdma_cm_id`. For many clients, a single event channel may be sufficient, however, when managing a large number of connections or `cm_ids`, users may find it useful to direct events for different `cm_id`s to different channels for processing.

All created event channels must be destroyed by calling `rdma_destroy_event_channel`. Users should call `rdma_get_cm_event` to retrieve events on an event channel.

Each event channel is mapped to a file descriptor. The associated file descriptor can be used and manipulated like any other `fd` to change its behavior. Users may make the `fd` non-blocking, poll or select the `fd`, etc.

See also: `rdma_cm`, `rdma_get_cm_event`, `rdma_destroy_event_channel`.

6.1.2. rdma_destroy_event_channel

- Template: `void rdma_destroy_event_channel (struct rdma_event_channel *channel)`
- Input Parameters:
  - `channel` The communication channel to destroy.
- Output Parameters:
  - none
- Return Value:
  - none

Description: Close an event communication channel. Release all resources associated with an event channel and closes the associated file descriptor.

All `rdma_cm_id`s associated with the event channel must be destroyed, and all returned events must be acked before calling this function.

See Also: `rdma_create_event_channel`, `rdma_get_cm_event`, `rdma_ack_cm_event`.

6.2. Connection Manager (CM) ID Operations

6.2.1. rdma_create_id

Template: `int rdma_create_id(struct rdma_event_channel *channel, struct rdma_cm_id **id, void *context, enum rdma_port_space ps)`

Input Parameters:
`channel` The communication channel that events associated with the allocated `rdma_cm_id` will be reported on.
`id` A reference where the allocated communication identifier will be returned.
`context` User specified context associated with the `rdma_cm_id`.
`ps` RDMA port space.

Output Parameters:
None

Return Value:
0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
Creates an identifier that is used to track communication information.

Notes:

- **rdma_cm_ids** are conceptually equivalent to a socket for RDMA communication. The difference is that RDMA communication requires explicitly binding to a specified RDMA device before communication can occur, and most operations are asynchronous in nature. Communication events on an **rdma_cm_id** are reported through the associated event channel. Users must release the **rdma_cm_id** by calling **rdma_destroy_id**.
- **PORT_SPACES** Details of the services provided by the different port spaces are outlined below.
- **RDMA_PS_TCP** Provides reliable, connection-oriented QP communication. Unlike TCP, the RDMA port space provides message, not stream, based communication.
- **RDMA_PS_UDP** Provides unreliable, connection-less QP communication. Supports both datagram and multicast communication.

See Also:
rdma_cm, **rdma_create_event_channel**, **rdma_destroy_id**, **rdma_get_devices**, **rdma_bind_addr**, **rdma_resolve_addr**, **rdma_connect**, **rdma_listen**, **rdma_set_option**

6.2.2. rdma_destroy_id

Template:
int rdma_destroy_id (struct rdma_cm_id *id)

Input Parameters:
id The communication identifier to destroy.

Output Parameters:
None

Return Value:
0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
Destroys the specified **rdma_cm_id** and cancels any outstanding asynchronous operation.

Notes:
Users must free any associated QP with the **rdma_cm_id** before calling this routine and ack an related events.

See Also:
rdma_create_id, **rdma_destroy_qp**, **rdma_ack_cm_event**

6.2.3. rdma_migrate_id

Template:
`int rdma_migrate_id(struct rdma_cm_id *id, struct rdma_event_channel *channel)`

Input Parameters:
`id` An existing RDMA communication identifier to migrate
`channel` The new event channel for `rdma_cm_id` events

Output Parameters:
None

Return Value:
0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
`rdma_migrate_id` migrates a communication identifier to a different event channel and moves any pending events associated with the `rdma_cm_id` to the new channel.

No polling for events on the `rdma_cm_id` 's current channel nor running of any routines on the `rdma_cm_id` should be done while migrating between channels. `rdma_migrate_id` will block while there are any unacknowledged events on the current event channel.

If the channel parameter is NULL, then the specified rdma_cm_id will be placed into synchronous operation mode. All calls on the id will block until the operation completes.

6.2.4. rdma_set_option

Template:
`int rdma_set_option(struct rdma_cm_id *id, int level, int optname, void *optval, size_t optlen)`

Input Parameters:
`id` RDMA communication identifier
`level` Protocol level of the option to set
`optname` Name of the option to set
`optval` Reference to the option data
`optlen` The size of the option data (optval) buffer

Output Parameters:
None

Return Value:
0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
`rdma_set_option` sets communication options for an rdma_cm_id. Option levels and details may be found in the enums in the relevant header files.

6.2.5. rdma_create_ep

Template:
`int rdma_create_ep( struct rdma_cm_id **id, struct rdma_addrinfo *res, struct ibv_pd *pd, struct ibv_qp_init_attr *qp_init_attr );`

Input Parameters:
`id` A reference where the allocated communication identifier will be returned.

res Address information associated with the **rdma_cm_id** (usually returned from a prior call to **rdma_getaddrinfo**).
pd Optional protection domain if a Queue Pair (QP) is going to be associated with the **rdma_cm_id**.
qp_init_attr Optional initial QP attributes

Output Parameters:
id The communication identifier is returned to the user through this reference

Return Value:
0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure

Description:
rdma_create_ep creates an identifier and an optional Queue Pair (QP) used to track communication information.

If **qp_init_attr** is not **NULL**, then a QP will be automatically allocated and associated with the new **rdma_cm_id** (**id**).

- If a protection domain (**pd**) is provided, then the QP will be created securely on that specific PD.
- Otherwise, the QP will be allocated on a default PD.

By default, the newly created **rdma_cm_id** will be set to use synchronous operations (meaning functions like **connect**, **listen**, and **get_request** will block the program until they finish). To use asynchronous operations instead, the **rdma_cm_id** must be migrated to a user-allocated event channel using the **rdma_migrate_id** function.

The **rdma_cm_id** must be properly released after use by calling the **rdma_destroy_ep** function.

As part of the **res** input parameter, the function relies on the **rdma_addrinfo** structure, which is defined as follows:

```
struct rdma_addrinfo {
    int                     ai_flags;
    int                     ai_family;
    int                     ai_qp_type;
    int                     ai_port_space;
    socklen_t               ai_src_len;
    socklen_t               ai_dst_len;
    struct sockaddr        *ai_src_addr;
    struct sockaddr        *ai_dst_addr;
    char                   *ai_src_canonname;
    char                   *ai_dst_canonname;
    size_t                  ai_route_len;
    void                   *ai_route;
    size_t                  ai_connect_len;
    void                   *ai_connect;
    struct rdma_addrinfo   *ai_next;
};
```

Field details:

- **ai_flags** Hint flags which control the operation. Supported flags are **RAI_PASSIVE**, **RAI_NUMERICHOST**, and **RAI_NOROUTE**.
- **ai_family** Address family for the source and destination address (e.g., **AF_INET**, **AF_INET6**, **AF_IB**).
- **ai_qp_type** The type of RDMA Queue Pair used.
- **ai_port_space** RDMA port space used (e.g., **RDMA_PS_UDP** or **RDMA_PS_TCP**).

- **ai_src_len** Length of the source address referenced by **ai_src_addr**.
- **ai_dst_len** Length of the destination address referenced by **ai_dst_addr**.
- **ai_src_addr** Address of local RDMA device, if provided.
- **ai_dst_addr** Address of destination RDMA device, if provided.
- **ai_src_canonname** The canonical name for the source.
- **ai_dst_canonname** The canonical name for the destination.
- **ai_route_len** Size of the routing information buffer referenced by **ai_route**.
- **ai_route** Routing information for RDMA transports that require routing data as part of connection establishment.
- **ai_connect_len** Size of connection information referenced by **ai_connect**.
- **ai_connect** Data exchanged as part of the connection establishment process.
- **ai_next** Pointer to the next **rdma_addrinfo** structure in the list.

6.2.6. rdma_destroy_ep

Template:
int rdma_destroy_ep (struct rdma_cm_id *id)

Input Parameters:
id The communication identifier to destroy

Output Parameters:
None

Return Value:
0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure

Description:
rdma_destroy_ep destroys the specified **rdma_cm_id** and all associated resources, including QPs associated with the **id**.

6.2.7. rdma_resolve_addr

Template:
int rdma_resolve_addr (struct rdma_cm_id *id, struct sockaddr *src_addr, struct sockaddr *dst_addr, int timeout_ms)

Input Parameters:
id RDMA identifier.
src_addr Source address information. This parameter may be NULL.
dst_addr Destination address information.
timeout_ms Time to wait for resolution to complete.

Output Parameters:
None

Return Value:
0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_resolve_addr resolves destination and optional source addresses from IP addresses to an RDMA address. If successful, the specified rdma_cm_id will be bound to a local device.

Notes:
This call is used to map a given destination IP address to a usable RDMA address. The IP to RDMA address mapping is done using the local routing tables, or via ARP. If a source address is given, the **rdma_cm_id** is bound to that address, the same as if **rdma_bind_addr** were called. If no source address is given, and the **rdma_cm_id** has not yet been bound to a device, then the **rdma_cm_id** will be bound to a source address based on the local routing tables. After this call, the **rdma_cm_id** will be bound to an RDMA device. This call is typically made from the active side of a connection before calling **rdma_resolve_route** and **rdma_connect**.

InfiniBand-specific note: This call maps the destination and, if given, source IP addresses to GIDs. In order to perform the mapping, IPoIB must be running on both the local and remote nodes.

See Also:
rdma_create_id, **rdma_resolve_route**, **rdma_connect**, **rdma_create_qp**, **rdma_get_cm_event**, **rdma_bind_addr**, **rdma_get_src_port**, **rdma_get_dst_port**, **rdma_get_local_addr**, **rdma_get_peer_addr**

6.2.8. rdma_bind_addr

Template:
int rdma_bind_addr (struct rdma_cm_id *id, struct sockaddr *addr)

Input Parameters:
id RDMA identifier.
addr Local address information. Wildcard values are permitted.

Output Parameters:
None

Return Value:
0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_bind_addr associates a source address with an **rdma_cm_id**. The address may be wild carded. If binding to a specific local address, the **rdma_cm_id** will also be bound to a local RDMA device.

Notes:
Typically, this routine is called before calling **rdma_listen** to bind to a specific port number, but it may also be called on the active side of a connection before calling **rdma_resolve_addr** to bind to a specific address. If used to bind to port 0, the **rdma_cm** will select an available port, which can be retrieved with **rdma_get_src_port**.

See Also:
rdma_create_id, **rdma_listen**, **rdma_resolve_addr**, **rdma_create_qp**, **rdma_get_local_addr**, **rdma_get_src_port**

6.2.9. rdma_resolve_route

Template:
int rdma_resolve_route (struct rdma_cm_id *id, int timeout_ms)

Input Parameters:
id RDMA identifier.
addr Local address information. Wildcard values are permitted.

Output Parameters:
None

Return Value:
0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_resolve_route resolves an RDMA route to the destination address in order to establish a connection. The destination must already have been resolved by calling **rdma_resolve_addr** . Thus, this function is called on the client side after **rdma_resolve_addr** but before calling **rdma_connect** . For InfiniBand connections, the call obtains a path record which is used by the connection.

6.2.10. rdma_listen

Template:
int rdma_listen(struct rdma_cm_id *id, int backlog)

Input Parameters:
id RDMA communication identifier
backlog The backlog of incoming connection requests

Output Parameters:
None

Return Value:
0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_listen initiates a listen for incoming connection requests or datagram service lookup. The listen is restricted to the locally bound source address.
Please note that the **rdma_cm_id** must already have been bound to a local address by calling **rdma_bind_addr** before calling **rdma_listen** . If the **rdma_cm_id** is bound to a specific IP address, the listen will be restricted to that address and the associated RDMA device. If the **rdma_cm_id** is bound to an RDMA port number only, the listen will occur across all RDMA devices.

6.2.11. rdma_connect

Template:
int rdma_connect(struct rdma_cm_id *id, struct rdma_conn_param *conn_param)

Input Parameters:
id RDMA communication identifier conn_paramOptional connection parameters

Output Parameters:
none

Return Value:
0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_connect initiates an active connection request. For a connected **rdma_cm_id** , **id** , the call initiates a connection request to a remote destination. or an unconnected **rdma_cm_id** , it initiates a lookup of the remote QP providing the datagram service. The user must already have resolved a route to the destination address by having called **rdma_resolve_route** or **rdma_create_ep** before calling this method.

For InfiniBand specific connections, the QPs are configured with minimum RNR NAK timer and local ACK values. The minimum RNR NAK timer value is set to 0, for a delay of 655 ms. The local ACK timeout is calculated based on the packet lifetime and local HCA ACK delay. The packet lifetime is determined by the InfiniBand Subnet Administrator and is part of the resolved route (path record) information. The HCA ACK delay is a property of the locally used HCA. Retry count and RNR retry count values are 3-bit values.
Connections established over iWarp RDMA devices currently require that the active side of the connection send the first message.

Struct **rdma_conn_param** is defined as follows:

```
struct rdma_conn_param {
const void *private_data; uint8_t private_data_len; uint8_t responder_resources; uint8_t initiator_depth; uint8_t
flow_control;
uint8_t retry_count;ignored when accepting uint8_t rnr_retry_count;
uint8_t srq;ignored if QP created on the rdma_cm_id
uint32_t qp_num;ignored if QP created on the rdma_cm_id
};
```

The following is a more detailed description of the **rdma_conn_param** structure members:

- **private_data** References a user-controlled data buffer. The contents of the buffer are copied and transparently passed to the remote side as part of the communication request. May be NULL if **private_data** is not required.
- **private_data_len** Specifies the size of the user-controlled data buffer. Note that the actual amount of data transferred to the remote side is transport dependent and may be larger than that requested.
- **responder_resources** The maximum number of outstanding RDMA read and atomic operations that the local side will accept from the remote side. Applies only to **RDMA_PS_TCP** . This value must be less than or equal to the local RDMA device attribute **max_qp_rd_atom** and remote RDMA device attribute **max_qp_init_rd_atom** . The remote endpoint can adjust this value when accepting the connection.
- **initiator_depth** The maximum number of outstanding RDMA read and atomic operations that the local side will have to the remote side. Applies only to **RDMA_PS_TCP** . This value must be less than or equal to the local RDMA device attribute **max_qp_init_rd_atom** and remote RDMA device attribute **max_qp_rd_atom** . The remote endpoint can adjust this value when accepting the connection.
- **flow_control** Specifies if hardware flow control is available. This value is exchanged with the remote peer and is not used to configure the QP. Applies only to **RDMA_PS_TCP** .
- **retry_count** The maximum number of times that a data transfer operation should be retried on the connection when an error occurs. This setting controls the number of times

to retry send, RDMA, and atomic operations when timeouts occur. Applies only to **RDMA_PS_TCP**.

- **rnr_retry_count** The maximum number of times that a send operation from the remote peer should be retried on a connection after receiving a receiver not ready (RNR) error. RNR errors are generated when a send request arrives before a buffer has been posted to receive the incoming data. Applies only to **RDMA_PS_TCP**.
- **srq** Specifies if the QP associated with the connection is using a shared receive queue. This field is ignored by the library if a QP has been created on the **rdma_cm_id**. Applies only to **RDMA_PS_TCP**.
- **qp_num** Specifies the QP number associated with the connection. This field is ignored by the library if a QP has been created on the **rdma_cm_id**. Applies only to **RDMA_PS_TCP**.

6.2.12. rdma_get_request

Template:
int rdma_get_request (struct rdma_cm_id *listen, struct rdma_cm_id **id)

Input Parameters:
listen Listening **rdma_cm_id**
id **rdma_cm_id** associated with the new connection

Output Parameters:
id A pointer to **rdma_cm_id** associated with the request

Return Value:
0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_get_request retrieves the next pending connection request event. The call may only be used on listening **rdma_cm_ids** operating synchronously. If the call is successful, a new **rdma_cm_id** (id) representing the connection request will be returned to the user. The new **rdma_cm_id** will reference event information associated with the request until the user calls **rdma_reject**, **rdma_accept**, or **rdma_destroy_id** on the newly created identifier. For a description of the event data, see **rdma_get_cm_event**.
If QP attributes are associated with the listening endpoint, the returned **rdma_cm_id** will also reference an allocated QP.

6.2.13. rdma_accept

Template:
int rdma_accept(struct rdma_cm_id *id, struct rdma_conn_param *conn_param);

Input Parameters:
id RDMA communication identifier
conn_param Optional connection parameters (these parameters are described in detail under the **rdma_connect** function documentation)

Output Parameters:
None

Return Value:
0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

rdma_accept is called from the listening (server) side to accept an incoming connection or a datagram service lookup request.

Unlike the standard TCP/IP socket **accept()** routine, **rdma_accept** is not called directly on the listening **rdma_cm_id**.

Instead, the process works like this:

1. The user starts listening by calling **rdma_listen**.
2. The user then waits for an **RDMA_CM_EVENT_CONNECT_REQUEST** event to occur.
3. When a connection request event arrives, it automatically hands the user a newly created **rdma_cm_id** (similar to how standard sockets give you a new socket file descriptor upon connection). This new **rdma_cm_id** is already securely bound to a specific RDMA device.
4. Finally, **rdma_accept** is called on this new **rdma_cm_id** to fully establish the connection, rather than the original listening ID.

6.2.14. rdma_reject

Template:
int rdma_reject(struct rdma_cm_id *id, const void *private_data, uint8_t private_data_len)

Input Parameters:
id RDMA communication identifier
private_data Optional private data to send with the reject message
private_data_len Size (in bytes) of the private data being sent

Output Parameters:
None

Return Value:
0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_reject is called from the listening side to reject a connection or datagram service lookup request.
After receiving a connection request event, a user may call **rdma_reject** to reject the request. The optional private data will be passed to the remote side if the underlying RDMA transport supports private data in the reject message.

6.2.15. rdma_notify

Template:
int rdma_notify(struct rdma_cm_id *id, enum ibv_event_type event)

Input Parameters:
id RDMA communication identifier
event Asynchronous event

Output Parameters:
None

Return Value:
0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_notify is used to notify the librdmacm of asynchronous events which have occurred on a QP associated with the **rdma_cm_id**, **id**.
Asynchronous events that occur on a QP are reported through the user's device event handler. This routine is used to notify the librdmacm of communication events. In most cases, use of this routine is not necessary, however if connection establishment is done out of band (such as done through InfiniBand), it is possible to receive data on a QP that is not yet considered connected. This routine forces the connection into an established state in this case in order to handle the rare situation where the connection never forms on its own. Calling this routine ensures the delivery of the **RDMA_CM_EVENT_ESTABLISHED** event to the application. Events that should be reported to the CM are: **IB_EVENT_COMM_EST**.

6.2.16. rdma_disconnect

Template:
int rdma_disconnect(struct rdma_cm_id *id)

Input Parameters:
id RDMA communication identifier

Output Parameters:
None

Return Value:
0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_disconnect disconnects a connection and transitions any associated QP to the error state. This action will result in any posted work requests being flushed to the completion queue. **rdma_disconnect** may be called by both the client and server side of the connection. After successfully disconnecting, an **RDMA_CM_EVENT_DISCONNECTED** event will be generated on both sides of the connection.

6.2.17. rdma_get_src_port

Template:
uint16_t rdma_get_src_port(struct rdma_cm_id *id)

Input Parameters:
id RDMA communication identifier

Output Parameters:
None

Return Value:
Returns the 16-bit port number associated with the local endpoint of 0 if the **rdma_cm_id**, **id**, is not bound to a port

Description:
`rdma_get_src_port` retrieves the local port number for an `rdma_cm_id` (id) which has been bound to a local address. If the id is not bound to a port, the routine will return 0.

6.2.18. rdma_get_dst_port

Template:
`uint16_t rdma_get_dst_port(struct rdma_cm_id *id)`

Input Parameters:
`id` RDMA communication identifier

Output Parameters:
None

Return Value:
Returns the 16-bit port number associated with the peer endpoint of 0 if the `rdma_cm_id`, `id`, is not connected

Description:
`rdma_get_dst_port` retrieves the port associated with the peer endpoint. If the `rdma_cm_id`, `id`, is not connected, then the routine will return 0.

6.2.19. rdma_get_local_addr

Template:
`struct sockaddr {}rdma_get_local_addr{*}(struct rdma_cm_id *id)`

Input Parameters:
`id` RDMA communication identifier

Output Parameters:
None

Return Value:
Returns a pointer to the local sockaddr address of the `rdma_cm_id`, `id`. If the id is not bound to an address, then the contents of the sockaddr structure will be set to all zeros

Description:
`rdma_get_local_addr` retrieves the local IP address for the `rdma_cm_id` which has been bound to a local device.

6.2.20. rdma_get_peer_addr

Template:
`struct sockaddr * rdma_get_peer_addr (struct rdma_cm_id *id)`

Input Parameters:
`id` RDMA communication identifier

Output Parameters:
None

Return Value:
A pointer to the sockaddr address of the connected peer. If the **rdma_cm_id** is not connected, then the contents of the sockaddr structure will be set to all zeros.

Description:
rdma_get_peer_addr retrieves the remote IP address of a bound **rdma_cm_id**.

6.2.21. rdma_get_devices

Template:
struct ibv_context ** rdma_get_devices (int *num_devices)

Input Parameters:
num_devices If non-NULL, set to the number of devices returned

Output Parameters:
num_devices Number of RDMA devices currently available

Return Value:
Array of available RDMA devices on success or NULL if the request fails

Description:
rdma_get_devices retrieves an array of RDMA devices currently available. Devices remain opened while librdmacm is loaded and the array must be released by calling **rdma_free_devices**.

6.2.22. rdma_free_devices

Template:
void rdma_free_devices (struct ibv_context **list)

Input Parameters:
list List of devices returned from **rdma_get_devices**.

Output Parameters:
None

Return Value:
None

Description:
rdma_free_devices frees the device array returned by the **rdma_get_devices** routine.

6.2.23. rdma_getaddrinfo

Template:
int rdma_getaddrinfo(char *node, char *service, struct rdma_addrinfo *hints, struct rdma_addrinfo **res)

Input Parameters:
nodeOptional name, dotted-decimal IPv4 or IPv6 hex address to resolve
service The service name or port number of the address
hints Reference to an **rmda_addrinfo** structure containing hints about the type of service the caller supports
res A pointer to a linked list of **rdma_addrinfo** structures containing response information

Output Parameters:
res An **rdma_addrinfo** structure which returns information needed to establish communication

Return Value:
0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_getaddrinfo provides transport independent address translation. It resolves the destination node and service address and returns information required to establish device communication. It is the functional equivalent of **getaddrinfo**.
Please note that either node or service must be provided. If hints are provided, the operation will be controlled by **hints.ai_flags**. If **RAI_PASSIVE** is specified, the call will resolve address information for use on the passive side of a connection.
The **rdma_addrinfo** structure is described under the **rdma_create_ep** routine.

6.2.24. rdma_freeaddrinfo

Template:
void rdma_freeaddrinfo(struct rdma_addrinfo *res)

Input Parameters:
res The **rdma_addrinfo** structure to free

Output Parameters:
None

Return Value:
None

Description:
rdma_freeaddrinfo releases the **rdma_addrinfo** (**res**) structure returned by the **rdma_getaddrinfo** routine. Note that if **ai_next** is not NULL, **rdma_freeaddrinfo** will free the entire list of addrinfo structures.

6.2.25. rdma_create_qp

Template:
int rdma_create_qp (struct rdma_cm_id *id, struct ibv_pd *pd, struct ibv_qp_init_attr *qp_init_attr)

Input Parameters:
id RDMA identifier.
pd protection domain for the QP.
qp_init_attr initial QP attributes.

Output Parameters:
qp_init_attr The actual capabilities and properties of the created QP are returned through this structure

Return Value:
0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_create_qp allocates a QP associated with the specified **rdma_cm_id** and transitions it for

sending and receiving. The actual capabilities and properties of the created QP will be returned to the user through the **qp_init_attr** parameter.

Notes:
The **rdma_cm_id** must be bound to a local RDMA device before calling this function, and the protection domain must be for that same device. QPs allocated to an **rdma_cm_id** are automatically transitioned by the librdmacm through their states. After being allocated, the QP will be ready to handle posting of receives. If the QP is unconnected, it will be ready to post sends.

See Also:
rdma_bind_addr, **rdma_resolve_addr**, **rdma_destroy_qp**, **ibv_create_qp**, **ibv_modify_qp**.

6.2.26. rdma_destroy_qp

Template:
void rdma_destroy_qp (struct rdma_cm_id *id)

Input Parameters:
id RDMA identifier.

Output Parameters:
none

Return Value:
none

Description:
rdma_destroy_qp destroys a QP allocated on the **rdma_cm_id**.

Notes:
Users must destroy any QP associated with an **rdma_cm_id** before destroying the ID.

See Also:
rdma_create_qp, **rdma_destroy_id**, **ibv_destroy_qp**.

6.2.27. rdma_join_multicast

Template:
int rdma_join_multicast (struct rdma_cm_id *id, struct sockaddr *addr, void *context)

Input Parameters:
id Communication identifier associated with the request.
addr Multicast address identifying the group to join.
context User-defined context associated with the join request.

Output Parameters:
none

Return Value:
0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_join_multicast joins a multicast group and attaches an associated QP to the group.

Notes:
Before joining a multicast group, the **rdma_cm_id** must be bound to an RDMA device by calling **rdma_bind_addr** or **rdma_resolve_addr**. Use of **rdma_resolve_addr** requires the local routing tables to resolve the multicast address to an RDMA device, unless a specific source address is provided. The user must call **rdma_leave_multicast** to leave the multicast group and release any multicast resources. After the join operation completes, any associated QP is automatically attached to the multicast group, and the join context is returned to the user through the **private_data** field in the **rdma_cm_event**.

See Also:
rdma_leave_multicast, **rdma_bind_addr**, **rdma_resolve_addr**, **rdma_create_qp**, **rdma_get_cm_event**.

6.2.28. rdma_leave_multicast

Template:
int rdma_leave_multicast (struct rdma_cm_id *id, struct sockaddr *addr)

Input Parameters:
id Communication identifier associated with the request.
addr Multicast address identifying the group to leave.

Output Parameters:
none

Return Value:
0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_leave_multicast leaves a multicast group and detaches an associated QP from the group.

Notes:
Calling this function before a group has been fully joined results in canceling the join operation. Users should be aware that messages received from the multicast group may stilled be queued for completion processing immediately after leaving a multicast group. Destroying an **rdma_cm_id** will automatically leave all multicast groups.

See Also:
rdma_join_multicast, **rdma_destroy_qp**.

6.3. RDMA_CM Event Handling Operations

6.3.1. rdma_get_cm_event

Template:
int rdma_get_cm_event (struct rdma_event_channel *channel, struct rdma_cm_event **event)

Input Parameters:
channel Event channel to check for events.
event Allocated information about the next communication event.

Output Parameters:
none

Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description: Retrieves a communication event. If no events are pending, by default, the call will block until an event is received.

> The default synchronous behavior of this routine can be changed by modifying the file descriptor associated with the given channel. All events that are reported must be acknowledged by calling **rdma_ack_cm_event**. Destruction of an **rdma_cm_id** will block until related events have been acknowledged.

6.3.2. Event Data

Communication event details are returned in the **rdma_cm_event** structure. This structure is allocated by the rdma_cm and released by the **rdma_ack_cm_event** routine. Details of the **rdma_cm_event** structure are given below.

- **id** The **rdma_cm** identifier associated with the event. If the event type is **RDMA_CM_EVENT_CONNECT_REQUEST**, then this references a new ID for that communication.
- **listen_id** For **RDMA_CM_EVENT_CONNECT_REQUEST** event types, this references the corresponding listening request identifier.
- **event** Specifies the type of communication event which occurred. See EVENT TYPES below.
- **status** Returns any asynchronous error information associated with an event. The status is zero unless the corresponding operation failed.
- **param** Provides additional details based on the type of event. Users should select the conn or UD subfields based on the **rdma_port_space** of the **rdma_cm_id** associated with the event. See UD EVENT DATA and CONN EVENT DATA below.

6.3.3. UD Event Data

Event parameters related to unreliable datagram (UD) services:

- **RDMA_PS_UDP** and **RDMA_PS_IPOIB**. The UD event data is valid for **RDMA_CM_EVENT_ESTABLISHED** and **RDMA_CM_EVENT_MULTICAST_JOIN** events, unless stated otherwise.
- **private_data** References any user-specified data associated with **RDMA_CM_EVENT_CONNECT_REQUEST** or **RDMA_CM_EVENT_ESTABLISHED** events. The data referenced by this field matches that specified by the remote side when calling **rdma_connect** or **rdma_accept**. This field is NULL if the event does not include private data. The buffer referenced by this pointer is deallocated when calling **rdma_ack_cm_event**.
- **private_data_len** The size of the private data buffer. Users should note that the size of the private data buffer may be larger than the amount of private data sent by the remote side. Any additional space in the buffer will be zeroed out.
- **ah_attr** Address information needed to send data to the remote endpoint(s). Users should use this structure when allocating their address handle.
- **qp_num** QP number of the remote endpoint or multicast group.

- **qkey** QKey needed to send data to the remote endpoint(s).

6.3.4. Conn Event Data

Event parameters related to connected QP services: **RDMA_PS_TCP**. The connection related event data is valid for **RDMA_CM_EVENT_CONNECT_REQUEST** and **RDMA_CM_EVENT_ESTABLISHED** events, unless stated otherwise.

- **private_data** References any user-specified data associated with the event. The data referenced by this field matches that specified by the remote side when calling **rdma_connect** or **rdma_accept**. This field is NULL if the event does not include private data. The buffer referenced by this pointer is deallocated when calling **rdma_ack_cm_event**.
- **private_data_len** The size of the private data buffer. Users should note that the size of the private data buffer may be larger than the amount of private data sent by the remote side. Any additional space in the buffer will be zeroed out.
- **responder_resources** The number of responder resources requested of the recipient. This field matches the initiator depth specified by the remote node when calling **rdma_connect** and **rdma_accept**.
- **initiator_depth** The maximum number of outstanding RDMA read/atomic operations that the recipient may have outstanding. This field matches the responder resources specified by the remote node when calling **rdma_connect** and **rdma_accept**.
- **flow_control** Indicates if hardware level flow control is provided by the sender.
- **retry_count** For **RDMA_CM_EVENT_CONNECT_REQUEST** events only, indicates the number of times that the recipient should retry send operations.
- **rnr_retry_count** The number of times that the recipient should retry receiver not ready (RNR) NACK errors.
- **srq** Specifies if the sender is using a shared-receive queue.
- **qp_num** Indicates the remote QP number for the connection.

6.3.5. Event Types

The following types of communication events may be reported.

- **RDMA_CM_EVENT_ADDR_RESOLVED** Address resolution (**rdma_resolve_addr**) completed successfully.
- **RDMA_CM_EVENT_ADDR_ERROR** Address resolution (**rdma_resolve_addr**) failed.
- **RDMA_CM_EVENT_ROUTE_RESOLVED** Route resolution (**rdma_resolve_route**) completed successfully.
- **RDMA_CM_EVENT_ROUTE_ERROR** Route resolution (**rdma_resolve_route**) failed.
- **RDMA_CM_EVENT_CONNECT_REQUEST** Generated on the passive side to notify the user of a new connection request.
- **RDMA_CM_EVENT_CONNECT_RESPONSE** Generated on the active side to notify the user of a successful response to a connection request. It is only generated on **rdma_cm_id**s that do not have a QP associated with them.
- **RDMA_CM_EVENT_CONNECT_ERROR** Indicates that an error has occurred trying to establish or a connection. May be generated on the active or passive side of a connection.

- **RDMA_CM_EVENT_UNREACHABLE** Generated on the active side to notify the user that the remote server is not reachable or unable to respond to a connection request.
- **RDMA_CM_EVENT_REJECTED** Indicates that a connection request or response was rejected by the remote end point.
- **RDMA_CM_EVENT_ESTABLISHED** Indicates that a connection has been established with the remote end point.
- **RDMA_CM_EVENT_DISCONNECTED** The connection has been disconnected.
- **RDMA_CM_EVENT_DEVICE_REMOVAL** The local RDMA device associated with the **rdma_cm_id** has been removed. Upon receiving this event, the user must destroy the related **rdma_cm_id** .
- **RDMA_CM_EVENT_MULTICAST_JOIN** The multicast join operation (**rdma_join_multicast**) completed successfully.
- **RDMA_CM_EVENT_MULTICAST_ERROR** An error either occurred joining a multicast group, or, if the group had already been joined, on an existing group. The specified multicast group is no longer accessible and should be rejoined, if desired.
- **RDMA_CM_EVENT_ADDR_CHANGE** The network device associated with this ID through address resolution changed its HW address, eg following of bonding failover. This event can serve as a hint for applications who want the links used for their RDMA sessions to align with the network stack.
- **RDMA_CM_EVENT_TIMEWAIT_EXIT** The QP associated with a connection has exited its timewait state and is now ready to be re-used. After a QP has been disconnected, it is maintained in a timewait state to allow any in flight packets to exit the network. After the timewait state has completed, the **rdma_cm** will report this event.

See Also: **rdma_ack_cm_event** , **rdma_create_event_channel** , **rdma_resolve_addr** , **rdma_resolve_route** , **rdma_connect** , **rdma_listen** , **rdma_join_multicast** , **rdma_destroy_id** , **rdma_event_str** .

6.3.6. rdma_ack_cm_event

Template:
int rdma_ack_cm_event (struct rdma_cm_event *event)

Input Parameters:
event Event to be released.

Output Parameters:
none

Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_ack_cm_event frees a communication event. All events which are allocated by **rdma_get_cm_event** must be released, there should be a one-to-one correspondence between successful gets and acks. This call frees the event structure and any memory that it references.

See Also: **rdma_get_cm_event** , **rdma_destroy_id** .

6.3.7. rdma_event_str

Template:
`char {}rdma_event_str* (enum rdma_cm_event_type event)`

Input Parameters:
event Asynchronous event.

Output Parameters:
none

Return Value: A pointer to a static character string corresponding to the event

Description:
rdma_event_str returns a string representation of an asynchronous event.

See Also: **rdma_get_cm_event** .

5.6.2.3.1.7 7. RDMA Verbs API

This chapter describes the details of the RDMA verbs API.

7.1. RDMA Protection Domain Operations

7.1.1. rdma_reg_msgs

Template:
`struct ibv_mr {}rdma_reg_msgs{*}(struct rdma_cm_id *id, void *addr, size_t length)`

Input Parameters:
id A reference to the communication identifier where the message buffer(s) will be used
addr The address of the memory buffer(s) to register
length The total length of the memory to register

Output Parameters:
ibv_mr A reference to an **ibv_mr** struct of the registered memory region

Return Value: A reference to the registered memory region on success or NULL on failure

Description:
rdma_reg_msgs registers an array of memory buffers for sending or receiving messages or for RDMA operations. The registered memory buffers may then be posted to an **rdma_cm_id** using **rdma_post_send** or **rdma_post_recv** . They may also be specified as the target of an RDMA read operation or the source of an RDMA write request.
The memory buffers are registered with the protection domain associated with the **rdma_cm_id** .
The start of the data buffer array is specified through the addr parameter and the total size of the array is given by the length.
All data buffers must be registered before being posted as a work request. They must be deregistered by calling **rdma_dereg_mr** .

7.1.2. rdma_reg_read

Template:
`struct ibv_mr * rdma_reg_read(struct rdma_cm_id *id, void *addr, size_t length)`

Input Parameters:
id A reference to the communication identifier where the message buffer(s) will be used.
addr The address of the memory buffer(s) to register.
length The total length of the memory to register.

Output Parameters:
ibv_mr A reference to an **ibv_mr** struct of the registered memory region.

Return Value: A reference to the registered memory region on success or NULL on failure. If an error occurs, errno will be set to indicate the failure reason.

Description:

rdma_reg_read Registers a memory buffer that will be accessed by a remote RDMA read operation. Memory buffers registered using **rdma_reg_read** may be targeted in an RDMA read request, allowing the buffer to be specified on the remote side of an RDMA connection as the **remote_addr** of **rdma_post_read**, or similar call.
rdma_reg_read is used to register a data buffer that will be the target of an RDMA read operation on a queue pair associated with an **rdma_cm_id**. The memory buffer is registered with the protection domain associated with the identifier. The start of the data buffer is specified through the addr parameter, and the total size of the buffer is given by length.
All data buffers must be registered before being posted as work requests. Users must deregister all registered memory by calling the **rdma_dereg_mr**.

See Also **rdma_cm**, **rdma_create_id**, **rdma_create_ep**, **rdma_reg_msgs**, **rdma_reg_write**, **ibv_reg_mr**, **ibv_dereg_mr**, **rdma_post_read**.

7.1.3. rdma_reg_write

Template:
struct ibv_mr {}rdma_reg_write{*}(struct rdma_cm_id *id, void *addr, size_t length)

Input Parameters:
id A reference to the communication identifier where the message buffer(s) will be used.
addr The address of the memory buffer(s) to register.
length The total length of the memory to register.

Output Parameters:
ibv_mr A reference to an **ibv_mr** struct of the registered memory region.

Return Value: A reference to the registered memory region on success or NULL on failure. If an error occurs, errno will be set to indicate the failure reason.

Description:
rdma_reg_write registers a memory buffer which will be accessed by a remote RDMA write operation. Memory buffers registered using this routine may be targeted in an RDMA write request, allowing the buffer to be specified on the remote side of an RDMA connection as the **remote_addr** of an **rdma_post_write** or similar call.
The memory buffer is registered with the protection domain associated with the **rdma_cm_id**. The start of the data buffer is specified through the addr parameter, and the total size of the buffer is given by the length.

All data buffers must be registered before being posted as work requests. Users must deregister all registered memory by calling the **rdma_dereg_mr**.

See Also: **rdma_cm**, **rdma_create_id**, **rdma_create_ep**, **rdma_reg_msgs**, **rdma_reg_read**, **ibv_reg_mr**, **ibv_dereg_mr**, **rdma_post_write**.

7.1.4. rdma_dereg_mr

Template:
int rdma_dereg_mr(struct ibv_mr *mr)

Input Parameters:
mr A reference to a registered memory buffer

Output Parameters:
None

Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_dereg_mr deregisters a memory buffer which has been registered for RDMA or message operations. This routine must be called for all registered memory associated with a given **rdma_cm_id** before destroying the **rdma_cm_id**.

7.1.5. rdma_create_srq

Template:
int rdma_create_srq(struct rdma_cm_id *id, struct ibv_pd *pd, struct ibv_srq_init_attr *attr)

Input Parameters:
id The RDMA communication identifier
pd Optional protection domain for the shared request queue (SRQ)
attr Initial SRQ attributes

Output Parameters:
attr The actual capabilities and properties of the created SRQ are returned through this structure

Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_create_srq allocates a shared request queue associated with the **rdma_cm_id**, **id**. The **id** must be bound to a local RMDA device before calling this routine. If the protection domain, **pd**, is provided, it must be for that same device. After being allocated, the SRQ will be ready to handle posting of receives. If a **pd** is NULL, then the **rdma_cm_id** will be created using a default protection domain. One default protection domain is allocated per RDMA device. The initial SRQ attributes are specified by the **attr** parameter.
If a completion queue, CQ, is not specified for the XRC SRQ, then a CQ will be allocated by the **rdma_cm** for the SRQ, along with corresponding completion channels. Completion channels and CQ data created by the **rdma_cm** are exposed to the user through the **rdma_cm_id** structure. The actual capabilities and properties of the created SRQ will be returned to the user through the attr

parameter.
An **rdma_cm_id** may only be associated with a single SRQ.

7.1.6. rdma_destroy_srq

Template:
void rdma_destroy_srq(struct rdma_cm_id *id)

Input Parameters:
id The RDMA communication identifier whose associated SRQ we wish to destroy.

Output Parameters:
None

Return Value: None

Description:
rdma_destroy_srq destroys an SRQ allocated on the **rdma_cm_id**, **id**. Any SRQ associated with an **rdma_cm_id** must be destroyed before destroying the **rdma_cm_id**, **id**.

7.2. RDMA Active Queue Pair Operations

7.2.1. rdma_post_recvv

Template:
int rdma_post_recvv(struct rdma_cm_id *id, void *context, struct ibv_sge *sgl, int nsge)

Input Parameters:
id A reference to the communication identifier where the message buffer(s) will be posted
context A user-defined context associated with the request
sgl A scatter-gather list of memory buffers posted as a single request
nsge The number of scatter-gather entries in the sgl array

Output Parameters:
None

Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_post_recvv posts a single work request to the receive queue of the queue pair associated with the **rdma_cm_id**, **id**. The posted buffers will be queued to receive an incoming message sent by the remote peer.
Please note that this routine supports multiple scatter-gather entries. The user is responsible for ensuring that the receive is posted, and the total buffer space is large enough to contain all sent data before the peer posts the corresponding send message. The message buffers must have been registered before being posted, and the buffers must remain registered until the receive completes.
Messages may be posted to an **rdma_cm_id** only after a queue pair has been associated with it. A queue pair is bound to an **rdma_cm_id** after calling **rdma_create_ep** or **rdma_create_qp**, if the **rdma_cm_id** is allocated using **rdma_create_id**.
The user-defined context associated with the receive request will be returned to the user through the work completion work request identifier (**wr_id**) field.

7.2.2. rdma_post_sendv

Template:
int rdma_post_sendv(struct rdma_cm_id *id, void *context, struct ibv_sge *sgl, int nsge, int flags)

Input Parameters:
id A reference to the communication identifier where the message buffer will be posted
context A user-defined context associated with the request
sgl A scatter-gather list of memory buffers posted as a single request
nsge The number of scatter-gather entries in the sgl array
flags Optional flags used to control the send operation

Output Parameters:
None

Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_post_sendv posts a work request to the send queue of the queue pair associated with the **rdma_cm_id** , **id** . The contents of the posted buffers will be sent to the remote peer of the connection.
The user is responsible for ensuring that the remote peer has queued a receive request before issuing the send operations. Also, unless the send request is using inline data, the message buffers must already have been registered before being posted. The buffers must remain registered until the send completes.
This routine supports multiple scatter-gather entries.
Send operations may not be posted to an **rdma_cm_id** or the corresponding queue pair until a connection has been established.
The user-defined context associated with the send request will be returned to the user through the work completion work request identifier (**wr_id**) field.

7.2.3. rdma_post_readv

Template:
int rdma_post_readv(struct rdma_cm_id *id, void *context, struct ibv_sge *sgl, int nsge, int flags, uint64_t remote_addr, uint32_t rkey)

Input Parameters:
id A reference to the communication identifier where the request will be posted
context A user-defined context associated with the request
sgl A scatter-gather list of the destination buffers of the read
nsge The number of scatter-gather entries in the sgl array
flags Optional flags used to control the read operation
remote_addr The address of the remote registered memory to read from rkey The registered memory key associated with the remote address

Output Parameters:
None

Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_post_readv posts a work request to the send queue of the queue pair associated with the **rdma_cm_id** , **id** . The contents of the remote memory region at **remote_addr** will be read into the local data buffers given in the sgl array.
The user must ensure that both the remote and local data buffers have been registered before the read is issued. The buffers must remain registered until the read completes.
Read operations may not be posted to an **rdma_cm_id** or the corresponding queue pair until a connection has been established.
The user-defined context associated with the read request will be returned to the user through the work completion work request identifier (**wr_id**) field.

7.2.4. rdma_post_writev

Template:
int rdma_post_writev(struct rdma_cm_id *id, void *context, struct ibv_sge *sgl, int nsge, int flags, uint64_t remote_addr, uint32_t rkey)

Input Parameters:
id A reference to the communication identifier where the request will be posted
context A user-defined context associated with the request
sgl A scatter-gather list of the source buffers of the write
nsge The number of scatter-gather entries in the sgl array
flags Optional flags used to control the write operation
remote_addr The address of the remote registered memory to write into
rkey The registered memory key associated with the remote address

Output Parameters:
None

Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_post_writev posts a work request to the send queue of the queue pair associated with the **rdma_cm_id** , **id** . The contents of the local data buffers in the sgl array will be written to the remote memory region at remote_addr.
Unless inline data is specified, the local data buffers must have been registered before the write is issued, and the buffers must remain registered until the write completes. The remote buffers must always be registered.
Write operations may not be posted to an **rdma_cm_id** or the corresponding queue pair until a connection has been established.
The user-defined context associated with the write request will be returned to the user through the work completion work request identifier (**wr_id**) field.

7.2.5. rdma_post_recv

Template:
int rdma_post_recv(struct rdma_cm_id *id, void *context, void *addr, size_t length, struct ibv_mr *mr)

Input Parameters:
id A reference to the communication identifier where the message buffer will be posted

context A user-defined context associated with the request addr The address of the memory buffer to post
length The length of the memory buffer
mr A registered memory region associated with the posted buffer

Output Parameters:
None

Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_post_recv posts a work request to the receive queue of the queue pair associated with the **rdma_cm_id** , **id** . The posted buffer will be queued to receive an incoming message sent by the remote peer.
The user is responsible for ensuring that receive buffer is posted and is large enough to contain all sent data before the peer posts the corresponding send message. The buffer must have already been registered before being posted, with the mr parameter referencing the registration. The buffer must remain registered until the receive completes.
Messages may be posted to an **rdma_cm_id** only after a queue pair has been associated with it. A queue pair is bound to an **rdma_cm_id** after calling **rdma_create_ep** or **rdma_create_qp** , if the **rdma_cm_id** is allocated using **rdma_create_id** .
The user-defined context associated with the receive request will be returned to the user through the work completion request identifier (**wr_id**) field.
Please note that this is a simple receive call. There are no scatter-gather lists involved here.

7.2.6. rdma_post_send

Template:
int rdma_post_send(struct rdma_cm_id *id, void *context, void *addr, size_t length, struct ibv_mr *mr, int flags)

Input Parameters:
id A reference to the communication identifier where the message buffer will be posted
context A user-defined context associated with the request
addr The address of the memory buffer to post
length The length of the memory buffer
mr Optional registered memory region associated with the posted buffer
flags Optional flags used to control the send operation

Output Parameters:
None

Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_post_send posts a work request to the send queue of the queue pair associated with the **rdma_cm_id** , **id** . The contents of the posted buffer will be sent to the remote peer of the connection.
The user is responsible for ensuring that the remote peer has queued a receive request before

issuing the send operations. Also, unless the send request is using inline data, the message buffer must already have been registered before being posted, with the **mr** parameter referencing the registration. The buffer must remain registered until the send completes.
Send operations may not be posted to an **rdma_cm_id** or the corresponding queue pair until a connection has been established.
The user-defined context associated with the send request will be returned to the user through the work completion work request identifier (**wr_id**) field.

7.2.7. rdma_post_read

Template:
int rdma_post_read(struct rdma_cm_id *id, void *context, void *addr, size_t length, struct ibv_mr *mr, int flags, uint64_t remote_addr, uint32_t rkey)

Input Parameters:
id A reference to the communication identifier where the request will be posted
context A user-defined context associated with the request
addr The address of the local destination of the read request
length The length of the read operation
mr Registered memory region associated with the local buffer
flags Optional flags used to control the read operation
remote_addr The address of the remote registered memory to read from
rkey The registered memory key associated with the remote address

Output Parameters:
None

Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_post_read posts a work request to the send queue of the queue pair associated with the **rdma_cm_id** . The contents of the remote memory region will be read into the local data buffer.
For a list of supported flags, see **ibv_post_send** . The user must ensure that both the remote and local data buffers must have been registered before the read is issued, and the buffers must remain registered until the read completes.
Read operations may not be posted to an **rdma_cm_id** or the corresponding queue pair until it has been connected.
The user-defined context associated with the read request will be returned to the user through the work completion **wr_id** , work request identifier, field.

7.2.8. rdma_post_write

Template:
int rdma_post_write(struct rdma_cm_id *id, void *context, void *addr, size_t length, struct ibv_mr *mr, int flags, uint64_t remote_addr, uint32_t rkey)

Input Parameters:
id A reference to the communication identifier where the request will be posted
context A user-defined context associated with the request

addr The local address of the source of the write request
length The length of the write operation
mr Optional registered memory region associated with the local buffer
flags Optional flags used to control the write operation
remote_addr The address of the remote registered memory to write into
rkey The registered memory key associated with the remote address

Output Parameters:

None

Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_post_write posts a work request to the send queue of the queue pair associated with the **rdma_cm_id** , **id** . The contents of the local data buffer will be written into the remote memory region.
Unless inline data is specified, the local data buffer must have been registered before the write is issued, and the buffer must remain registered until the write completes. The remote buffer must always be registered.
Write operations may not be posted to an **rdma_cm_id** or the corresponding queue pair until a connection has been established.
The user-defined context associated with the write request will be returned to the user through the work completion work request identifier (**wr_id**) field.

7.2.9. rdma_post_ud_send

Template:
int rdma_post_ud_send(struct rdma_cm_id *id, void *context, void *addr, size_t length, struct ibv_mr *mr, int flags, struct ibv_ah *ah, uint32_t remote_qpn)

Input Parameters:
id A reference to the communication identifier where the request will be posted
context A user-defined context associated with the request
addr The address of the memory buffer to post
length The length of the memory buffer
mr Optional registered memory region associated with the posted buffer
flags Optional flags used to control the send operation
ah An address handle describing the address of the remote node
remote_qpn The destination node's queue pair number

Output Parameters:
None

Return Value: 0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_post_ud_send posts a work request to the send queue of the queue pair associated with the **rdma_cm_id** , ID. The contents of the posted buffer will be sent to the specified destination

queue pair, **remote_qpn**.
The user is responsible for ensuring that the destination queue pair has queued a receive request before issuing the send operations. Unless the send request is using inline data, the message buffer must have been registered before being posted, with the **mr** parameter referencing the registration. The buffer must remain registered until the send completes.
The user-defined context associated with the send request will be returned to the user through the work completion work request identifier (**wr_id**) field.

7.2.10. rdma_get_send_comp

Template:
int rdma_get_send_comp(struct rdma_cm_id *id, struct ibv_wc *wc)

Input Parameters:
id A reference to the communication identifier to check for completions
wc A reference to a work completion structure to fill in

Output Parameters:
wc A reference to a work completion structure. The structure will contain information about the completed request when routine returns

Return Value: A non-negative value (0 or 1) equal to the number of completions found on success, or -1 on failure. If the call fails, errno will be set to indicate the reason for the failure.

Description:
rdma_get_send_comp retrieves a completed work request for a send, RDMA read or RDMA write operation. Information about the completed request is returned through the **ibv_wc** , **wc** parameter, with the **wr_id** set to the context of the request. Please see **ibv_poll_cq** for details on the work completion structure, **ibv_wc** . Note that this call polls the send completion queue associated with the **rdma_cm_id** , **id** . If a completion is not found, the call blocks until a request completes. This means, therefore, that the call should only be used on **rdma_cm_ids** which do not share CQs with other **rdma_cm_ids** , and maintain separate CQs for sends and receive completions.

7.2.11. rdma_get_recv_comp

Template:
int rdma_get_recv_comp(struct rdma_cm_id *id, struct ibv_wc *wc)

Input Parameters:
id A reference to the communication identifier to check for completions
wc A reference to a work completion structure to fill in

Output Parameters:
wc A reference to a work completion structure. The structure will contain information about the completed request when routine returns

Return Value: A non-negative value equal to the number of completions found on success, or errno on failure

Description:
rdma_get_recv_comp retrieves a completed work request a receive operation. Information about

the completed request is returned through the **ibv_wc** , **wc** parameter, with the **wr_id** set to the context of the request. Please see **ibv_poll_cq** for details on the work completion structure, **ibv_wc** . Please note that this call polls the receive completion queue associated with the **rdma_cm_id** , **id** . If a completion is not found, the call blocks until a request completes. This means, therefore, that the call should only be used on **rdma_cm_ids** which do not share CQs with other **rdma_cm_ids** , and maintain separate CQs for sends and receive completions.

5.6.2.3.1.8 8. Events

This chapter describes the details of the events that occur when using the VPI API

8.1. IBV Events

8.1.1. IBV_EVENT_CQ_ERR

This event is triggered when a Completion Queue (CQ) overrun occurs or (rare condition) due to a protection error. When this happens, there are no guarantees that completions from the CQ can be pulled. All of the QPs associated with this CQ either in the Read or Send Queue will also get the **IBV_EVENT_QP_FATAL** event. When this event occurs, the best course of action is for the user to destroy and recreate the resources.

8.1.2. IBV_EVENT_QP_FATAL

This event is generated when an error occurs on a Queue Pair (QP) which prevents the generation of completions while accessing or processing the Work Queue on either the Send or Receive Queues. The user must modify the QP state to Reset for recovery. It is the responsibility of the software to ensure that all error processing is completed prior to calling the modify QP verb to change the QP state to Reset. If the problem that caused this event is in the CQ of that Work Queue, the appropriate CQ will also receive the **IBV_EVENT_CQ_ERR** event. In the event of a CQ error, it is best to destroy and recreate the resources

8.1.3. IBV_EVENT_QP_REQ_ERR

This event is generated when the transport layer of the RDMA device detects a transport error violation on the responder side. The error may be caused by the use of an unsupported or reserved opcode, or the use of an out of sequence opcode.

These errors are rare but may occur when there are problems in the subnet or when an RDMA device sends illegal packets.

When this happens, the QP is automatically transitioned to the **IBV_QPS_ERR** state by the RDMA device. The user must modify the states of any such QPs from the error state to the Reset state for recovery.

This event applies only to RC QPs.

8.1.4. IBV_EVENT_QP_ACCESS_ERR

This event is generated when the transport layer of the RDMA device detects a request error violation on the responder side. The error may be caused by

- Misaligned atomic request
- Too many RDMA Read or Atomic requests R_Key violation
- Length errors without immediate data

These errors usually occur because of bugs in the user code.

When this happens, the QP is automatically transitioned to the **IBV_QPS_ERR** state by the RDMA device. The user must modify the QP state to Reset for recovery.

This event is relevant only to RC QPs.

8.1.5. IBV_EVENT_COMM_EST

This event is generated when communication is established on a given QP. This event implies that a QP whose state is **IBV_QPS_RTR** has received the first packet in its Receive Queue and the packet was processed without error. This event is relevant only to connection oriented QPs (RC and UC QPs). It may be generated for UD QPs as well but that is driver implementation specific.

8.1.6. IBV_EVENT_SQ_DRAINED

This event is generated when all outstanding messages have been drained from the Send Queue (SQ) of a QP whose state has now changed from **IBV_QPS_RTS** to **IBV_QPS_SQD**. For RC QPs, this means that all the messages received acknowledgements as appropriate.

Generally, this event will be generated when the internal QP state changes from **SQD.draining** to **SQD.drained**. The event may also be generated if the transition to the state **IBV_QPS_SQD** is aborted because of a transition, either by the RDMA device or by the user, into the **IBV_QPS_SQE**, **IBV_QPS_ERR**, or **IBV_QPS_RESET** QP states.

After this event, and after ensuring that the QP is in the **IBV_QPS_SQD** state, it is safe for the user to start modifying the Send Queue attributes since there aren't are no longer any send messages in progress. Thus it is now safe to modify the operational characteristics of the QP and transition it back to the fully operational RTS state.

8.1.7. IBV_EVENT_PATH_MIG

This event is generated when a connection successfully migrates to an alternate path. The event is relevant only for connection oriented QPs, that is, it is relevant only for RC and UC QPs.When this event is generated, it means that the alternate path attributes are now in use as the primary path attributes. If it is necessary to load attributes for another alternate path, the user may do that after this event is generated.

8.1.8. IBV_EVENT_PATH_MIG_ERR

This event is generated when an error occurs which prevents a QP which has alternate path attributes loaded from performing a path migration change. The attempt to effect the path migration may have been attempted automatically by the RDMA device or explicitly by the user. This error usually occurs if the alternate path attributes are not consistent on the two ends of the connection. It could be, for example, that the DLID is not set correctly or if the source port is invalid. CQ The event may also occur if a cable to the alternate port is unplugged.

8.1.9. IBV_EVENT_DEVICE_FATAL

This event is generated when a catastrophic error is encountered on the channel adapter. The port and possibly the channel adapter becomes unusable.When this event occurs, the behavior of the RDMA device is undetermined and it is highly recommended to close the process immediately. Trying to destroy the RDMA resources may fail and thus the device may be left in an unstable state.

8.1.10. IBV_EVENT_PORT_ACTIVE

This event is generated when the link on a given port transitions to the active state. The link is now available for send/receive packets. This event means that the **port_attr** state has moved from one of the following states:

- **IBV_PORT_DOWN**
- **IBV_PORT_INIT**
- **IBV_PORT_ARMED**

To either:

- **IBV_PORT_ACTIVE**
- **IBV_PORT_ACTIVE_DEFER**

This might happen for example when the SM configures the port. The event is generated by the device only if the **IBV_DEVICE_PORT_ACTIVE_EVENT** attribute is set in the **dev_cap.device_cap_flags**.

8.1.11. IBV_EVENT_PORT_ERR

This event is generated when the link on a given port becomes inactive and is thus unavailable to send/receive packets.The port_attr.state must have been in either in either **IBV_PORT_ACTIVE** or **IBV_PORT_ACTIVE_DEFER** state and transitions to one of the following states:

- **IBV_PORT_DOWN**
- **IBV_PORT_INIT**
- **IBV_PORT_ARMED**

This can happen when there are connectivity problems within the IB fabric, for example when a cable is accidentally pulled. This will not affect the QPs associated with this port, although if this is a reliable connection, the retry count may be exceeded if the link takes a long time to come back up.

8.1.12. IBV_EVENT_LID_CHANGE

The event is generated when the LID on a given port changes. This is done by the SM. If this is not the first time that the SM configures the port LID, it may indicate that there is a new SM on the subnet or that the SM has reconfigured the subnet. If the user cached the structure returned from **ibv_query_port()**, then these values must be flushed when this event occurs.

8.1.13. IBV_EVENT_PKEY_CHANGE

This event is generated when the P_Key table changes on a given port. The PKEY table is configured by the SM and this also means that the SM can change it. When that happens, an **IBV_EVENT_PKEY_CHANGE** event is generated.

Since QPs use GID table indexes rather than absolute values (as the source GID), it is suggested for clients to check that the GID indexes used by the client's QPs are not changed as a result of this event.

If a user caches the values of the P_Key table, then these must be flushed when the **IBV_EVENT_PKEY_CHANGE** event is received.

8.1.14. IBV_EVENT_SM_CHANGE

This event is generated when the SM being used at a given port changes. The user application must re-register with the new SM. This means that all subscriptions previously registered from the given port, such as one to join a multicast group, must be re-registered.

8.1.15. IBV_EVENT_SRQ_ERR

This event is generated when an error occurs on a Shared Receive Queue (SRQ) which prevents the RDMA device from dequeuing WRs from the SRQ and reporting of receive completions.

When an SRQ experiences this error, all the QPs associated with this SRQ will be transitioned to the **IBV_QPS_ERR** state and the **IBV_EVENT_QP_FATAL** asynchronous event will be generated for them. Any QPs which have transitioned to the error state must have their state modified to Reset for recovery.

8.1.16. IBV_EVENT_SRQ_LIMIT_REACHED

This event is generated when the limit for the SRQ resources is reached. This means that the number of SRQ Work Requests (WRs) is less than the SRQ limit. This event may be used by the user as an indicator that more WRs need to be posted to the SRQ and rearm it.

8.1.17. IBV_EVENT_QP_LAST_WQE_REACHED

This event is generated when a QP which is associated with an SRQ is transitioned into the **IBV_QPS_ERR** state either automatically by the RDMA device or explicitly by the user. This may have happened either because a completion with error was generated for the last WQE, or the QP transitioned into the **IBV_QPS_ERR** state and there are no more WQEs on the Receive Queue of the QP.

This event actually means that no more WQEs will be consumed from the SRQ by this QP.

If an error occurs to a QP and this event is not generated, the user must destroy all of the QPs associated with this SRQ as well as the SRQ itself in order to reclaim all of the WQEs associated with the offending QP. At the minimum, the QP which is in the error state must have its state changed to Reset for recovery.

8.1.18. IBV_EVENT_CLIENT_REREGISTER

This event is generated when the SM sends a request to a given port for client re-registration for all subscriptions previously requested for the port. This could happen if the SM suffers a failure and as a result, loses its own records of the subscriptions. It may also happen if a new SM becomes operational on the subnet.The event will be generated by the device only if the bit that indicates a client reregister is supported is set in **port_attr.port_cap_flags**.

8.1.19. IBV_EVENT_GID_CHANGE

This event is generated when a GID changes on a given port. The GID table is configured by the SM and this also means that the SM can change it. When that happens, an **IBV_EVENT_GID_CHANGE** event is generated. If a user caches the values of the GID table, then these must be flushed when the **IBV_EVENT_GID_CHANGE** event is received.

8.2. IBV WC Events

8.2.1. IBV_WC_SUCCESS

The Work Request completed successfully.

8.2.2. IBV_WC_LOC_LEN_ERR

This event is generated when the receive buffer is smaller than the incoming send. It is generated on the receiver side of the connection.

8.2.3. IBV_WC_LOC_QP_OP_ERR

This event is generated when a QP error occurs. For example, it may be generated if a user neglects to specify **responder_resources** and **initiator_depth** values in struct **rdma_conn_param** before calling **rdma_connect()** on the client side and **rdma_accept()** on the server side.

8.2.4. IBV_WC_LOC_EEC_OP_ERR

This event is generated when there is an error related to the local EEC's receive logic while executing the request packet. The responder is unable to complete the request. This error is not caused by the sender.

8.2.5. IBV_WC_LOC_PROT_ERR

This event is generated when a user attempts to access an address outside of the registered memory region. For example, this may happen if the Lkey does not match the address in the WR.

8.2.6. IBV_WC_WR_FLUSH_ERR

This event is generated when an invalid remote error is thrown when the responder detects an invalid request. It may be that the operation is not supported by the request queue or there is insufficient buffer space to receive the request.

8.2.7. IBV_WC_MW_BIND_ERR

This event is generated when a memory management operation error occurs. The error may be due to the fact that the memory window and the QP belong to different protection domains. It may also be that the memory window is not allowed to be bound to the specified MR or the access permissions may be wrong.

8.2.8. IBV_WC_BAD_RESP_ERR

This event is generated when an unexpected transport layer opcode is returned by the responder.

8.2.9. IBV_WC_LOC_ACCESS_ERR

This event is generated when a local protection error occurs on a local data buffer during the process of an RDMA Write with Immediate Data operation sent from the remote node.

8.2.10. IBV_WC_REM_INV_REQ_ERR

This event is generated when the receive buffer is smaller than the incoming send. It is generated on the sender side of the connection. It may also be generated if the QP attributes are not set correctly, particularly those governing MR access.

8.2.11. IBV_WC_REM_ACCESS_ERR

This event is generated when a protection error occurs on a remote data buffer to be read by an RDMA Read, written by an RDMA Write or accessed by an atomic operation. The error is reported only on RDMA operations or atomic operations.

8.2.12. IBV_WC_REM_OP_ERR

This event is generated when an operation cannot be completed successfully by the responder. The failure to complete the operation may be due to QP related errors which prevent the responder from completing the request or a malformed WQE on the Receive Queue.

8.2.13. IBV_WC_RETRY_EXC_ERR

This event is generated when a sender is unable to receive feedback from the receiver. This means that either the receiver just never ACKs sender messages in a specified time period, or it has been disconnected or it is in a bad state which prevents it from responding.

8.2.14. IBV_WC_RNR_RETRY_EXC_ERR

This event is generated when the RNR NAK retry count is exceeded. This may be caused by lack of receive buffers on the responder side.

8.2.15. IBV_WC_LOC_RDD_VIOL_ERR

This event is generated when the RDD associated with the QP does not match the RDD associated with the EEC.

8.2.16. IBV_WC_REM_INV_RD_REQ_ERR

This event is generated when the responder detects an invalid incoming RD message. The message may be invalid because it has in invalid Q_Key or there may be a Reliable Datagram Domain (RDD) violation.

8.2.17. IBV_WC_REM_ABORT_ERR

This event is generated when an error occurs on the responder side which causes it to abort the operation.

8.2.18. IBV_WC_INV_EECN_ERR

This event is generated when an invalid End to End Context Number (EECN) is detected.

8.2.19. IBV_WC_INV_EEC_STATE_ERR

This event is generated when an illegal operation is detected in a request for the specified EEC state.

8.2.20. IBV_WC_FATAL_ERR

This event is generated when a fatal transport error occurs. The user may have to restart the RDMA device driver or reboot the server to recover from the error.

8.2.21. IBV_WC_RESP_TIMEOUT_ERR

This event is generated when the responder is unable to respond to a request within the timeout period. It generally indicates that the receiver is not ready to process requests.

8.2.22. IBV_WC_GENERAL_ERR

This event is generated when there is a transport error which cannot be described by the other specific events discussed here.

8.3. RDMA_CM Events

8.3.1. RDMA_CM_EVENT_ADDR_RESOLVED

This event is generated on the client (active) side in response to `rdma_resolve_addr()`. It is generated when the system is able to resolve the server address supplied by the client.

8.3.2. RDMA_CM_EVENT_ADDR_ERROR

This event is generated on the client (active) side. It is generated in response to `rdma_resolve_addr()` in the case where an error occurs. This may happen, for example, if the device cannot be found such as when a user supplies an incorrect device. Specifically, if the remote device has both ethernet and IB interfaces, and the client side supplies the ethernet device name instead of the IB device name of the server side, an `RDMA_CM_EVENT_ADDR_ERROR` will be generated.

8.3.3. RDMA_CM_EVENT_ROUTE_RESOLVED

This event is generated on the client (active) side in response to `rdma_resolve_route()`. It is generated when the system is able to resolve the server address supplied by the client.

8.3.4. RDMA_CM_EVENT_ROUTE_ERROR

This event is generated when `rdma_resolve_route()` fails.

8.3.5. RDMA_CM_EVENT_CONNECT_REQUEST

This is generated on the passive side of the connection to notify the user of a new connection request. It indicates that a connection request has been received.

8.3.6. RDMA_CM_EVENT_CONNECT_RESPONSE

This event may be generated on the active side of the connection to notify the user that the connection request has been successful. The event is only generated on `rdma_cm_id`s which do not have a QP associated with them.

8.3.7. RDMA_CM_EVENT_CONNECT_ERROR

This event may be generated on the active or passive side of the connection. It is generated when an error occurs while attempting to establish a connection.

8.3.8. RDMA_CM_EVENT_UNREACHABLE

This event is generated on the active side of a connection. It indicates that the (remote) server is unreachable or unable to respond to a connection request.

8.3.9. RDMA_CM_EVENT_REJECTED

This event may be generated on the client (active) side and indicates that a connection request or response has been rejected by the remote device. This may happen for example if an attempt is made to connect with the remote end point on the wrong port.

8.3.10. RDMA_CM_EVENT_ESTABLISHED

This event is generated on both sides of a connection. It indicates that a connection has been established with the remote end point.

8.3.11. RDMA_CM_EVENT_DISCONNECTED

This event is generated on both sides of the connection in response to `rdma_disconnect()`. The event will be generated to indicate that the connection between the local and remote devices has been disconnected. Any associated QP will transition to the error state. All posted work requests are flushed. The user must change any such QP's state to Reset for recovery.

8.3.12. RDMA_CM_EVENT_DEVICE_REMOVAL

This event is generated when the RDMA CM indicates that the device associated with the `rdma_cm_id` has been removed. Upon receipt of this event, the user must destroy the related `rdma_cm_id`.

8.3.13. RDMA_CM_EVENT_MULTICAST_JOIN

This event is generated in response to `rdma_join_multicast()`. It indicates that the multicast join operation has completed successfully.

8.3.14. RDMA_CM_EVENT_MULTICAST_ERROR

This event is generated when an error occurs while attempting to join a multicast group or on an existing multicast group if the group had already been joined. When this happens, the multicast group will no longer be accessible and must be rejoined if necessary.

8.3.15. RDMA_CM_EVENT_ADDR_CHANGE

This event is generated when the network device associated with this ID through address resolution changes its hardware address. For example, this may happen following bonding fail over. This event may serve to aid applications which want the links used for their RDMA sessions to align with the network stack.

8.3.16. RDMA_CM_EVENT_TIMEWAIT_EXIT

This event is generated when the QP associated with the connection has exited its timewait state and is now ready to be re-used. After a QP has been disconnected, it is maintained in a timewait state to allow any in flight packets to exit the network. After the timewait state has completed, the `rdma_cm` will report this event.

5.6.2.3.1.9 9. Programming Examples Using IBV Verbs

This chapter provides code examples using the IBV Verbs

9.1. Synopsis for RDMA_RC Example Using IBV Verbs

The following is a synopsis of the functions in the programming example, in the order that they are called.

9.1.1. Main

1. Parse the command line: The user may set the TCP port, device name, and device port for the test.
    - If set, these values will override the default values in the configuration.
    - The last parameter is the server name. If a server name is provided, the program connects to it and operates in client mode. Otherwise, the program defaults to server mode.
2. Call `print_config`.

3. Call **resources_init**.
4. Call **resources_create**.
5. Call **connect_qp**.
6. If in server mode, call **post_send** with the **IBV_WR_SEND** operation.
7. Call **poll_completion**.

> ⚠ The server side expects a completion from the SEND request, while the client side expects a RECEIVE completion.

8. If in client mode, show the message received via the RECEIVE operation; if in server mode, load the buffer with a new message.
9. Sync the client and server.
10. At this point, the server goes directly to the next sync. All RDMA operations are done strictly by the client.

9.1.1.1. Client-Only Operations

1. Call **post_send** with **IBV_WR_RDMA_READ** to perform an RDMA read of the server's buffer.
2. Call **poll_completion**.
3. Show the server's message.
4. Set up the send buffer with a new message.
5. Call **post_send** with **IBV_WR_RDMA_WRITE** to perform an RDMA write to the server's buffer.
6. Call **poll_completion**.

9.1.1.2. Post-Client Operations

1. Sync the client and server.
2. If in server mode, show the buffer to prove the RDMA write was successful.
3. Call **resources_destroy**.
4. Free the device name string.
5. Done.

9.1.2. print_config

Prints out the configuration information.

9.1.3. resources_init

Clears the resources struct.

9.1.4. resources_create

1. Call **sock_connect** to connect a TCP socket to the peer.
2. Get the list of devices, locate the desired one, and open it.
3. Free the device list.
4. Get the port information.
5. Create a Protection Domain (PD).
6. Create a Completion Queue (CQ).

7. Allocate a memory buffer, initialize it, and register it.
8. Create a Queue Pair (QP).

9.1.5. sock_connect

- Client mode – Resolves the DNS address of the server and initiates a connection to it.
- Server mode – Listens for an incoming connection on the indicated port.

9.1.6. connect_qp

1. Call **modify_qp_to_init**.
2. Call **post_receive**.
3. Call **sock_sync_data** to exchange information between the server and the client.
4. Call **modify_qp_to_rtr**.
5. Call **modify_qp_to_rts**.
6. Call **sock_sync_data** to synchronize the client and server.

9.1.7. modify_qp_to_init

Transitions the QP to the INIT state.

9.1.8. post_receive

1. Prepare a scatter/gather entry for the receive buffer.
2. Prepare a Receive Request (RR).
3. Post the RR.

9.1.9. sock_sync_data

Uses the TCP socket created with **sock_connect** to synchronize the given set of data between the client and the server.

> ⚠ Because this function is blocking, it is also called with dummy data to synchronize the timing of the client and server.

9.1.10. modify_qp_to_rtr

Transitions the QP to the RTR (Ready to Receive) state.

9.1.11. modify_qp_to_rts

Transitions the QP to the RTS (Ready to Send) state.

9.1.12. post_send

1. Prepare a scatter/gather entry for the data to be sent (or received, in the case of an RDMA read).
2. Create a Send Request (SR).

> ⚠ The **IBV_SEND_SIGNALED** flag is redundant.

3. If this is an RDMA operation, set the address and key.
4. Post the SR.

9.1.13. poll_completion

Polls the CQ until an entry is found or the **MAX_POLL_CQ_TIMEOUT** (in milliseconds) is reached.

9.1.14. resources_destroy

Releases, frees, and deallocates all items within the resource struct.

9.2. Code for Send, Receive, RDMA Read, RDMA Write

```
/*
* BUILD COMMAND:
* gcc -Wall -I/usr/local/ofed/include -O2 -o RDMA_RC_example -L/usr/local/ofed/lib64 -L/usr/local/ofed/lib
-libverbs RDMA_RC_example.c
*
*/
/****************************************************************************
*
* RDMA Aware Networks Programming Example
*
* This code demonstrates how to perform the following operations using the * VPI Verbs API:
*
* Send
* Receive
* RDMA Read
* RDMA Write
*
****************************************************************************/

#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include

/* poll CQ timeout in millisec (2 seconds) */
#define MAX_POLL_CQ_TIMEOUT 2000
#define MSG "SEND operation "
#define RDMAMSGR "RDMA read operation "
#define RDMAMSGW "RDMA write operation"
#define MSG_SIZE (strlen(MSG) + 1)

#if __BYTE_ORDER == __LITTLE_ENDIAN
static inline uint64_t htonll(uint64_t x) { return bswap_64(x); }
static inline uint64_t ntohll(uint64_t x) { return bswap_64(x); }
#elif __BYTE_ORDER == __BIG_ENDIAN
static inline uint64_t htonll(uint64_t x) { return x; }
static inline uint64_t ntohll(uint64_t x) { return x; }
#else
#error __BYTE_ORDER is neither __LITTLE_ENDIAN nor __BIG_ENDIAN
#endif

/* structure of test parameters */
struct config_t
{
```

9.3. Synopsis for Multicast Example Using RDMA_CM and IBV Verbs

This code example for Multicast, uses RDMA-CM and VPI (and hence can be run both over IB and over LLE).

Notes:

1. In order to run the multicast example on either IB or LLE, no change is needed to the test's code. However if RDMA_CM is used, it is required that the network interface will be configured and up (whether it is used over RoCE or over IB).
2. For the IB case, a join operation is involved, yet it is performed by the rdma_cm kernel code.
3. For the LLE case, no join is required. All MGIDs are resolved into MACs at the host.
4. To inform the multicast example which port to use, you need to specify **-b <IP-address>** to bind to the desired device port.

9.3.1. Main

1. Get command line parameters.
    - m – MC address, destination port
    - M – unmapped MC address, requires also bind address (parameter "b")
    - s – sender flag.
    - b – bind address.
    - c – connections amount.
    - C – message count.
    - S – message size.
    - p – port space (UDP default; IPoIB)
2. Create event channel to receive asynchronous events.
3. Allocate Node and creates an identifier that is used to track communication information
4. Start the "run" main function.
5. On ending – release and free resources.

API definition files: **rdma/rdma_cma.h and infiniband/verbs.h** .

9.3.2. Run

1. Get source (if provided for binding) and destination addresses. Convert the input addresses to socket presentation.
2. Joining:
    a. For all connections:
    if source address is specifically provided, then bind the rdma_cm object to the corresponding network interface. (Associates a source address with an rdma_cm identifier).
    if unmapped MC address with bind address provided, check the remote address and then bind.
    b. Poll on all the connection events and wait that all rdma_cm objects joined the MC group.
3. Send & receive:
    a. If sender: send the messages to all connection nodes (function “post_sends”).
    b. If receiver: poll the completion queue (function “poll_cqs”) till messages arrival.
4. Upon ending – release network resources (per all connections: leaves the multicast group and detaches its associated QP from the group)

9.4. Code for Multicast Using RDMA_CM and IBV Verbs

```
Multicast Code Example
/*
 * BUILD COMMAND:
```

```
 * gcc -g -Wall -D_GNU_SOURCE -g -O2 -o examples/mckey  examples/mckey.c  -libverbs -lrdmacm
 *
* $Id$
 */

#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include

#include

struct cmatest_node
{
    int         id;
    struct rdma_cm_id   *cma_id;
    int         connected;
    struct ibv_pd   *pd;
    struct ibv_cq   *cq;
    struct ibv_mr   *mr;
    struct ibv_ah   *ah;
    uint32_t        remote_qpn;
    uint32_t        remote_qkey;
    void            *mem;
};

struct cmatest
{
    struct rdma_event_channel *channel;
    struct cmatest_node *nodes;
    int conn_index;
    int connects_left;

    struct sockaddr_in6 dst_in;
    struct sockaddr     *dst_addr;
    struct sockaddr_in6 src_in;
    struct sockaddr     *src_addr;
};

static struct cmatest test;
static int connections = 1;
static int message_size = 100;
static int message_count = 10;
static int is_sender;
static int unmapped_addr;
static char *dst_addr;
static char *src_addr;
static enum rdma_port_space port_space = RDMA_PS_UDP;

static int create_message(struct cmatest_node *node)
{
    if (!message_size)
    message_count = 0;

    if (!message_count)
    return 0;

    node->mem = malloc(message_size + sizeof(struct ibv_grh));
    if (!node->mem)
    {
    printf("failed message allocation\n");
        return -1;
    }
    node->mr = ibv_reg_mr(node->pd, node->mem,  message_size + sizeof(struct ibv_grh),
    IBV_ACCESS_LOCAL_WRITE);
    if (!node->mr)
    {
    printf("failed to reg MR\n");
    goto err;
    }
    return 0;
err:
    free(node->mem);
    return -1;
}

static int verify_test_params(struct cmatest_node *node)
{
    struct ibv_port_attr port_attr;
    int ret;

    ret = ibv_query_port(node->cma_id->verbs, node->cma_id->port_num, &port_attr);
    if (ret)
    return ret;

    if (message_count && message_size > (1 << (port_attr.active_mtu + 7)))
    {
    printf("mckey: message_size %d is larger than active mtu %d\n", message_size, 1 <<  (port_attr.active_mtu +
7));
    return -EINVAL;
    }

    return 0;
}

static int init_node(struct cmatest_node *node)
{
```

```
    struct ibv_qp_init_attr init_qp_attr;
    int cqe, ret;

    node->pd = ibv_alloc_pd(node->cma_id->verbs);
    if (!node->pd)
    {
    ret = -ENOMEM;
    printf("mckey: unable to allocate PD\n");
    goto out;
    }

    cqe = message_count ? message_count * 2 : 2;
    node->cq = ibv_create_cq(node->cma_id->verbs, cqe, node, 0, 0);
    if (!node->cq)
    {
    ret = -ENOMEM;
    printf("mckey: unable to create CQ\n");
    goto out;
    }

    memset(&init_qp_attr, 0, sizeof init_qp_attr);
    init_qp_attr.cap.max_send_wr = message_count ? message_count : 1;
    init_qp_attr.cap.max_recv_wr = message_count ? message_count : 1;
    init_qp_attr.cap.max_send_sge = 1;
    init_qp_attr.cap.max_recv_sge = 1;
    init_qp_attr.qp_context = node;
    init_qp_attr.sq_sig_all = 0;
    init_qp_attr.qp_type = IBV_QPT_UD;
    init_qp_attr.send_cq = node->cq;
    init_qp_attr.recv_cq = node->cq;
    ret = rdma_create_qp(node->cma_id, node->pd, &init_qp_attr);
    if (ret)
    {
    printf("mckey: unable to create QP: %d\n", ret);
    goto out;
    }

    ret = create_message(node);
    if (ret)
    {
    printf("mckey: failed to create messages: %d\n", ret);
    goto out;
    }
out:
    return ret;
}

static int post_recvs(struct cmatest_node *node)
{
    struct ibv_recv_wr recv_wr, *recv_failure;
    struct ibv_sge sge;
    int i, ret = 0;

    if (!message_count)
    return 0;

    recv_wr.next = NULL;
    recv_wr.sg_list = &sge;
    recv_wr.num_sge = 1;
    recv_wr.wr_id = (uintptr_t) node;

    sge.length = message_size + sizeof(struct ibv_grh);
    sge.lkey = node->mr->lkey;
    sge.addr = (uintptr_t) node->mem;

    for (i = 0; i < message_count && !ret; i++ )
    {
    ret = ibv_post_recv(node->cma_id->qp, &recv_wr, &recv_failure);
    if (ret)
    {
    printf("failed to post receives: %d\n", ret);
    break;
    }
    }
    return ret;
}

static int post_sends(struct cmatest_node *node, int signal_flag)
{
    struct ibv_send_wr send_wr, *bad_send_wr;
    struct ibv_sge sge;
    int i, ret = 0;

    if (!node->connected || !message_count)
    return 0;

    send_wr.next = NULL;
    send_wr.sg_list = &sge;
    send_wr.num_sge = 1;
    send_wr.opcode = IBV_WR_SEND_WITH_IMM;
    send_wr.send_flags = signal_flag;
    send_wr.wr_id = (unsigned long)node;
    send_wr.imm_data = htonl(node->cma_id->qp->qp_num);

    send_wr.wr.ud.ah = node->ah;
    send_wr.wr.ud.remote_qpn = node->remote_qpn;
    send_wr.wr.ud.remote_qkey = node->remote_qkey;

    sge.length = message_size;
    sge.lkey = node->mr->lkey;
    sge.addr = (uintptr_t) node->mem;

    for (i = 0; i < message_count && !ret; i++)
    {
    ret = ibv_post_send(node->cma_id->qp, &send_wr, &bad_send_wr);
```

```
    if (ret)
    printf("failed to post sends: %d\n", ret);
    }
    return ret;
}

static void connect_error(void)
{
    test.connects_left--;
}

static int addr_handler(struct cmatest_node *node)
{
    int ret;

    ret = verify_test_params(node);
    if (ret)
    goto err;

    ret = init_node(node);
    if (ret)
    goto err;

    if (!is_sender)
    {
    ret = post_recvs(node);
    if (ret)
    goto err;
    }

    ret = rdma_join_multicast(node->cma_id, test.dst_addr, node);
    if (ret)
    {
    printf("mckey: failure joining: %d\n", ret);
    goto err;
    }
    return 0;
err:
    connect_error();
    return ret;
}

static int join_handler(struct cmatest_node *node,  struct rdma_ud_param *param)
{
    char buf[40];

    inet_ntop(AF_INET6, param->ah_attr.grh.dgid.raw, buf, 40);
    printf("mckey: joined dgid: %s\n", buf);

    node->remote_qpn = param->qp_num;
    node->remote_qkey = param->qkey;
    node->ah = ibv_create_ah(node->pd, ¶m->ah_attr);
    if (!node->ah)
    {
    printf("mckey: failure creating address handle\n");
    goto err;
    }

    node->connected = 1;
    test.connects_left--;
    return 0;
err:
    connect_error();
    return -1;
}

static int cma_handler(struct rdma_cm_id *cma_id, struct rdma_cm_event *event)
{
    int ret = 0;

    switch (event->event)
    {
    case RDMA_CM_EVENT_ADDR_RESOLVED:
    ret = addr_handler(cma_id->context);
    break;
    case RDMA_CM_EVENT_MULTICAST_JOIN:
    ret = join_handler(cma_id->context, &event->param.ud);
    break;
    case RDMA_CM_EVENT_ADDR_ERROR:
    case RDMA_CM_EVENT_ROUTE_ERROR:
    case RDMA_CM_EVENT_MULTICAST_ERROR:
    printf("mckey: event: %s, error: %d\n",      rdma_event_str(event->event), event->status);
    connect_error();
    ret = event->status;
    break;
    case RDMA_CM_EVENT_DEVICE_REMOVAL:
    /* Cleanup will occur after test completes. */
    break;
    default:
    break;
    }
    return ret;
}

static void destroy_node(struct cmatest_node *node)
{
    if (!node->cma_id)
    return;

    if (node->ah)
    ibv_destroy_ah(node->ah);

    if (node->cma_id->qp)
    rdma_destroy_qp(node->cma_id);
```

```
    if (node->cq)
    ibv_destroy_cq(node->cq);

    if (node->mem)
    {
    ibv_dereg_mr(node->mr);
    free(node->mem);
    }

    if (node->pd)
    ibv_dealloc_pd(node->pd);

    /* Destroy the RDMA ID after all device resources */
    rdma_destroy_id(node->cma_id);
}

static int alloc_nodes(void)
{
    int ret, i;

    test.nodes = malloc(sizeof *test.nodes * connections);
    if (!test.nodes)
    {
    printf("mckey: unable to allocate memory for test nodes\n");
    return -ENOMEM;
    }
    memset(test.nodes, 0, sizeof *test.nodes * connections);

    for (i = 0; i < connections; i++)
    {
    test.nodes[i].id = i;
    ret = rdma_create_id(test.channel, &test.nodes[i].cma_id, &test.nodes[i], port_space);
    if (ret)
    goto err;
    }
    return 0;
err:
    while (--i >= 0)
    rdma_destroy_id(test.nodes[i].cma_id);
    free(test.nodes);
    return ret;
}

static void destroy_nodes(void)
{
    int i;

    for (i = 0; i < connections; i++)
    destroy_node(&test.nodes[i]);
    free(test.nodes);
}

static int poll_cqs(void)
{
    struct ibv_wc wc[8];
    int done, i, ret;

    for (i = 0; i < connections; i++)
    {
    if (!test.nodes[i].connected)
    continue;

    for (done = 0; done < message_count; done += ret)
    {
    ret = ibv_poll_cq(test.nodes[i].cq, 8, wc);
    if (ret < 0)
    {
    printf("mckey: failed polling CQ: %d\n", ret);
    return ret;
    }
    }
    }
    return 0;
}

static int connect_events(void)
{
    struct rdma_cm_event *event;
    int ret = 0;

    while (test.connects_left && !ret)
    {
        ret = rdma_get_cm_event(test.channel, &event);
        if (!ret)
        {
            ret = cma_handler(event->id, event);
            rdma_ack_cm_event(event);
        }
    }
    return ret;
}

static int get_addr(char *dst, struct sockaddr *addr)
{
    struct addrinfo *res;
    int ret;

    ret = getaddrinfo(dst, NULL, NULL, &res);
    if (ret)
    {
    printf("getaddrinfo failed - invalid hostname or IP address\n");
    return ret;
    }

    memcpy(addr, res->ai_addr, res->ai_addrlen);
```

```
    freeaddrinfo(res);
    return ret;
}

static int run(void)
{
    int i, ret;

    printf("mckey: starting %s\n", is_sender ? "client" : "server");
    if (src_addr)
    {
    ret = get_addr(src_addr, (struct sockaddr *) &test.src_in);
    if (ret)
    return ret;
    }

    ret = get_addr(dst_addr, (struct sockaddr *) &test.dst_in);
    if (ret)
    return ret;

    printf("mckey: joining\n");
    for (i = 0; i < connections; i++)
    {
        if (src_addr)
        {
            ret = rdma_bind_addr(test.nodes[i].cma_id, test.src_addr);
            if (ret)
            {
                printf("mckey: addr bind failure: %d\n", ret);
                connect_error();
                return ret;
            }
        }

        if (unmapped_addr)
            ret = addr_handler(&test.nodes[i]);
        else
            ret = rdma_resolve_addr(test.nodes[i].cma_id,                   test.src_addr, test.dst_addr,
2000);
        if (ret)
        {
            printf("mckey: resolve addr failure: %d\n", ret);
            connect_error();
            return ret;
        }
    }

    ret = connect_events();
    if (ret)
        goto out;

    /*
     * Pause to give SM chance to configure switches.  We don't want to
     * handle reliability issue in this simple test program.
     */
    sleep(3);

    if (message_count)
    {
        if (is_sender)
        {
            printf("initiating data transfers\n");
            for (i = 0; i < connections; i++)
            {
                ret = post_sends(&test.nodes[i], 0);
                if (ret)
                    goto out;
            }
    }
    else
    {
        printf("receiving data transfers\n");
        ret = poll_cqs();
        if (ret)
            goto out;
    }
    printf("data transfers complete\n");
    }
out:
    for (i = 0; i < connections; i++)
    {
        ret = rdma_leave_multicast(test.nodes[i].cma_id, test.dst_addr);
        if (ret)
            printf("mckey: failure leaving: %d\n", ret);
    }
    return ret;
}

int main(int argc, char **argv)
{
    int op, ret;


    while ((op = getopt(argc, argv, "m:M:sb:c:C:S:p:")) != -1)
    {
        switch (op)
        {
        case 'm':
            dst_addr = optarg;
            break;
        case 'M':
            unmapped_addr = 1;
            dst_addr = optarg;
            break;
        case 's':
```

```
            is_sender = 1;
            break;
        case 'b':
            src_addr = optarg;
            test.src_addr = (struct sockaddr *) &test.src_in;
            break;
        case 'c':
            connections = atoi(optarg);
            break;
        case 'C':
            message_count = atoi(optarg);
            break;
        case 'S':
            message_size = atoi(optarg);
            break;
        case 'p':
            port_space = strtol(optarg, NULL, 0);
            break;
        default:
            printf("usage: %s\n", argv[0]);
            printf("\t-m multicast_address\n");
            printf("\t[-M unmapped_multicast_address]\n"
                   "\t replaces -m and requires -b\n");
            printf("\t[-s(ender)]\n");
            printf("\t[-b bind_address]\n");
            printf("\t[-c connections]\n");
            printf("\t[-C message_count]\n");
            printf("\t[-S message_size]\n");
            printf("\t[-p port_space - %#x for UDP (default), %#x for IPOIB]\n", RDMA_PS_UDP, RDMA_PS_IPOIB);
            exit(1);
        }
    }

    test.dst_addr = (struct sockaddr *) &test.dst_in;
    test.connects_left = connections;

    test.channel = rdma_create_event_channel();
    if (!test.channel)
    {
        printf("failed to create event channel\n");
        exit(1);
    }

    if (alloc_nodes())
        exit(1);

    ret = run();

    printf("test complete\n");
    destroy_nodes();
    rdma_destroy_event_channel(test.channel);

    printf("return status %d\n", ret);
    return ret;
}
```

9.5. Programming Examples Using RDMA Verbs

This chapter provides code examples using the RDMA Verbs

9.5.1. Automatic Path Migration (APM)

```
//*
 * Compile Command:
 * gcc apm.c -o apm -libverbs -lrdmacm
 *
 * Description:
 * This example demonstrates Automatic Path Migration (APM). The basic flow is
 * as follows:
 * 1. Create connection between client and server
 * 2. Set the alternate path details on each side of the connection
 * 3. Perform send operations back and forth between client and server
 * 4. Cause the path to be migrated (manually or automatically)
 * 5. Complete sends using the alternate path
 *
 * There are two ways to cause the path to be migrated.
 * 1. Use the ibv_modify_qp verb to set path_mig_state = IBV_MIG_MIGRATED
 * 2. Assuming there are two ports on at least one side of the connection, and
 *    each port has a path to the other host, pull out the cable of the original
 *    port and watch it migrate to the other port.
 *
 * Running the Example:
 * This example requires a specific IB network configuration to properly
 * demonstrate APM. Two hosts are required, one for the client and one for the
 * server. At least one of these two hosts must have a IB card with two ports.
 * Both of these ports should be connected to the same subnet and each have a
 * route to the other host through an IB switch.
 * The executable can operate as either the client or server application. Start
 * the server side first on one host then start the client on the other host. With default parameters, the
 * client and server will exchange 100 sends over 100 seconds. During that time,
 * manually unplug the cable connected to the original port of the two port
 * host, and watch the path get migrated to the other port. It may take up to
 * a minute for the path to migrated. To see the path get migrated by software,
 * use the -m option on the client side.
 *
 * Server:
 * ./apm -s
 *
```

```
 * Client (-a is IP of remote interface):
 * ./apm -a 192.168.1.12
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <getopt.h>
#include <rdma/rdma_verbs.h>

#define VERB_ERR(verb, ret) \
        fprintf(stderr, "%s returned %d errno %d\n", verb, ret, errno)

/* Default parameter values */
#define DEFAULT_PORT "51216"
#define DEFAULT_MSG_COUNT 100
#define DEFAULT_MSG_LENGTH 1000000
#define DEFAULT_MSEC_DELAY 500

/* Resources used in the example */
struct context
{
    /* User parameters */
    int server;
    char *server_name;
    char *server_port;
    int msg_count;
    int msg_length;
    int msec_delay;
    uint8_t alt_srcport;
    uint16_t alt_dlid;
    uint16_t my_alt_dlid;
    int migrate_after;

    /* Resources */
    struct rdma_cm_id *id;
    struct rdma_cm_id *listen_id;
    struct ibv_mr *send_mr;
    struct ibv_mr *recv_mr;
    char *send_buf;
    char *recv_buf;
    pthread_t async_event_thread;
};

/*
 * Function:    async_event_thread
 *
 * Input:
 *      arg    The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      NULL
 *
 * Description:
 *      Reads any Asynchronous events that occur during the sending of data
 *      and prints out the details of the event. Specifically migration
 *      related events.
 */
static void *async_event_thread(void *arg)
{
    struct ibv_async_event event;
    int ret;

    struct context *ctx = (struct context *) arg;

    while (1) {
        ret = ibv_get_async_event(ctx->id->verbs, &event);
        if (ret) {
            VERB_ERR("ibv_get_async_event", ret);
            break;
        }

        switch (event.event_type) {
        case IBV_EVENT_PATH_MIG:
            printf("QP path migrated\n");
            break;
        case IBV_EVENT_PATH_MIG_ERR:
            printf("QP path migration error\n");
            break;
        default:
            printf("Async Event %d\n", event.event_type);
            break;
        }

        ibv_ack_async_event(&event);
    }

    return NULL;
}

/*
 * Function:    get_alt_dlid_from_private_data
 *
 * Input:
 *      event  The RDMA event containing private data
 *
 * Output:
 *      dlid   The DLID that was sent in the private data
 *
 * Returns:
```

```
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Takes the private data sent from the remote side and returns the
 *      destination LID that was contained in the private data
 */
int get_alt_dlid_from_private_data(struct rdma_cm_event *event, uint16_t *dlid)
{
    if (event->param.conn.private_data_len < 4) {
        printf("unexpected private data len: %d",
               event->param.conn.private_data_len);
        return -1;
    }

    *dlid = ntohs(*((uint16_t *) event->param.conn.private_data));
    return 0;
}

/*
 * Function:    get_alt_port_details
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      First, query the device to determine if path migration is supported.
 *      Next, queries all the ports on the device to determine if there is
 *      different port than the current one to use as an alternate port. If so,
 *      copy the port number and dlid to the context so they can be used when
 *      the alternate path is loaded.
 *
 * Note:
 *      This function assumes that if another port is found in the active state,
 *      that the port is connected to the same subnet as the initial port and
 *      that there is a route to the other hosts alternate port.
 */
int get_alt_port_details(struct context *ctx)
{
    int ret, i;
    struct ibv_qp_attr qp_attr;
    struct ibv_qp_init_attr qp_init_attr;
    struct ibv_device_attr dev_attr;

    /* This example assumes the alternate port we want to use is on the same
     * HCA. Ports from other HCAs can be used as alternate paths as well. Get
     * a list of devices using ibv_get_device_list or rdma_get_devices.*/
    ret = ibv_query_device(ctx->id->verbs, &dev_attr);
    if (ret) {
        VERB_ERR("ibv_query_device", ret);
        return ret;
    }

    /* Verify the APM is supported by the HCA */
    if (!(dev_attr.device_cap_flags | IBV_DEVICE_AUTO_PATH_MIG)) {
        printf("device does not support auto path migration!\n");
        return -1;
    }

    /* Query the QP to determine which port we are bound to */
    ret = ibv_query_qp(ctx->id->qp, &qp_attr, 0, &qp_init_attr);
    if (ret) {
        VERB_ERR("ibv_query_qp", ret);
        return ret;
    }

    for (i = 1; i <= dev_attr.phys_port_cnt; i++) {
        /* Query all ports until we find one in the active state that is
         * not the port we are currently connected to. */

        struct ibv_port_attr port_attr;
        ret = ibv_query_port(ctx->id->verbs, i, &port_attr);
        if (ret) {
            VERB_ERR("ibv_query_device", ret);
            return ret;
        }

        if (port_attr.state == IBV_PORT_ACTIVE) {
            ctx->my_alt_dlid = port_attr.lid;
            ctx->alt_srcport = i;
            if (qp_attr.port_num != i)
                break;
        }
    }

    return 0;
}

/*
 * Function:    load_alt_path
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
```

```
 * Description:
 *      Uses ibv_modify_qp to load the alternate path information and set the
 *      path migration state to rearm.
 */
int load_alt_path(struct context *ctx)
{
    int ret;
    struct ibv_qp_attr qp_attr;
    struct ibv_qp_init_attr qp_init_attr;

    /* query to get the current attributes of the qp */
    ret = ibv_query_qp(ctx->id->qp, &qp_attr, 0, &qp_init_attr);
    if (ret) {
        VERB_ERR("ibv_query_qp", ret);
        return ret;
    }

    /* initialize the alternate path attributes with the current path
     * attributes */
    memcpy(&qp_attr.alt_ah_attr, &qp_attr.ah_attr, sizeof (struct ibv_ah_attr));

    /* set the alt path attributes to some basic values */
    qp_attr.alt_pkey_index = qp_attr.pkey_index;
    qp_attr.alt_timeout = qp_attr.timeout;
    qp_attr.path_mig_state = IBV_MIG_REARM;

    /* if an alternate path was supplied, set the source port and the dlid */
    if (ctx->alt_srcport)
        qp_attr.alt_port_num = ctx->alt_srcport;
    else
        qp_attr.alt_port_num = qp_attr.port_num;

    if (ctx->alt_dlid)
        qp_attr.alt_ah_attr.dlid = ctx->alt_dlid;

    printf("loading alt path - local port: %d, dlid: %d\n",
           qp_attr.alt_port_num, qp_attr.alt_ah_attr.dlid);

    ret = ibv_modify_qp(ctx->id->qp, &qp_attr,
                        IBV_QP_ALT_PATH | IBV_QP_PATH_MIG_STATE);
    if (ret) {
        VERB_ERR("ibv_modify_qp", ret);
        return ret;
    }
}

/*
 * Function:    reg_mem
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Registers memory regions to use for our data transfer
 */
int reg_mem(struct context *ctx)
{
    ctx->send_buf = (char *) malloc(ctx->msg_length);
    memset(ctx->send_buf, 0x12, ctx->msg_length);

    ctx->recv_buf = (char *) malloc(ctx->msg_length);
    memset(ctx->recv_buf, 0x00, ctx->msg_length);

    ctx->send_mr = rdma_reg_msgs(ctx->id, ctx->send_buf, ctx->msg_length);
    if (!ctx->send_mr) {
        VERB_ERR("rdma_reg_msgs", -1);
        return -1;
    }

    ctx->recv_mr = rdma_reg_msgs(ctx->id, ctx->recv_buf, ctx->msg_length);
    if (!ctx->recv_mr) {
        VERB_ERR("rdma_reg_msgs", -1);
        return -1;
    }

    return 0;
}

/*
 * Function:    getaddrinfo_and_create_ep
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Gets the address information and creates our endpoint
 */
int getaddrinfo_and_create_ep(struct context *ctx)
{
    int ret;
    struct rdma_addrinfo *rai, hints;
    struct ibv_qp_init_attr qp_init_attr;
```

```
    memset(&hints, 0, sizeof (hints));
    hints.ai_port_space = RDMA_PS_TCP;
    if (ctx->server == 1)
        hints.ai_flags = RAI_PASSIVE; /* this makes it a server */

    printf("rdma_getaddrinfo\n");
    ret = rdma_getaddrinfo(ctx->server_name, ctx->server_port, &hints, &rai);
    if (ret) {
        VERB_ERR("rdma_getaddrinfo", ret);
        return ret;
    }

    memset(&qp_init_attr, 0, sizeof (qp_init_attr));

    qp_init_attr.cap.max_send_wr = 1;
    qp_init_attr.cap.max_recv_wr = 1;
    qp_init_attr.cap.max_send_sge = 1;
    qp_init_attr.cap.max_recv_sge = 1;

    printf("rdma_create_ep\n");
    ret = rdma_create_ep(&ctx->id, rai, NULL, &qp_init_attr);
    if (ret) {
        VERB_ERR("rdma_create_ep", ret);
        return ret;
    }

    rdma_freeaddrinfo(rai);

    return 0;
}

/*
 * Function:    get_connect_request
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Wait for a connect request from the client
 */
int get_connect_request(struct context *ctx)
{
    int ret;

    printf("rdma_listen\n");
    ret = rdma_listen(ctx->id, 4);
    if (ret) {
        VERB_ERR("rdma_listen", ret);
        return ret;
    }

    ctx->listen_id = ctx->id;

    printf("rdma_get_request\n");
    ret = rdma_get_request(ctx->listen_id, &ctx->id);
    if (ret) {
        VERB_ERR("rdma_get_request", ret);
        return ret;
    }

    if (ctx->id->event->event != RDMA_CM_EVENT_CONNECT_REQUEST) {
        printf("unexpected event: %s",
                rdma_event_str(ctx->id->event->event));
        return ret;
    }

    /* If the alternate path info was not set on the command line, get
     * it from the private data */
    if (ctx->alt_dlid == 0 && ctx->alt_srcport == 0) {
        ret = get_alt_dlid_from_private_data(ctx->id->event, &ctx->alt_dlid);
        if (ret) {
            return ret;
        }
    }

    return 0;
}

/*
 * Function:    establish_connection
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Create the connection. For the client, call rdma_connect. For the
 *      server, the connect request was already received, so just do
 *      rdma_accept to complete the connection.
 */
int establish_connection(struct context *ctx)
{
    int ret;
    uint16_t private_data;
```

```
    struct rdma_conn_param conn_param;

    /* post a receive to catch the first send */
    ret = rdma_post_recv(ctx->id, NULL, ctx->recv_buf, ctx->msg_length,
                         ctx->recv_mr);
    if (ret) {
        VERB_ERR("rdma_post_recv", ret);
        return ret;
    }

    /* send the dlid for the alternate port in the private data */
    private_data = htons(ctx->my_alt_dlid);

    memset(&conn_param, 0, sizeof (conn_param));
    conn_param.private_data_len = sizeof (int);
    conn_param.private_data = &private_data;
    conn_param.responder_resources = 2;
    conn_param.initiator_depth = 2;
    conn_param.retry_count = 5;
    conn_param.rnr_retry_count = 5;

    if (ctx->server) {
        printf("rdma_accept\n");
        ret = rdma_accept(ctx->id, &conn_param);
        if (ret) {
            VERB_ERR("rdma_accept", ret);
            return ret;
        }
    }
    else {
        printf("rdma_connect\n");
        ret = rdma_connect(ctx->id, &conn_param);
        if (ret) {
            VERB_ERR("rdma_connect", ret);
            return ret;
        }

        if (ctx->id->event->event != RDMA_CM_EVENT_ESTABLISHED) {
            printf("unexpected event: %s",
                   rdma_event_str(ctx->id->event->event));
            return -1;
        }

        /* If the alternate path info was not set on the command line, get
         * it from the private data */
        if (ctx->alt_dlid == 0 && ctx->alt_srcport == 0) {
            ret = get_alt_dlid_from_private_data(ctx->id->event,
                                                 &ctx->alt_dlid);
            if (ret)
                return ret;
        }
    }

    return 0;
}

/*
 * Function:    send_msg
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Performs an Send and gets the completion
 *
 */
int send_msg(struct context *ctx)
{
    int ret;
    struct ibv_wc wc;

    ret = rdma_post_send(ctx->id, NULL, ctx->send_buf, ctx->msg_length,
                         ctx->send_mr, IBV_SEND_SIGNALED);
    if (ret) {
        VERB_ERR("rdma_send_recv", ret);
        return ret;
    }

    ret = rdma_get_send_comp(ctx->id, &wc);
    if (ret < 0) {
        VERB_ERR("rdma_get_send_comp", ret);
        return ret;
    }

    return 0;
}

/*
 * Function:    recv_msg
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
```

```
 * Description:
 *      Waits for a receive completion and posts a new receive buffer
 */
int recv_msg(struct context *ctx)
{
    int ret;
    struct ibv_wc wc;

    ret = rdma_get_recv_comp(ctx->id, &wc);
    if (ret < 0) {
        VERB_ERR("rdma_get_recv_comp", ret);
        return ret;
    }

    ret = rdma_post_recv(ctx->id, NULL, ctx->recv_buf, ctx->msg_length,
                         ctx->recv_mr);
    if (ret) {
        VERB_ERR("rdma_post_recv", ret);
        return ret;
    }

    return 0;
}

/*
 * Function:    main
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *
 */
int main(int argc, char** argv)
{
    int ret, op, i, send_cnt, recv_cnt;
    struct context ctx;
    struct ibv_qp_attr qp_attr;

    memset(&ctx, 0, sizeof (ctx));
    memset(&qp_attr, 0, sizeof (qp_attr));

    ctx.server = 0;
    ctx.server_port = DEFAULT_PORT;
    ctx.msg_count = DEFAULT_MSG_COUNT;
    ctx.msg_length = DEFAULT_MSG_LENGTH;
    ctx.msec_delay = DEFAULT_MSEC_DELAY;
    ctx.alt_dlid = 0;
    ctx.alt_srcport = 0;
    ctx.migrate_after = -1;

    while ((op = getopt(argc, argv, "sa:p:c:l:d:r:m:")) != -1) {
        switch (op) {
        case 's':
            ctx.server = 1;
            break;
        case 'a':
            ctx.server_name = optarg;
            break;
        case 'p':
            ctx.server_port = optarg;
            break;
        case 'c':
            ctx.msg_count = atoi(optarg);
            break;
        case 'l':
            ctx.msg_length = atoi(optarg);
            break;
        case 'd':
            ctx.alt_dlid = atoi(optarg);
            break;
        case 'r':
            ctx.alt_srcport = atoi(optarg);
            break;
        case 'm':
            ctx.migrate_after = atoi(optarg);
            break;
        case 'w':
            ctx.msec_delay = atoi(optarg);
            break;
        default:
            printf("usage: %s [-s or -a required]\n", argv[0]);
            printf("\t[-s[erver mode]\n");
            printf("\t[-a ip_address]\n");
            printf("\t[-p port_number]\n");
            printf("\t[-c msg_count]\n");
            printf("\t[-l msg_length]\n");
            printf("\t[-d alt_dlid] (requires -r)\n");
            printf("\t[-r alt_srcport] (requires -d)\n");
            printf("\t[-m num_iterations_then_migrate] (client only)\n");
            printf("\t[-w msec_wait_between_sends]\n");
            exit(1);
        }
    }

    printf("mode:       %s\n", (ctx.server) ? "server" : "client");
    printf("address:    %s\n", (!ctx.server_name) ? "NULL" : ctx.server_name);
    printf("port:       %s\n", ctx.server_port);
    printf("count:      %d\n", ctx.msg_count);
```

```
    printf("length:     %d\n", ctx.msg_length);
    printf("alt_dlid:   %d\n", ctx.alt_dlid);
    printf("alt_port:   %d\n", ctx.alt_srcport);
    printf("mig_after:  %d\n", ctx.migrate_after);
    printf("msec_wait:  %d\n", ctx.msec_delay);
    printf("\n");

    if (!ctx.server && !ctx.server_name) {
        printf("server address must be specified for client mode\n");
        exit(1);
    }

    /* both of these must be set or neither should be set */
    if (!((ctx.alt_dlid > 0 && ctx.alt_srcport > 0) ||
        (ctx.alt_dlid == 0 && ctx.alt_srcport == 0))) {
        printf("-d and -r must be used together\n");
        exit(1);
    }

    if (ctx.migrate_after > ctx.msg_count) {
        printf("num_iterations_then_migrate must be less than msg_count\n");
        exit(1);
    }

    ret = getaddrinfo_and_create_ep(&ctx);
    if (ret)
        goto out;

    if (ctx.server) {
        ret = get_connect_request(&ctx);
        if (ret)
            goto out;
    }

    /* only query for alternate port if information was not specified on the
     * command line */
    if (ctx.alt_dlid == 0 && ctx.alt_srcport == 0) {
        ret = get_alt_port_details(&ctx);
        if (ret)
            goto out;
    }

    /* create a thread to handle async events */
    pthread_create(&ctx.async_event_thread, NULL, async_event_thread, &ctx);

    ret = reg_mem(&ctx);
    if (ret)
        goto out;

    ret = establish_connection(&ctx);

    /* load the alternate path after the connection was created. This can be
     * done at connection time, but the connection must be created and
     * established using all ib verbs */
    ret = load_alt_path(&ctx);
    if (ret)
        goto out;

    send_cnt = recv_cnt = 0;

    for (i = 0; i < ctx.msg_count; i++) {
        if (ctx.server) {
            if (recv_msg(&ctx))
                break;

            printf("recv: %d\n", ++recv_cnt);
        }

        if (ctx.msec_delay > 0)
            usleep(ctx.msec_delay * 1000);

        if (send_msg(&ctx))
            break;

        printf("send: %d\n", ++send_cnt);

        if (!ctx.server) {
            if (recv_msg(&ctx))
                break;

            printf("recv: %d\n", ++recv_cnt);
        }

        /* migrate the path manually if desired after the specified number of
         * sends */
        if (!ctx.server && i == ctx.migrate_after) {
            qp_attr.path_mig_state = IBV_MIG_MIGRATED;
            ret = ibv_modify_qp(ctx.id->qp, &qp_attr, IBV_QP_PATH_MIG_STATE);
            if (ret) {
                VERB_ERR("ibv_modify_qp", ret);
                goto out;
            }
        }
    }

    rdma_disconnect(ctx.id);

out:
    if (ctx.send_mr)
        rdma_dereg_mr(ctx.send_mr);

    if (ctx.recv_mr)
        rdma_dereg_mr(ctx.recv_mr);

    if (ctx.id)
```

```
        rdma_destroy_ep(ctx.id);

    if (ctx.listen_id)
        rdma_destroy_ep(ctx.listen_id);

    if (ctx.send_buf)
        free(ctx.send_buf);

    if (ctx.recv_buf)
        free(ctx.recv_buf);

    return ret;
}
```

9.5.2. Multicast Code Example Using RDMA CM

```
/*
 * Compile Command:
 * gcc mc.c -o mc -libverbs -lrdmacm
 *
 * Description:
 * Both the sender and receiver create a UD Queue Pair and join the specified
 * multicast group (ctx.mcast_addr). If the join is successful, the sender must
 * create an Address Handle (ctx.ah). The sender then posts the specified
 * number of sends (ctx.msg_count) to the multicast group. The receiver waits
 * to receive each one of the sends and then both sides leave the multicast
 * group and cleanup resources.
 *
 * Running the Example:
 * The executable can operate as either the sender or receiver application. It
 * can be demonstrated on a simple fabric of two nodes with the sender
 * application running on one node and the receiver application running on the
 * other. Each node must be configured to support IPoIB and the IB interface
 * (ex. ib0) must be assigned an IP Address. Finally, the fabric must be
 * initialized using OpenSM.
 *
 * Receiver (-m is the multicast address, often the IP of the receiver):
 * ./mc -m 192.168.1.12
 *
 * Sender (-m is the multicast address, often the IP of the receiver):
 * ./mc -s -m 192.168.1.12
 *
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <getopt.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <rdma/rdma_verbs.h>

#define VERB_ERR(verb, ret) \
        fprintf(stderr, "%s returned %d errno %d\n", verb, ret, errno)

/* Default parameter values */
#define DEFAULT_PORT "51216"
#define DEFAULT_MSG_COUNT 4
#define DEFAULT_MSG_LENGTH 64

/* Resources used in the example */
struct context
{
    /* User parameters */
    int sender;
    char *bind_addr;
    char *mcast_addr;
    char *server_port;
    int msg_count;
    int msg_length;

    /* Resources */
    struct sockaddr mcast_sockaddr;
    struct rdma_cm_id *id;
    struct rdma_event_channel *channel;
    struct ibv_pd *pd;
    struct ibv_cq *cq;
    struct ibv_mr *mr;
    char *buf;
    struct ibv_ah *ah;
    uint32_t remote_qpn;
    uint32_t remote_qkey;
    pthread_t cm_thread;
};

/*
 * Function:    cm_thread
 *
 * Input:
 *      arg     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      NULL
 *
 * Description:
 *      Reads any CM events that occur during the sending of data
 *      and prints out the details of the event
```

```
 */
static void *cm_thread(void *arg)
{
    struct rdma_cm_event *event;
    int ret;

    struct context *ctx = (struct context *) arg;

    while (1) {
        ret = rdma_get_cm_event(ctx->channel, &event);
        if (ret) {
            VERB_ERR("rdma_get_cm_event", ret);
            break;
        }

        printf("event %s, status %d\n",
               rdma_event_str(event->event), event->status);

        rdma_ack_cm_event(event);
    }

    return NULL;
}

/*
 * Function:    get_cm_event
 *
 * Input:
 *      channel The event channel
 *      type    The event type that is expected
 *
 * Output:
 *      out_ev  The event will be passed back to the caller, if desired
 *              Set this to NULL and the event will be acked automatically
 *              Otherwise the caller must ack the event using rdma_ack_cm_event
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Waits for the next CM event and check that is matches the expected
 *      type.
 */
int get_cm_event(struct rdma_event_channel *channel,
                 enum rdma_cm_event_type type,
                 struct rdma_cm_event **out_ev)
{
    int ret = 0;
    struct rdma_cm_event *event = NULL;

    ret = rdma_get_cm_event(channel, &event);
    if (ret) {
        VERB_ERR("rdma_resolve_addr", ret);
        return -1;
    }

    /* Verify the event is the expected type */
    if (event->event != type) {
        printf("event: %s, status: %d\n",
               rdma_event_str(event->event), event->status);
        ret = -1;
    }

    /* Pass the event back to the user if requested */
    if (!out_ev)
        rdma_ack_cm_event(event);
    else
        *out_ev = event;

    return ret;
}

/*
 * Function:    resolve_addr
 *
 * Input:
 *      ctx     The context structure
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Resolves the multicast address and also binds to the source address
 *      if one was provided in the context
 */
int resolve_addr(struct context *ctx)
{
    int ret;
    struct rdma_addrinfo *bind_rai = NULL;
    struct rdma_addrinfo *mcast_rai = NULL;
    struct rdma_addrinfo hints;

    memset(&hints, 0, sizeof (hints));
    hints.ai_port_space = RDMA_PS_UDP;

    if (ctx->bind_addr) {
        hints.ai_flags = RAI_PASSIVE;

        ret = rdma_getaddrinfo(ctx->bind_addr, NULL, &hints, &bind_rai);
        if (ret) {
            VERB_ERR("rdma_getaddrinfo (bind)", ret);
            return ret;
```

```
        }
    }

    hints.ai_flags = 0;

    ret = rdma_getaddrinfo(ctx->mcast_addr, NULL, &hints, &mcast_rai);
    if (ret) {
        VERB_ERR("rdma_getaddrinfo (mcast)", ret);
        return ret;
    }

    if (ctx->bind_addr) {
        /* bind to a specific adapter if requested to do so */
        ret = rdma_bind_addr(ctx->id, bind_rai->ai_src_addr);
        if (ret) {
            VERB_ERR("rdma_bind_addr", ret);
            return ret;
        }

        /* A PD is created when we bind. Copy it to the context so it can
         * be used later on */
        ctx->pd = ctx->id->pd;
    }

    ret = rdma_resolve_addr(ctx->id, (bind_rai) ? bind_rai->ai_src_addr : NULL,
                            mcast_rai->ai_dst_addr, 2000);
    if (ret) {
        VERB_ERR("rdma_resolve_addr", ret);
        return ret;
    }

    ret = get_cm_event(ctx->channel, RDMA_CM_EVENT_ADDR_RESOLVED, NULL);
    if (ret) {
        return ret;
    }

    memcpy(&ctx->mcast_sockaddr,
           mcast_rai->ai_dst_addr,
           sizeof (struct sockaddr));

    return 0;
}

/*
 * Function:    create_resources
 *
 * Input:
 *      ctx     The context structure
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Creates the PD, CQ, QP and MR
 */
int create_resources(struct context *ctx)
{
    int ret, buf_size;
    struct ibv_qp_init_attr attr;

    memset(&attr, 0, sizeof (attr));

    /* If we are bound to an address, then a PD was already allocated
     * to the CM ID */
    if (!ctx->pd) {
        ctx->pd = ibv_alloc_pd(ctx->id->verbs);
        if (!ctx->pd) {
            VERB_ERR("ibv_alloc_pd", -1);
            return ret;
        }
    }

    ctx->cq = ibv_create_cq(ctx->id->verbs, 2, 0, 0, 0);
    if (!ctx->cq) {
        VERB_ERR("ibv_create_cq", -1);
        return ret;
    }

    attr.qp_type = IBV_QPT_UD;
    attr.send_cq = ctx->cq;
    attr.recv_cq = ctx->cq;
    attr.cap.max_send_wr = ctx->msg_count;
    attr.cap.max_recv_wr = ctx->msg_count;
    attr.cap.max_send_sge = 1;
    attr.cap.max_recv_sge = 1;

    ret = rdma_create_qp(ctx->id, ctx->pd, &attr);
    if (ret) {
        VERB_ERR("rdma_create_qp", ret);
        return ret;
    }

    /* The receiver must allow enough space in the receive buffer for
     * the GRH */
    buf_size = ctx->msg_length + (ctx->sender ? 0 : sizeof (struct ibv_grh));

    ctx->buf = calloc(1, buf_size);
    memset(ctx->buf, 0x00, buf_size);

    /* Register our memory region */
    ctx->mr = rdma_reg_msgs(ctx->id, ctx->buf, buf_size);
    if (!ctx->mr) {
```

```
        VERB_ERR("rdma_reg_msgs", -1);
        return -1;
    }

    return 0;
}

/*
 * Function:    destroy_resources
 *
 * Input:
 *      ctx     The context structure
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Destroys AH, QP, CQ, MR, PD and ID
 */
void destroy_resources(struct context *ctx)
{
    if (ctx->ah)
        ibv_destroy_ah(ctx->ah);

    if (ctx->id->qp)
        rdma_destroy_qp(ctx->id);

    if (ctx->cq)
        ibv_destroy_cq(ctx->cq);

    if (ctx->mr)
        rdma_dereg_mr(ctx->mr);

    if (ctx->buf)
        free(ctx->buf);

    if (ctx->pd && ctx->id->pd == NULL)
        ibv_dealloc_pd(ctx->pd);

    rdma_destroy_id(ctx->id);
}

/*
 * Function:    post_send
 *
 * Input:
 *      ctx     The context structure
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Posts a UD send to the multicast address
 */
int post_send(struct context *ctx)
{
    int ret;
    struct ibv_send_wr wr, *bad_wr;
    struct ibv_sge sge;

    memset(ctx->buf, 0x12, ctx->msg_length); /* set the data to non-zero */

    sge.length = ctx->msg_length;
    sge.lkey = ctx->mr->lkey;
    sge.addr = (uint64_t) ctx->buf;

    /* Multicast requires that the message is sent with immediate data
     * and that the QP number is the contents of the immediate data */
    wr.next = NULL;
    wr.sg_list = &sge;
    wr.num_sge = 1;
    wr.opcode = IBV_WR_SEND_WITH_IMM;
    wr.send_flags = IBV_SEND_SIGNALED;
    wr.wr_id = 0;
    wr.imm_data = htonl(ctx->id->qp->qp_num);
    wr.wr.ud.ah = ctx->ah;
    wr.wr.ud.remote_qpn = ctx->remote_qpn;
    wr.wr.ud.remote_qkey = ctx->remote_qkey;

    ret = ibv_post_send(ctx->id->qp, &wr, &bad_wr);
    if (ret) {
        VERB_ERR("ibv_post_send", ret);
        return -1;
    }

    return 0;
}

/*
 * Function:    get_completion
 *
 * Input:
 *      ctx     The context structure
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
```

```
 *
 * Description:
 *      Waits for a completion and verifies that the operation was successful
 */
int get_completion(struct context *ctx)
{
    int ret;
    struct ibv_wc wc;

    do {
        ret = ibv_poll_cq(ctx->cq, 1, &wc);
        if (ret < 0) {
            VERB_ERR("ibv_poll_cq", ret);
            return -1;
        }
    }
    while (ret == 0);

    if (wc.status != IBV_WC_SUCCESS) {
        printf("work completion status %s\n",
               ibv_wc_status_str(wc.status));
        return -1;
    }

    return 0;
}

/*
 * Function:    main
 *
 * Input:
 *      argc    The number of arguments
 *      argv    Command line arguments
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Main program to demonstrate multicast functionality.
 *      Both the sender and receiver create a UD Queue Pair and join the
 *      specified multicast group (ctx.mcast_addr). If the join is successful,
 *      the sender must create an Address Handle (ctx.ah). The sender then posts
 *      the specified number of sends (ctx.msg_count) to the multicast group.
 *      The receiver waits to receive each one of the sends and then both sides
 *      leave the multicast group and cleanup resources.
 */
int main(int argc, char** argv)
{
    int ret, op, i;
    struct context ctx;
    struct ibv_port_attr port_attr;
    struct rdma_cm_event *event;
    char buf[40];

    memset(&ctx, 0, sizeof (ctx));

    ctx.sender = 0;
    ctx.msg_count = DEFAULT_MSG_COUNT;
    ctx.msg_length = DEFAULT_MSG_LENGTH;
    ctx.server_port = DEFAULT_PORT;

    // Read options from command line
    while ((op = getopt(argc, argv, "shb:m:p:c:l:")) != -1) {
        switch (op) {
        case 's':
            ctx.sender = 1;
            break;
        case 'b':
            ctx.bind_addr = optarg;
            break;
        case 'm':
            ctx.mcast_addr = optarg;
            break;
        case 'p':
            ctx.server_port = optarg;
            break;
        case 'c':
            ctx.msg_count = atoi(optarg);
            break;
        case 'l':
            ctx.msg_length = atoi(optarg);
            break;
        default:
            printf("usage: %s -m mc_address\n", argv[0]);
            printf("\t[-s[ender mode]\n");
            printf("\t[-b bind_address]\n");
            printf("\t[-p port_number]\n");
            printf("\t[-c msg_count]\n");
            printf("\t[-l msg_length]\n");
            exit(1);
        }
    }
    if(ctx.mcast_addr == NULL) {
        printf("multicast address must be specified with -m\n");
        exit(1);
    }

    ctx.channel = rdma_create_event_channel();
    if (!ctx.channel) {
        VERB_ERR("rdma_create_event_channel", -1);
        exit(1);
    }
```

```
    ret = rdma_create_id(ctx.channel, &ctx.id, NULL, RDMA_PS_UDP);
    if (ret) {
        VERB_ERR("rdma_create_id", -1);
        exit(1);
    }

    ret = resolve_addr(&ctx);
    if (ret)
        goto out;

    /* Verify that the buffer length is not larger than the MTU */
    ret = ibv_query_port(ctx.id->verbs, ctx.id->port_num, &port_attr);
    if (ret) {
        VERB_ERR("ibv_query_port", ret);
        goto out;
    }

    if (ctx.msg_length > (1 << port_attr.active_mtu + 7)) {
        printf("buffer length %d is larger then active mtu %d\n",
               ctx.msg_length, 1 << (port_attr.active_mtu + 7));
        goto out;
    }

    ret = create_resources(&ctx);
    if (ret)
        goto out;

    if (!ctx.sender) {
        for (i = 0; i < ctx.msg_count; i++) {
            ret = rdma_post_recv(ctx.id, NULL, ctx.buf,
                                 ctx.msg_length + sizeof (struct ibv_grh),
                                 ctx.mr);
            if (ret) {
                VERB_ERR("rdma_post_recv", ret);
                goto out;
            }
        }
    }

    /* Join the multicast group */
    ret = rdma_join_multicast(ctx.id, &ctx.mcast_sockaddr, NULL);
    if (ret) {
        VERB_ERR("rdma_join_multicast", ret);
        goto out;
    }

    /* Verify that we successfully joined the multicast group */
    ret = get_cm_event(ctx.channel, RDMA_CM_EVENT_MULTICAST_JOIN, &event);
    if (ret)
        goto out;

    inet_ntop(AF_INET6, event->param.ud.ah_attr.grh.dgid.raw, buf, 40);
    printf("joined dgid: %s, mlid 0x%x, sl %d\n", buf,
           event->param.ud.ah_attr.dlid, event->param.ud.ah_attr.sl);

    ctx.remote_qpn = event->param.ud.qp_num;
    ctx.remote_qkey = event->param.ud.qkey;

    if (ctx.sender) {
        /* Create an address handle for the sender */
        ctx.ah = ibv_create_ah(ctx.pd, &event->param.ud.ah_attr);
        if (!ctx.ah) {
            VERB_ERR("ibv_create_ah", -1);
            goto out;
        }
    }

    rdma_ack_cm_event(event);

    /* Create a thread to handle any CM events while messages are exchanged */
    pthread_create(&ctx.cm_thread, NULL, cm_thread, &ctx);

    if (!ctx.sender)
        printf("waiting for messages...\n");

    for (i = 0; i < ctx.msg_count; i++) {
        if (ctx.sender) {
            ret = post_send(&ctx);
            if (ret)
                goto out;
        }

        ret = get_completion(&ctx);
        if (ret)
            goto out;

        if (ctx.sender)
            printf("sent message %d\n", i + 1);
        else
            printf("received message %d\n", i + 1);
    }

out:
    ret = rdma_leave_multicast(ctx.id, &ctx.mcast_sockaddr);
    if (ret)
        VERB_ERR("rdma_leave_multicast", ret);

    destroy_resources(&ctx);

    return ret;
}
```

9.5.3. Shared Received Queue (SRQ)

```
/*
 * Compile Command:
 * gcc srq.c -o srq -libverbs -lrdmacm
 *
 * Description:
 * Both the client and server use an SRQ. A number of Queue Pairs (QPs) are
 * created (ctx.qp_count) and each QP uses the SRQ. The connection between the
 * client and server is established using the IP address details passed on the
 * command line. After the connection is established, the client starts
 * blasting sends to the server and stops when the maximum work requests
 * (ctx.max_wr) have been sent. When the server has received all the sends, it
 * performs a send to the client to tell it to continue. The process repeats
 * until the number of requested number of sends (ctx.msg_count) have been
 * performed.
 *
 * Running the Example:
 * The executable can operate as either the client or server application. It
 * can be demonstrated on a simple fabric of two nodes with the server
 * application running on one node and the client application running on the
 * other. Each node must be configured to support IPoIB and the IB interface
 * (ex. ib0) must be assigned an IP Address. Finally, the fabric must be
 * initialized using OpenSM.
 *
 * Server (-a is IP of local interface):
 * ./srq -s -a 192.168.1.12
 *
 * Client (-a is IP of remote interface):
 * ./srq -a 192.168.1.12
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <getopt.h>
#include <rdma/rdma_verbs.h>

#define VERB_ERR(verb, ret) \
        fprintf(stderr, "%s returned %d errno %d\n", verb, ret, errno)

/* Default parameters values */
#define DEFAULT_PORT "51216"
#define DEFAULT_MSG_COUNT 100
#define DEFAULT_MSG_LENGTH 100000
#define DEFAULT_QP_COUNT 4
#define DEFAULT_MAX_WR 64

/* Resources used in the example */
struct context
{
    /* User parameters */
    int server;
    char *server_name;
    char *server_port;
    int msg_count;
    int msg_length;
    int qp_count;
    int max_wr;

    /* Resources */
    struct rdma_cm_id *srq_id;
    struct rdma_cm_id *listen_id;
    struct rdma_cm_id **conn_id;
    struct ibv_mr *send_mr;
    struct ibv_mr *recv_mr;
    struct ibv_srq *srq;
    struct ibv_cq *srq_cq;
    struct ibv_comp_channel *srq_cq_channel;
    char *send_buf;
    char *recv_buf;
};

/*
 * Function: init_resources
 *
 * Input:
 *      ctx     The context object
 *      rai     The RDMA address info for the connection
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      This function initializes resources that are common to both the client
 *      and server functionality.
 *      It creates our SRQ, registers memory regions, posts receive buffers
 *      and creates a single completion queue that will be used for the receive
 *      queue on each queue pair.
 */
int init_resources(struct context *ctx, struct rdma_addrinfo *rai)
{
    int ret, i;
    struct rdma_cm_id *id;

    /* Create an ID used for creating/accessing our SRQ */
    ret = rdma_create_id(NULL, &ctx->srq_id, NULL, RDMA_PS_TCP);
    if (ret) {
```

```
        VERB_ERR("rdma_create_id", ret);
        return ret;
    }

    /* We need to bind the ID to a particular RDMA device
     * This is done by resolving the address or binding to the address */
    if (ctx->server == 0) {
        ret = rdma_resolve_addr(ctx->srq_id, NULL, rai->ai_dst_addr, 1000);
        if (ret) {
            VERB_ERR("rdma_resolve_addr", ret);
            return ret;
        }
    }
    else {
        ret = rdma_bind_addr(ctx->srq_id, rai->ai_src_addr);
        if (ret) {
            VERB_ERR("rdma_bind_addr", ret);
            return ret;
        }
    }

    /* Create the memory regions being used in this example */
    ctx->recv_mr = rdma_reg_msgs(ctx->srq_id, ctx->recv_buf, ctx->msg_length);
    if (!ctx->recv_mr) {
        VERB_ERR("rdma_reg_msgs", -1);
        return -1;
    }

    ctx->send_mr = rdma_reg_msgs(ctx->srq_id, ctx->send_buf, ctx->msg_length);
    if (!ctx->send_mr) {
        VERB_ERR("rdma_reg_msgs", -1);
        return -1;
    }

    /* Create our shared receive queue */
    struct ibv_srq_init_attr srq_attr;
    memset(&srq_attr, 0, sizeof (srq_attr));
    srq_attr.attr.max_wr = ctx->max_wr;
    srq_attr.attr.max_sge = 1;

    ret = rdma_create_srq(ctx->srq_id, NULL, &srq_attr);
    if (ret) {
        VERB_ERR("rdma_create_srq", ret);
        return -1;
    }

    /* Save the SRQ in our context so we can assign it to other QPs later */
    ctx->srq = ctx->srq_id->srq;

    /* Post our receive buffers on the SRQ */
    for (i = 0; i < ctx->max_wr; i++) {
        ret = rdma_post_recv(ctx->srq_id, NULL, ctx->recv_buf, ctx->msg_length,
                             ctx->recv_mr);
        if (ret) {
            VERB_ERR("rdma_post_recv", ret);
            return ret;
        }
    }

    /* Create a completion channel to use with the SRQ CQ */
    ctx->srq_cq_channel = ibv_create_comp_channel(ctx->srq_id->verbs);
    if (!ctx->srq_cq_channel) {
        VERB_ERR("ibv_create_comp_channel", -1);
        return -1;
    }

    /* Create a CQ to use for all connections (QPs) that use the SRQ */
    ctx->srq_cq = ibv_create_cq(ctx->srq_id->verbs, ctx->max_wr, NULL,
                                ctx->srq_cq_channel, 0);
    if (!ctx->srq_cq) {
        VERB_ERR("ibv_create_cq", -1);
        return -1;
    }

    /* Make sure that we get notified on the first completion */
    ret = ibv_req_notify_cq(ctx->srq_cq, 0);
    if (ret) {
        VERB_ERR("ibv_req_notify_cq", ret);
        return ret;
    }

    return 0;
}

/*
 * Function:    destroy_resources
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      This function cleans up resources used by the application
 */
void destroy_resources(struct context *ctx)
{
    int i;

    if (ctx->conn_id) {
        for (i = 0; i < ctx->qp_count; i++) {
```

```
            if (ctx->conn_id[i]) {
                if (ctx->conn_id[i]->qp &&
                    ctx->conn_id[i]->qp->state == IBV_QPS_RTS) {
                    rdma_disconnect(ctx->conn_id[i]);
                }
                rdma_destroy_qp(ctx->conn_id[i]);
                rdma_destroy_id(ctx->conn_id[i]);
            }
        }

        free(ctx->conn_id);
    }

    if (ctx->recv_mr)
        rdma_dereg_mr(ctx->recv_mr);

    if (ctx->send_mr)
        rdma_dereg_mr(ctx->send_mr);

    if (ctx->recv_buf)
        free(ctx->recv_buf);

    if (ctx->send_buf)
        free(ctx->send_buf);

    if (ctx->srq_cq)
        ibv_destroy_cq(ctx->srq_cq);

    if (ctx->srq_cq_channel)
        ibv_destroy_comp_channel(ctx->srq_cq_channel);

    if (ctx->srq_id) {
        rdma_destroy_srq(ctx->srq_id);
        rdma_destroy_id(ctx->srq_id);
    }
}

/*
 * Function:    await_completion
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Waits for a completion on the SRQ CQ
 *
 */
int await_completion(struct context *ctx)
{
    int ret;
    struct ibv_cq *ev_cq;
    void *ev_ctx;

    /* Wait for a CQ event to arrive on the channel */
    ret = ibv_get_cq_event(ctx->srq_cq_channel, &ev_cq, &ev_ctx);
    if (ret) {
        VERB_ERR("ibv_get_cq_event", ret);
        return ret;
    }

    ibv_ack_cq_events(ev_cq, 1);

    /* Reload the event notification */
    ret = ibv_req_notify_cq(ctx->srq_cq, 0);
    if (ret) {
        VERB_ERR("ibv_req_notify_cq", ret);
        return ret;
    }

    return 0;
}

/*
 * Function:    run_server
 *
 * Input:
 *      ctx     The context object
 *      rai     The RDMA address info for the connection
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Executes the server side of the example
 */
int run_server(struct context *ctx, struct rdma_addrinfo *rai)
{
    int ret, i;
    uint64_t send_count = 0;
    uint64_t recv_count = 0;
    struct ibv_wc wc;
    struct ibv_qp_init_attr qp_attr;

    ret = init_resources(ctx, rai);
    if (ret) {
        printf("init_resources returned %d\n", ret);
```

```
    return ret;
}

/* Use the srq_id as the listen_id since it is already setup */
ctx->listen_id = ctx->srq_id;

ret = rdma_listen(ctx->listen_id, 4);
if (ret) {
    VERB_ERR("rdma_listen", ret);
    return ret;
}

printf("waiting for connection from client...\n");
for (i = 0; i < ctx->qp_count; i++) {
    ret = rdma_get_request(ctx->listen_id, &ctx->conn_id[i]);
    if (ret) {
        VERB_ERR("rdma_get_request", ret);
        return ret;
    }

    /* Create the queue pair */
    memset(&qp_attr, 0, sizeof (qp_attr));

    qp_attr.qp_context = ctx;
    qp_attr.qp_type = IBV_QPT_RC;
    qp_attr.cap.max_send_wr = ctx->max_wr;
    qp_attr.cap.max_recv_wr = ctx->max_wr;
    qp_attr.cap.max_send_sge = 1;
    qp_attr.cap.max_recv_sge = 1;
    qp_attr.cap.max_inline_data = 0;
    qp_attr.recv_cq = ctx->srq_cq;
    qp_attr.srq = ctx->srq;
    qp_attr.sq_sig_all = 0;

    ret = rdma_create_qp(ctx->conn_id[i], NULL, &qp_attr);
    if (ret) {
        VERB_ERR("rdma_create_qp", ret);
        return ret;
    }

    /* Set the new connection to use our SRQ */
    ctx->conn_id[i]->srq = ctx->srq;

    ret = rdma_accept(ctx->conn_id[i], NULL);
    if (ret) {
        VERB_ERR("rdma_accept", ret);
        return ret;
    }
}

while (recv_count < ctx->msg_count) {
    i = 0;
    while (i < ctx->max_wr && recv_count < ctx->msg_count) {
        int ne;

        ret = await_completion(ctx);
        if (ret) {
            printf("await_completion %d\n", ret);
            return ret;
        }

        do {
            ne = ibv_poll_cq(ctx->srq_cq, 1, &wc);
            if (ne < 0) {
                VERB_ERR("ibv_poll_cq", ne);
                return ne;
            }
            else if (ne == 0)
                break;

            if (wc.status != IBV_WC_SUCCESS) {
                printf("work completion status %s\n",
                       ibv_wc_status_str(wc.status));
                return -1;
            }

            recv_count++;
            printf("recv count: %d, qp_num: %d\n", recv_count, wc.qp_num);

            ret = rdma_post_recv(ctx->srq_id, (void *) wc.wr_id,
                                 ctx->recv_buf, ctx->msg_length,
                                 ctx->recv_mr);
            if (ret) {
                VERB_ERR("rdma_post_recv", ret);
                return ret;
            }

            i++;
        }
        while (ne);
    }

    ret = rdma_post_send(ctx->conn_id[0], NULL, ctx->send_buf,
                         ctx->msg_length, ctx->send_mr, IBV_SEND_SIGNALED);
    if (ret) {
        VERB_ERR("rdma_post_send", ret);
        return ret;
    }

    ret = rdma_get_send_comp(ctx->conn_id[0], &wc);
    if (ret <= 0) {
        VERB_ERR("rdma_get_send_comp", ret);
        return -1;
    }
```

```
        send_count++;
        printf("send count: %d\n", send_count);
    }

    return 0;
}

/*
 * Function:    run_client
 *
 * Input:
 *      ctx     The context object
 *      rai     The RDMA address info for the connection
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Executes the client side of the example
 */
int run_client(struct context *ctx, struct rdma_addrinfo *rai)
{
    int ret, i, ne;
    uint64_t send_count = 0;
    uint64_t recv_count = 0;
    struct ibv_wc wc;
    struct ibv_qp_init_attr attr;

    ret = init_resources(ctx, rai);
    if (ret) {
        printf("init_resources returned %d\n", ret);
        return ret;
    }

    for (i = 0; i < ctx->qp_count; i++) {
        memset(&attr, 0, sizeof (attr));

        attr.qp_context = ctx;
        attr.cap.max_send_wr = ctx->max_wr;
        attr.cap.max_recv_wr = ctx->max_wr;
        attr.cap.max_send_sge = 1;
        attr.cap.max_recv_sge = 1;
        attr.cap.max_inline_data = 0;
        attr.recv_cq = ctx->srq_cq;
        attr.srq = ctx->srq;
        attr.sq_sig_all = 0;

        ret = rdma_create_ep(&ctx->conn_id[i], rai, NULL, &attr);
        if (ret) {
            VERB_ERR("rdma_create_ep", ret);
            return ret;
        }

        ret = rdma_connect(ctx->conn_id[i], NULL);
        if (ret) {
            VERB_ERR("rdma_connect", ret);
            return ret;
        }
    }

    while (send_count < ctx->msg_count) {
        for (i = 0; i < ctx->max_wr && send_count < ctx->msg_count; i++) {
            /* perform our send to the server */
            ret = rdma_post_send(ctx->conn_id[i % ctx->qp_count], NULL,
                                 ctx->send_buf, ctx->msg_length, ctx->send_mr,
                                 IBV_SEND_SIGNALED);
            if (ret) {
                VERB_ERR("rdma_post_send", ret);
                return ret;
            }

            ret = rdma_get_send_comp(ctx->conn_id[i % ctx->qp_count], &wc);
            if (ret <= 0) {
                VERB_ERR("rdma_get_send_comp", ret);
                return ret;
            }

            send_count++;
            printf("send count: %d, qp_num: %d\n", send_count, wc.qp_num);
        }

        /* wait for a recv indicating that all buffers were processed */
        ret = await_completion(ctx);
        if (ret) {
            VERB_ERR("await_completion", ret);
            return ret;
        }

        do {
            ne = ibv_poll_cq(ctx->srq_cq, 1, &wc);
            if (ne < 0) {
                VERB_ERR("ibv_poll_cq", ne);
                return ne;
            }
            else if (ne == 0)
                break;

            if (wc.status != IBV_WC_SUCCESS) {
                printf("work completion status %s\n",
                       ibv_wc_status_str(wc.status));
                return -1;
            }
```

```
            recv_count++;
            printf("recv count: %d\n", recv_count);

            ret = rdma_post_recv(ctx->srq_id, (void *) wc.wr_id,
                                 ctx->recv_buf, ctx->msg_length, ctx->recv_mr);
            if (ret) {
                VERB_ERR("rdma_post_recv", ret);
                return ret;
            }
        }
        while (ne);
    }

    return ret;
}

/*
 * Function:    main
 *
 * Input:
 *      argc    The number of arguments
 *      argv    Command line arguments
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Main program to demonstrate SRQ functionality.
 *      Both the client and server use an SRQ. ctx.qp_count number of QPs are
 *      created and each one of them uses the SRQ. After the connection, the
 *      client starts blasting sends to the server upto ctx.max_wr. When the
 *      server has received all the sends, it performs a send to the client to
 *      tell it that it can continue. Process repeats until ctx.msg_count
 *      sends have been performed.
 */
int main(int argc, char** argv)
{
    int ret, op;
    struct context ctx;
    struct rdma_addrinfo *rai, hints;

    memset(&ctx, 0, sizeof (ctx));
    memset(&hints, 0, sizeof (hints));

    ctx.server = 0;
    ctx.server_port = DEFAULT_PORT;
    ctx.msg_count = DEFAULT_MSG_COUNT;
    ctx.msg_length = DEFAULT_MSG_LENGTH;
    ctx.qp_count = DEFAULT_QP_COUNT;
    ctx.max_wr = DEFAULT_MAX_WR;

    /* Read options from command line */
    while ((op = getopt(argc, argv, "sa:p:c:l:q:w:")) != -1) {
        switch (op) {
        case 's':
            ctx.server = 1;
            break;
        case 'a':
            ctx.server_name = optarg;
            break;
        case 'p':
            ctx.server_port = optarg;
            break;
        case 'c':
            ctx.msg_count = atoi(optarg);
            break;
        case 'l':
            ctx.msg_length = atoi(optarg);
            break;
        case 'q':
            ctx.qp_count = atoi(optarg);
            break;
        case 'w':
            ctx.max_wr = atoi(optarg);
            break;
        default:
            printf("usage: %s -a server_address\n", argv[0]);
            printf("\t[-s server mode]\n");
            printf("\t[-p port_number]\n");
            printf("\t[-c msg_count]\n");
            printf("\t[-l msg_length]\n");
            printf("\t[-q qp_count]\n");
            printf("\t[-w max_wr]\n");
            exit(1);
        }
    }

    if (ctx.server_name == NULL) {
        printf("server address required (use -a)!\n");
        exit(1);
    }

    hints.ai_port_space = RDMA_PS_TCP;
    if (ctx.server == 1)
        hints.ai_flags = RAI_PASSIVE; /* this makes it a server */

    ret = rdma_getaddrinfo(ctx.server_name, ctx.server_port, &hints, &rai);
    if (ret) {
        VERB_ERR("rdma_getaddrinfo", ret);
        exit(1);
    }
```

```
    /* allocate memory for our QPs and send/recv buffers */
    ctx.conn_id = (struct rdma_cm_id **) calloc(ctx.qp_count,
                                            sizeof (struct rdma_cm_id *));
    memset(ctx.conn_id, 0, sizeof (ctx.conn_id));

    ctx.send_buf = (char *) malloc(ctx.msg_length);
    memset(ctx.send_buf, 0, ctx.msg_length);
    ctx.recv_buf = (char *) malloc(ctx.msg_length);
    memset(ctx.recv_buf, 0, ctx.msg_length);

    if (ctx.server)
        ret = run_server(&ctx, rai);
    else
        ret = run_client(&ctx, rai);

    destroy_resources(&ctx);
    free(rai);

    return ret;
}
```

9.6. Experimental APIs

9.6.1. Dynamically Connected Transport

The Dynamically Connected (DC) transport provides reliable transport services from a DC Initiator (DCI) to a DC Target (DCT). A DCI can send data to multiple targets on the same or different subnet, and a DCT can simultaneously service traffic from multiple DCIs. No explicit connections are setup by the user, with the target DCT being identified by an address vector similar to that used in UD transport, DCT number, and DC access key.

9.6.1.1. DC Usage Model

1. Query Device: Query the device capabilities to see if the DC transport is supported, and if so, what its characteristics are.
2. Create DCIs: The user creates DCIs (Dynamically Connected Initiators). The number of DCIs depends on the user's strategy for handling concurrent data transmissions.
3. Create DCT: The user defines a DC Access Key and initializes a DCT (Dynamically Connected Target) using this access key.
4. Query: The user can query the DCI with the **ibv_exp_query_qp()** routine, and can query the DCT with the **ibv_exp_query_dct()** routine.
5. Arm DCT: The user can arm the DCT so that an event is generated when a DC Access Key violation occurs.
6. Post Send: Send work requests are posted to the DCIs. Data can be sent to a different DCT only after all previous sends complete, so send CQEs (Completion Queue Entries) can be used to detect such completions.
7. Detect Arrival: The CQ associated with the DCT is used to detect data arrival.
8. Destroy: Destroy resources when done.

9.6.1.2. Query Device

The following function is used to query for device capabilities:

```
int ibv_exp_query_device(struct ibv_context *context, struct ibv_exp_device_attr *attr)
```

- DC transport support: The flag **IBV_EXP_DEVICE_DC_TRANSPORT** in the field **exp_atomic_cap** of **struct ibv_exp_device_attr** defines if the DC transport is supported.
- Atomic support: The fields **int max_dc_req_rd_atom;** and **int max_dc_res_rd_atom;** in the same structure describe the DC's atomic support characteristics.

9.6.1.3. Create DCT

To create a DC target object, use:

```
struct ibv_dct *ibv_exp_create_dct(struct ibv_context *context, struct ibv_exp_dct_init_attr *attr);
```

Parameters:

- **context** : Context to the InfiniBand device as returned from **ibv_open_device** .
- **attr** : Defines attributes of the DCT, which include:
  - **struct ibv_pd *pd** : The PD to verify access validity with respect to protection domains.

    > ⚠ The PD is not checked against the PD of the scatter entry. This check is done with the PD of the DC target.

  - **struct ibv_cq *cq** : CQ used to report receive completions.
  - **struct ibv_srq *srq** : The SRQ that will provide the received buffers.
  - **dc_key** : A 64-bit key associated with the DCT.
  - **port** : The port number this DCT is bound to.
  - **access flags** : Semantics similar to RC QPs (e.g., remote read, remote write, remote atomics).
  - **min_rnr_timer** : Minimum RNR NAK time required from the requester between successive requests of a message that was previously rejected due to insufficient receive buffers (IB spec 9.7.5.2.8).
  - **tclass** : Used by packets sent by the DCT in case GRH is used.
  - **flow_label** : Used by packets sent by the DCT in case GRH is used.
  - **mtu** : Maximum Transmission Unit.
  - **pkey_index** : P_Key index used by the DC target.
  - **gid_index** : GID index associated with the DCT. Used to verify incoming packets if GRH is used.

    > ⚠ This field is mandatory.

  - **hop_limit** : Used by packets sent by the DCT in case GRH is used.

9.6.1.4. Destroy DCT

To destroy a DCT object, use:

```
int ibv_exp_destroy_dct(struct ibv_exp_dct *dct);
```

Destroys a DC target. This call may take some time until all DCRs (Dynamically Connected Responders) are disconnected.

9.6.1.5. Query DCT

To query DCT attributes, use:

```
int ibv_exp_query_dct(struct ibv_exp_dct *dct, struct ibv_exp_dct_attr *attr);
```

Attributes queried include:

- **state**
- **cq**
- **access_flags**
- **min_rnr_flags**
- **pd**
- **tclass**
- **flow_label**
- **dc_key**
- **mtu**
- **port**
- **pkey_index**
- **gid_index**
- **hop_limit**
- **key_violations**
- **srq**

9.6.1.6. Arm DCT

A DC target can be armed to request notification when DC key violations occur.

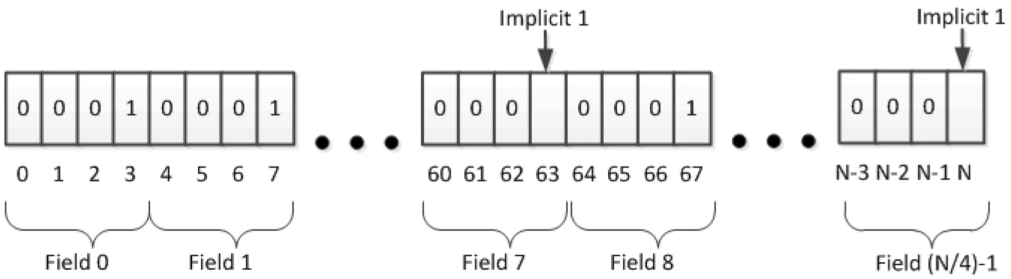
- After returning from a call to **ibv_exp_arm_dct**, the DC target is moved into the **ARMED** state.
- If a packet targeting this DCT with a wrong key is received, the DCT moves to the **FIRED** state, and the **IBV_EXP_EVENT_DCT_KEY_VIOLATION** event is generated.

9.6.1.7. Create DCI

DCIs (Dynamically Connected Initiators) are created by the user. The number of DCIs required depends heavily on the user's specific strategy for handling concurrent data transmissions.

9.7. Verbs API for Extended Atomics Support

The extended atomics capabilities provide support for performing Fetch&Add and masked Compare&Swap atomic operations on multiple fields. The figure below shows how the individual fields within the user-supplied-data field are specified.



In the figure above, the total operand size is N bits, with the length of each data field being four bits. The 1's in the mask indicate the termination of a data field. With ConnectX® family of NICs and Connect-IB®, there is always an implicit 1 in the mask.

9.7.1.1. Supported Hardware

NVIDIA NICs devices employ read-modify-write operations on regions that are sized as multiples of 64 bits with 64 bit alignment. Therefore, when operations are performed on user buffers that are smaller than 64 bits, the unmodified sections of such regions will be written back unmodified when the results are committed to user memory. Connect-IB® and subsequent devices operate on memory regions that are multiples of 32 or 64 bits, with natural alignment.

9.7.1.2. Verbs Interface Changes

Usage model:

- Query device to see if
  - Atomic Operations are supported
  - Endieness of atomic response
  - Extended atomics are supported, and the data sizes supported
- Initialize QP for use with atomic operations, taking device capabilities into account
- Use the atomic operations
- Destroy QP after finishing to use it

9.7.1.3. Query Device Capabilities

The device capabilities flags enumeration is updated to reflect the support for extended atomic operations by adding the flag **IBV_EXP_DEVICE_EXT_ATOMICS**.

The device attribute comp mask enumeration **ibv_exp_device_attr_comp_mask** is updated using the flag **IBV_EXP_DEVICE_ATTR_EXT_ATOMIC_ARGS**.

The device attributes struct, **ibv_exp_device_attr**, is modified by adding **struct ibv_exp_ext_atomics_params ext_atom**.

```
struct ibv_exp_ext_atomics_params {
 uint64_t atomic_arg_sizes; /* bit-mask of supported sizes */
 uint32_t max_fa_bit_boundary;
 uint32_t log_max_atomic_inline;
 };
```

Atomic fetch&add operations on subsections of the operands are also supported, with **max_fa_bit_boundary** being the log-base-2 of the largest such subfield, in bytes.

log_max_atomic_inline is the log of the largest amount of atomic data, in bytes, that can be put in the work request and includes the space for all required fields.

The returned data is formatted in units that correspond to the host's natural word size. For example, if extended atomics are used for a 16 byte field, and returned in big-endian format, each eight byte portion is arranged in big-endian format, regardless of the size the fields used in an association in a multi-field fetch-and-add operation.

9.7.1.4. Response Format

The returned data is formatted in units that correspond to the host's natural word size. For example, if extended atomics are used for a 16 byte field, and returned in big-endian format, each

eight byte portion is arranged in big-endian format, regardless of the size the fields used in an association in a multi-field fetch-and-add operation.

9.7.1.5. QP Initialization

QP initialization needs additional information with respect to the sizes of atomic operations that will be supported inline. This is needed to ensure the QP is provisioned with sufficient send resources to support the number of support WQE's.

The QP attribute enumeration comp-mask, **ibv_exp_qp_init_attr_comp_mask**, is expanded by adding **IBV_EXP_QP_INIT_ATTR_ATOMICS_ARG**.

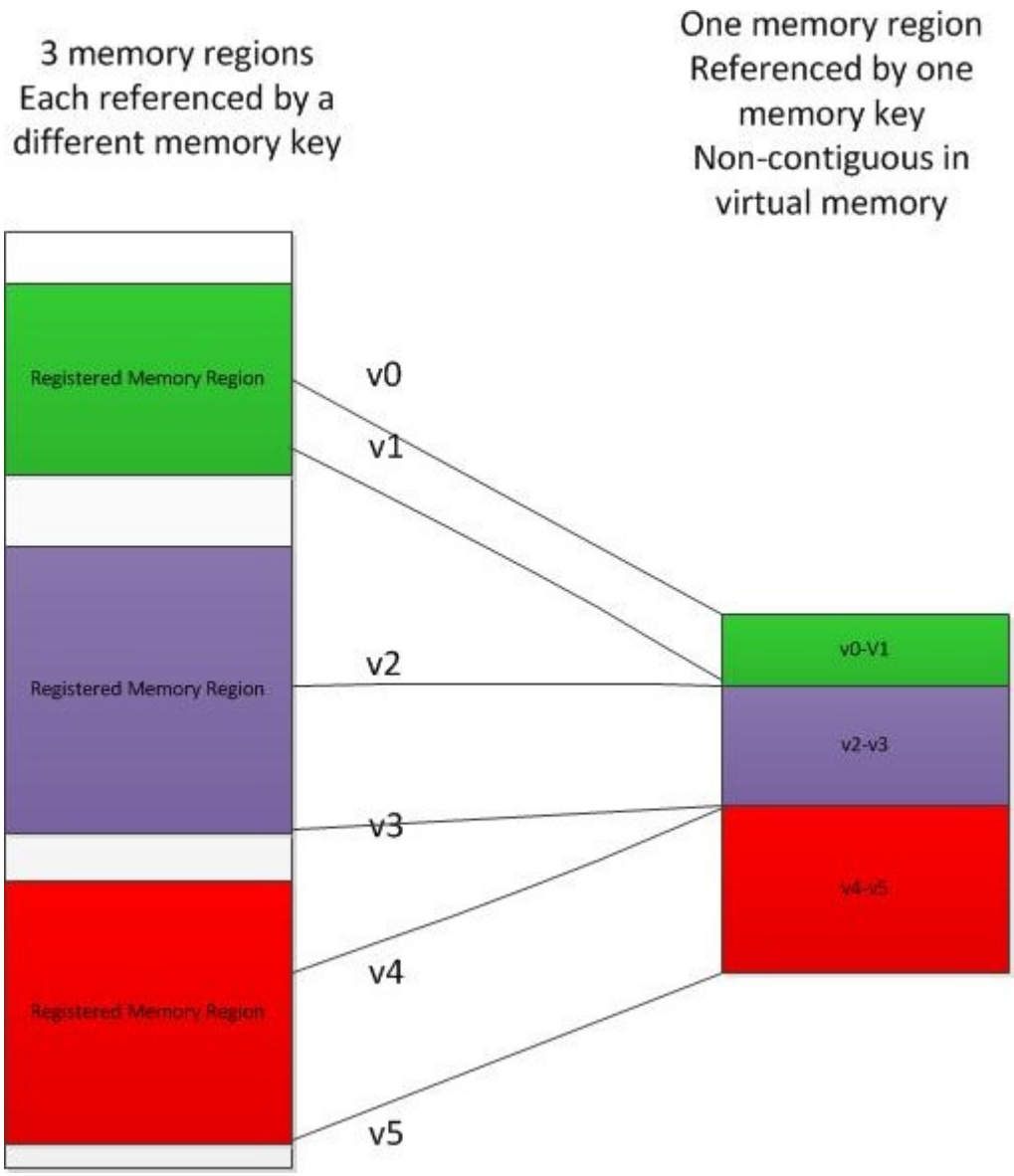
9.7.1.6. Send Work Request Changes

```
The send op codes are extended to include
+   IBV_EXP_WR_EXT_MASKED_ATOMIC_CMP_AND_SWP,
+   IBV_EXP_WR_EXT_MASKED_ATOMIC_FETCH_AND_ADD
ibv_exp_send_flags
The send flags, ibv_exp_send_flags, are expanded to include inline support for extended atomic operations with the
flag
+   IBV_EXP_SEND_EXT_ATOMIC_INLINE
The send work request is extended by appending
union {
    struct {
        /* Log base-2 of total operand size
         */
        uint32_t        log_arg_sz;
        uint64_t  remote_addr;
        uint32_t  rkey;  /* remote memory key */
        union {
            struct {
                /* For the next four fields:
                 * If operand_size < 8 bytes then inline data is in
                 * the corresponding field; for small operands,
                 * LSBs are used.
                 * Else the fields are pointers in the process's
                 * address space to
                 * where the arguments are stored
                 */
                union {
                    struct ibv_exp_cmp_swap cmp_swap;
                    struct ibv_exp_fetch_add fetch_add;
                } op;           } inline_data;
        /* in the future add support for non-inline
            * argument provisioning
            */
        } wr_data;
    } masked_atomics;
} ext_op;

To the end of work request, ibv_exp_send_wr,
with
struct ibv_exp_cmp_swap {
  uint64_t  compare_mask;
  uint64_t  compare_val;
  uint64_t  swap_val;
  uint64_t  swap----_mask;
};
and
struct ibv_exp_fetch_add {
  uint64_t  add_val;
  uint64_t  field_boundary;
};
```

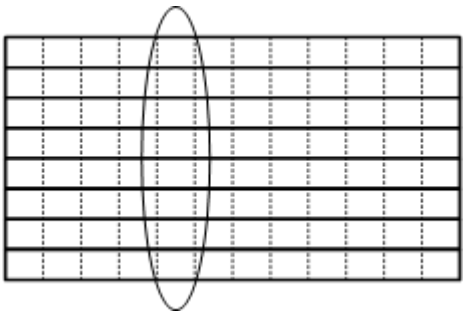
9.8. User-Mode Memory Registration

User-Mode Memory Registration (UMR) supports the creation of memory keys for non-contiguous memory regions. This includes the concatenation of arbitrary contiguous regions of memory, as well as regions with regular structure.

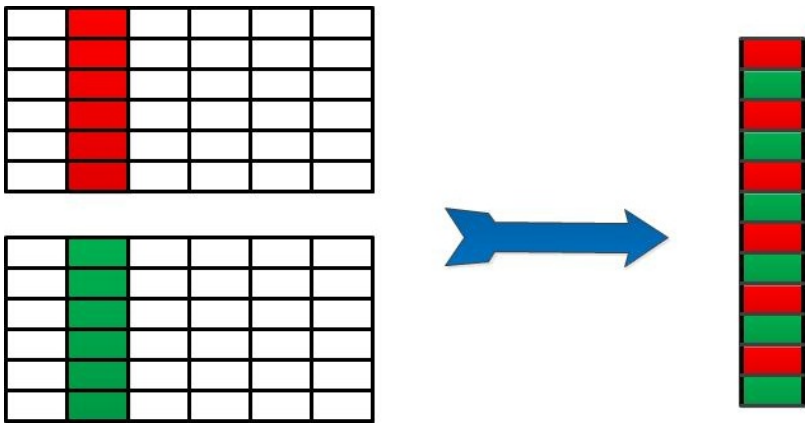
Three examples of non-contiguous regions of memory that are used to form new contiguous regions of memory are described below. This figure shows an example where portions of three separate contiguous regions of memory are combined to create a single logically contiguous region of memory. The base address of the new memory region is defined by the user when the new memory key is defined.



The following figure shows a non-contiguous memory region with regular. This region is defined by a base address, stride between adjacent elements, the extent of each element, and a repeat count.



The following figure shows an example where two non-contiguous memory regions are interleaved, using the repeat structure UMR.



9.8.1. Interfaces

The usage model for the UMR includes:

- Ability to with ibv_exp_query_device if UMR is supported.
- If UMR is supported, checking struct ibv_exp_device_attr for it's characteristics
- Using ibv_exp_create_mr() to create an uninitialized memory key for future UMR use
- Using ibv_exp_post_send() to define the new memory key. This can be posted to the same send queue that will use the memory key in future operations.
- Using the UMR defined as one would use any other memory keys
- Using ibv_exp_post_send() to invalidate the UMR memory key
- Releasing the memory key with the ibv_dereg_mr()

9.8.1.1. Device Capabilities

The query device capabilities is queried to see if the UMR capability is supported, and if so, what are it's characteristics. The routine used is:

```
int ibv_exp_query_device(struct ibv_context *context, struct ibv_exp_device_attr *attr)
```

The **struct ibv_exp_umr_caps umr_caps** field describes the UMR capabilities. This structure is defined as:

```
struct ibv_exp_umr_caps {
 uint32_t max_klm_list_size;
 uint32_t max_send_wqe_inline_klms;
 uint32_t max_umr_recursion_depth;
 uint32_t max_umr_stride_dimension;
};
```

The fields added to the struct struct ibv_exp_device_attr to support UMR include:

- **exp_device_cap_flags** – UMR support available if the flag IBV_EXP_DEVICE_ATTR_UMR is set.
- **max_mkey_klm_list_size** – maximum number of memory keys that may be input to UMR
- **max_send_wqe_inline_klms** – the largest number of KLM's that can be provided inline in the work request. When the list is larger than this, a buffer allocated via the **struct ibv_mr**

***ibv_exp_reg_mr(struct ibv_exp_reg_mr_in *in)** function, and provided to the driver as part of the memory key creation

- **max_umr_recursion_depth** – memory keys created by UMR operations may be input to UMR memory key creation. This specifies the limit on how deep this recursion can be.
- **max_umr_stride_dimension** – The maximum number of independent dimensions that may be used with the regular structure UMR operations. The current limit is one.

9.8.1.2. QP Creation

To configure QP UMR support the following routine must be used:

```
ibv_qp * ibv_exp_create_qp(struct ibv_context *context, struct ibv_exp_qp_init_attr *qp_init_attr)
```

When the attribute **IBV_EXP_QP_CREATE_UMR** is set in the exp_create_flags field of struct ibv_exp_qp_init_attr enables UMR support. The attribute **IBV_ IBV_EXP_QP_INIT_ATTR_MAX_INL_KLMS** is set in the field **comp_mask struct ibv_exp_qp_init_attr**, with the field **max_inl_send_klms** defining this number.

9.8.1.3. Memory Key Manipulation

To create an uninitialized memory key for future use:

```
struct ibv_mr *ibv_exp_create_mr(struct ibv_exp_create_mr_in *create_mr_in)
is used with
struct ibv_exp_create_mr_in {
    struct ibv_pd *pd;
    struct ibv_exp_mr_init_attr attr;
};
and
struct ibv_exp_mr_init_attr {
    uint64_t max_reg_descriptors; /* maximum number of entries */
    uint32_t create_flags; /* enum ibv_mr_create_flags */
    uint64_t access_flags; /* region's access rights */
    uint32_t comp_mask;
};
```

To query the resources associated with the memory key:

```
int ibv_exp_query_mkey(struct ibv_mr *mr, struct ibv_exp_mkey_attr *query_mkey_in)
is used with
struct ibv_exp_mkey_attr {
    int n_mkey_entries;  /* the maximum number of memory keys that can be supported */
     uint32_t comp_mask;
};
```

9.8.1.4. Non-inline Memory Objects

When the list of memory keys input into the UMR memory key creation is too large to fit into the work request, a hardware accessible buffer needs to be provided in the posted send request. This buffer will be populated by the driver with the relevant memory objects.

```
We will define the enum
enum memory_reg_type{
    IBV_MEM_REG_MKEY
};

The memory registration function is defined as:

struct non_inline_data  *ibv_exp_alloc_mkey_list_memory
            (struct ibv_exp_mkey_list_container_attr *attr)
where
struct ibv_exp_mkey_list_container_attr {
  struct ibv_pd *pd;
  uint32_t mkey_list_type;  /* use ibv_exp_mkey_list_type */
  uint32_t max_klm_list_size;
  uint32_t comp_mask; /*use ibv_exp_alloc_mkey_list_comp_mask */
```

```
};
This memory is freed with
int ibv_exp_dealloc_mkey_list_memory(struct ibv_exp_mkey_list_container *mem)

where
struct ibv_exp_mkey_list_container {
  uint32_t max_klm_list_size;
  struct ibv_context *context;
};  (NOTE - Need to check with Eli Cohen here - just reading the code).
```

9.8.1.5. Memory Key Initialization

The memory key is manipulated with the **ibv_exp_post_send()** routine. The opcodes **IBV_EXP_WR_UMR_FILL** and **IBV_EXP_WR_UMR_INVALIDATE** are used to define and invalidate, respectively, the memory key.

The **struct ibv_exp_send_wr** contains the following fields to support the UMR capabilities:

```
union {
    struct {
      uint32_t umr_type; /* use ibv_exp_umr_wr_type */
      struct ibv_exp_mkey_list_container *memory_objects; /* used when IBV_EXP_SEND_INLINE is not set */
      uint64_t exp_access; /* use ibv_exp_access_flags */
      struct ibv_mr *modified_mr;
      uint64_t base_addr;
      uint32_t num_mrs; /* array size of mem_repeat_block_list or mem_reg_list */
      union {
        struct ibv_exp_mem_region *mem_reg_list; /* array, size corresponds to num_mrs */
        struct {
          struct ibv_exp_mem_repeat_block *mem_repeat_block_list; /* array,  size corresponds to num_mr */
          size_t *repeat_count; /* array size corresponds to stride_dim */
          uint32_t stride_dim;
        } rb;
      } mem_list;
    } umr;

where
enum ibv_exp_umr_wr_type {
  IBV_EXP_UMR_MR_LIST,
  IBV_EXP_UMR_REPEAT
};

and

struct ibv_exp_mkey_list_container {
  uint32_t max_klm_list_size;
  struct ibv_context *context;
};

struct ibv_exp_mem_region {
  uint64_t base_addr;
  struct ibv_mr *mr;
  size_t length;
};

and

struct ibv_exp_mem_repeat_block {
  uint64_t base_addr; /* array, size corresponds to ndim */
  struct ibv_mr *mr;
  size_t *byte_count; /* array, size corresponds to ndim */
  size_t *stride; /* array, size corresponds to ndim */
};
```

9.9. Cross-Channel Communications Support

The Cross-Channel Communications adds support for work requests that are used for synchronizing communication between separate QP's and support for data reductions. This functionality, for example, is sufficient for implementing MPI collective communication with a single post of work requests, with the need to check only of full communication completion, rather than on completion of individual work requests.

Terms relevant to the Cross-Channel Synchronization are defined in the following table:

Term	Description
Cross Channel supported QP	QP that allows send_enable, recv_enable, wait, and reduction tasks.

Term	Description
Managed send QP	Work requests in the corresponding send queues must be explicitly enabled before they can be executed.
Managed receive QP	Work requests in the corresponding receive queues must be explicitly enabled before they can be executed.
Master Queue	Queue that uses send_enable and/or recv_enable work requests to enable tasks in managed QP. A QP can be both master and managed QP.
Wait task (n)	Task the completes when n completion tasks appear in the specified completion queue
Send Enable task (n)	Enables the next n send tasks in the specified send queue to be executable.
Receive Enable task	Enables the next n send tasks in the specified receive queue to be executable.
Reduction operation	Data reduction operation to be executed by the HCA on specified data.

9.9.1. Usage Model

- Creating completion queues, setting the ignore-overrun bit for the CQ's that only hardware will monitor.
- Creating and configuring the relevant QP's, setting the flags indicating that Cross-Channel Synchronization work requests are supported, and the appropriate master and managed flags (based on planned QP usage). For example, this may happen when an MPI library creates a new communicator.
- Posting tasks list for the compound operations.
- Checking the appropriate queue for compound operation completion (need to request completion notification from the appropriate work request). For example, a user may setup a CQ that receives completion notification for the work-request whose completion indicates the entire collective operation has completed locally.
- Destroying the QP's and CQ's created for Cross-Channel Synchronization operations, once the application is done using them. For example, an MPI library may destroy these resources after it frees all the communicator using these resources.

9.9.2. Resource Initialization

9.9.2.1. Device Capabilities

```
The device query function,
int ibv_exp_query_device(struct ibv_context *context,
              struct ibv_exp_device_attr *attr);
is used to query for device capabilities.
A value of
IBV_EXP_DEVICE_CROSS_CHANNEL
in exp_device_cap_flags indicates support for Cross-Channel capabilities.

In addition, the struct calc_cap is used to define what reduction capabilities are supported
struct ibv_exp_device_attr {
 …
    struct ibv_exp_device_calc_cap calc_cap;
 …
};

where,
struct ibv_exp_device_calc_cap {
  uint64_t    data_types;
  uint64_t    data_sizes;
  uint64_t    int_ops;
  uint64_t    uint_ops;
  uint64_t    fp_ops;
};
Where the operation types are given by:
IBV_EXP_CALC_OP_ADD , /* addition */
IBV_EXP_CALC_OP_BAND, /* bit-wise and */
IBV_EXP_CALC_OP_BXOR, /*bit wise xor */
IBV_EXP_CALC_OP_BOR, /* bit-wise or */
```

```
and data types supported are described by
IBV_EXP_CALC_DATA_SIZE_64_BIT
```

9.9.2.2. Completion Queue

Completion queue (CQ) that will be used with Cross Channel Synchronization operations needs to be marked as such as CQ at creation time. This CQ needs to be initialized with:

```
struct ibv_cq *ibv_exp_create_cq(struct ibv_context *context,
                 int cqe,
                 void *cq_context,
                 struct ibv_comp_channel *channel,
                 int comp_vector,
                 struct ibv_exp_cq_init_attr *attr)
where the new parameter is defined as:
struct ibv_exp_cq_init_attr{
    uint32_t comp_mask;
    unit32_t flags;
}
The appropriate flag to set is:
IBV_EXP_CQ_CREATE_CROSS_CHANNEL
The comp_mask needs to set the bit,
IBV_EXP_CQ_INIT_ATTR_FLAGS
To avoid the CQ's entering the error state due to lack of CQ processing, the overrun ignore (OI) bit of the
Completion Queue Context table must be set.
To set these bit the function
/**
 * ibv_exp_modify_cq - Modifies the attributes for the specified CQ.
 * @cq: The CQ to modify.
 * @cq_attr: Specifies the CQ attributes to modify.
 * @cq_attr_mask: A bit-mask used to specify which attributes of the CQ
 *   are being modified.
 */
static inline int ibv_exp_modify_cq(struct ibv_cq *cq,
            struct ibv_exp_cq_attr *cq_attr,
            int cq_attr_mask)
The bit IBV_EXP_CQ_CAP_FLAGS in  cq_attr_mask needs to be set, as does the bit IBV_EXP_CQ_ATTR_CQ_CAP_FLAGS  in
cq_attr_mask's comp_mask.  Finally, the bit IBV_EXP_CQ_IGNORE_OVERRUN needs to be set in the field cq_cap_flags.
```

9.9.2.3. QP Creation

To configure the QP for Cross-Channel use following function is used:

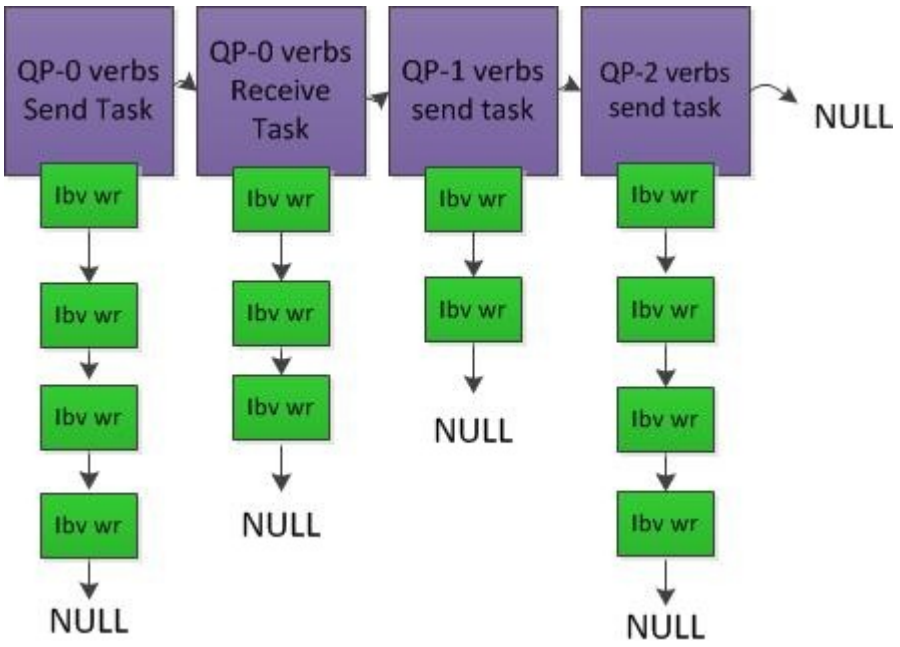
```
struct ibv_qp *ibv_exp_create_qp(struct ibv_context *context,
    struct ibv_exp_qp_init_attr *qp_init_attr)

where

struct ibv_exp_qp_init_attr {
  void           *qp_context;
  struct ibv_cq        *send_cq;
  struct ibv_cq        *recv_cq;
  struct ibv_srq         *srq;
  struct ibv_qp_cap cap;
  enum ibv_qp_type  qp_type;
  int     sq_sig_all;

  uint32_t    comp_mask; /* use ibv_exp_qp_init_attr_comp_mask */
  struct ibv_pd        *pd;
  struct ibv_xrcd        *xrcd;
  uint32_t    exp_create_flags; /* use ibv_exp_qp_create_flags */

  uint32_t    max_inl_recv;
  struct ibv_exp_qpg  qpg;
  uint32_t    max_atomic_arg;
  uint32_t                max_inl_send_klms;
};
```

The **exp_create_flags** that are available:

- **IBV_EXP_QP_CREATE_CROSS_CHANNEL** – This must be set for any QP to which cross-channel-synchronization work requests will be posted.
- **IBV_EXP_QP_CREATE_MANAGED_SEND** – This is set for a managed send QP, e.g. one for which send-enable operations are used to activate the posted send requests.
- **IBV_EXP_QP_CREATE_MANAGED_RECV** – This is set for a managed receive QP, e.g. one for which send-enable operations are used to activate the posted receive requests.

9.9.3. Posting Request List

A single operation is defined with by a set of work requests posted to multiple QP's, as described in the figure below.



The lists are of tasks are NULL terminated.

The routine:

```
int ibv_exp_post_task(struct ibv_context *context, struct ibv_exp_task *task, struct ibv_exp_task **bad_task)
is used to post the list of work requests, with

struct ibv_exp_task {
  enum ibv_exp_task_type  task_type;
  struct {
    struct ibv_qp  *qp;
    union {
      struct ibv_exp_send_wr  *send_wr;
      struct ibv_recv_wr  *recv_wr;
    };
  } item;
  struct ibv_exp_task    *next;
  uint32_t                comp_mask; /* reserved for future growth (must be 0) */
};

The task type is defined by:

IBV_EXP_TASK_SEND
  IBV_EXP_TASK_RECV

To support the new work requests, the struct ibv_exp_send_wr is expanded with
union {
    struct {
      uint64_t    remote_addr;
      uint32_t    rkey;
    } rdma;
    struct {
      uint64_t    remote_addr;
      uint64_t    compare_add;
      uint64_t    swap;
      uint32_t    rkey;
    } atomic;
    struct {
      struct ibv_cq *cq;
      int32_t  cq_count;
    } cqe_wait;
    struct {
      struct ibv_qp *qp;
      int32_t  wqe_count;
    } wqe_enable;
  } task;

The calc operation is also defined in ibv_exp_send_wr by the union:

union {
    struct {
      enum ibv_exp_calc_op        calc_op;
      enum ibv_exp_calc_data_type data_type;
```

```
        enum ibv_exp_calc_data_size data_size;
    } calc;
} op;
```

In addition, in the field **exp_send_flags** in **ibv_exp_send_wr** the flag **IBV_EXP_SEND_WITH_CALC** indicates the presence of a reduction operation, and **IBV_EXP_SEND_WAIT_EN_LAST** is used to signal the last wait task posted for a given CQ in the current task list.

For **ibv_exp_calc_data_type** the following types are supported:

- **IBV_EXP_CALC_DATA_TYPE_INT**
- **IBV_EXP_CALC_DATA_TYPE_UINT**
- **IBV_EXP_CALC_DATA_TYPE_FLOA**

The supported data size for **ibv_exp_data_size** is **IBV_EXP_CALC_DATA_SIZE_64_BIT**.

New send opcodes are defined for the new work requests. These include:

- **IBV_EXP_WR_SEND_ENABLE**
- **IBV_EXP_WR_RECV_ENABLE**
- **IBV_EXP_WR_CQE_WAIT**

9.9.3.1. ConnectX-3/Connect-IB Data Endianess

The ConnectX-3 and Connect-IB HCA's expect to get the data in network order.

5.6.2.3.2 RDMA over Converged Ethernet

Remote Direct Memory Access (RDMA) enables direct memory-to-memory data transfers between servers, bypassing CPU involvement. RoCE (RDMA over Converged Ethernet) extends RDMA functionality to lossless Ethernet networks, delivering high-throughput, ultra-low latency communication.

With advancements in reliable Ethernet technologies, the NVIDIA® ConnectX® Ethernet adapter family supports RDMA transport over 10GigE and 40GigE networks. By offloading RDMA services to hardware, these adapters significantly reduce CPU load and latency—making them ideal for performance-critical applications like financial systems, databases, storage, and content delivery networks.

5.6.2.3.2.1 Key Considerations for RoCE

When deploying RDMA applications over Ethernet, consider the following:

- No Subnet Manager (SM) – Unlike InfiniBand, RoCE does not require an SM. However, certain operations (e.g., multicast group joins) must be handled differently, even though the API remains unchanged.
- No LID (Local ID) – As LID is specific to InfiniBand, it is not set for RoCE ports and appears as zero in queries.
- No APM (Alternate Path Migration) – RoCE does not support APM, as it relies on InfiniBand-specific mechanisms.
- Manual Path Record Setup – Since path querying is not possible without an SM, users must manually populate the path record structure before establishing a connection. Using RDMA-CM is strongly recommended, as it automates path record handling.

- Priority Mapping via VLAN Tags – VLAN-tagged Ethernet frames use a 3-bit priority field derived from the 3 least significant bits of the InfiniBand Service Level (SL) field.
- RoCE Traffic Counters – RoCE traffic is offloaded by hardware and is not visible in standard Ethernet interface counters. Instead, RoCE traffic statistics can be found at:

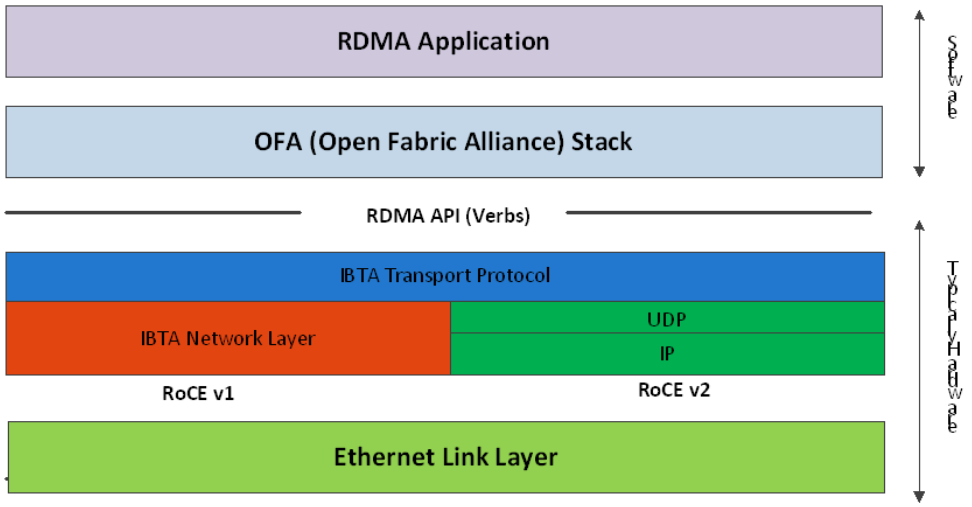
```
/sys/class/infiniband/<device>/ports/<port number>/counters/
```

5.6.2.3.2.2 RoCE Modes

RoCE encapsulates IB transport in one of the following Ethernet packets:

- RoCEv1 - uses a dedicated ethertype (**0x8915**)
- RoCEv2 - encapsulates RDMA in UDP/IP packets, using UDP port **4791**

RoCEv1 and RoCEv2 Protocol Stack



RoCEv1

RoCEv1 uses a dedicated Ethernet EtherType (**0x8915**). It may include or omit VLAN tags and adheres to the standard Ethernet MTU. It is a Layer 2 protocol and is limited to communication within the same Ethernet broadcast domain.

RoCEv2

RoCEv2 extends RDMA to Layer 3 networks by encapsulating RDMA traffic in UDP/IP packets. This allows traffic to traverse IP routers and use standard IP routing. Key features include:

- IP Header – Enables routing across IP networks.
- UDP Header – Used for stateless encapsulation; destination port **4791** .
- Source Port Field – Can serve as an opaque flow ID to assist with ECMP (Equal-Cost Multi-Path) routing.

Since the packet format change occurs at the wire level and does not affect the RDMA API, applications can operate seamlessly over RoCEv1 or RoCEv2 without modification.

⚠ Both RoCEv1 and RoCEv2 are supported by default. The driver associates GID indexes with both modes, creating one entry per version.

For deployment examples, see the Recommended Network Configuration Examples For RoCE Deployment community post.

5.6.2.3.2.3 GID Table Population

GID (Global Identifier) table entries are automatically created when an IP address is configured on an Ethernet device associated with a RoCE port. Each GID entry includes:

- GID value
- GID type (e.g., RoCEv1, RoCEv2)
- Associated network device

By default, each RoCE-capable port has two GID entries with the same value but different types (RoCEv1 and RoCEv2). GID format can be:

- IPv4 GID – An IPv4-mapped IPv6 address
- IPv6 GID – A native IPv6 address

The Layer 3 header depends on the GID type:

- For RoCEv2 – IPv4 or IPv6
- For RoCEv1 – Uses IPv4 GIDs as-is, with GRH if required

Accessing GID Table via Sysfs

The GID (Global Identifier) table is exposed to userspace via **sysfs** and provides details for each GID entry associated with a RoCE-capable port:

- GID value:

```
/sys/class/infiniband/{device}/ports/{port}/gids/{index}
```

- GID type (e.g., RoCEv1 or RoCEv2):

```
/sys/class/infiniband/{device}/ports/{port}/gid_attrs/types/{index}
```

- Associated network device:

```
/sys/class/infiniband/{device}/ports/{port}/gid_attrs/ndevs/{index}
```

Each file in this directory represents a GID index and contains the corresponding GID value.

Setting the RoCE Mode for a QP

For devices supporting both RoCEv1 and RoCEv2, selecting the desired mode depends on the Queue Pair (QP) type:

RC/UC QPs (Connected QPs)

To transition a connected QP from INIT to RTR state, you must provide an Address Vector (AV). Among its attributes, the AV must include the GID index from the port's GID table to define the source GID for the QP.

The GID type at the specified index will determine the RoCE mode (v1 or v2) for the QP.

UD QPs (Unconnected QPs)

UD QPs use GIDs for addressing, but since they are connectionless, setting the GID index in the AV similarly determines the RoCE version used for sending packets.

Setting RoCE Mode of RDMA_CM Applications

The **RDMA_CM** interface simplifies connection management by requiring only the active side to specify the IP address of the passive peer.

- The **RDMA_CM** stack selects a source GID from the local GID table.
- Since multiple entries may have the same GID value (but different types), the GID type must be specified for accurate lookup.

Configuring RDMA_CM GID Type

RDMA_CM uses a global module setting to determine which GID type to use when resolving the local GID. This can be controlled using the **cma_roce_mode** utility:

- To print current RoCE mode for a port:

```
cma_roce_mode -d <device> -p <port>
```

- To set RoCE mode (1 for RoCEv1, 2 for RoCEv2):

```
cma_roce_mode -d <device> -p <port> -m <1|2>
```

This affects **RDMA_CM** applications by defining how the source GID is selected during connection establishment.

GID Table Example

The following is an example of the GID table.

DEV	PORT	INDEX	GID	IPv4	Type	Netdev
mlx5_0	1	0	fe80:0000:0000:0000:ba59:9fff:fe1a:e3ea		v1	p4p1
mlx5_0	1	1	fe80:0000:0000:0000:ba59:9fff:fe1a:e3ea		v2	p4p1
mlx5_0	1	2	0000:0000:0000:0000:0000:ffff:0a0a:0a01	10.10.10.1	v1	p4p1
mlx5_0	1	3	0000:0000:0000:0000:0000:ffff:0a0a:0a01	10.10.10.1	v2	p4p1
mlx5_1	1	0	fe80:0000:0000:0000:ba59:9fff:fe1a:e3eb		v1	p4p2
mlx5_1	1	1	fe80:0000:0000:0000:ba59:9fff:fe1a:e3eb		v2	p4p2

Where:

- Entries on port 1 index 0/1 are the default GIDs, one for each supported RoCE type
- Entries on port 1 index 2/3 belong to IP address 192.168.1.70 on eth1
- Entries on port 1 index 4/5 belong to IP address 193.168.1.70 on eth1.100
- Packets from a QP that is associated with these GID indexes will have a VLAN header (VID=100)
- Entries on port 1 index 6/7 are IPv6 GID. Packets from a QP that is associated with these GID indexes will have an IPv6 header

5.6.2.3.2.4 RoCE Lossless Ethernet Configuration

To function reliably, RoCE requires a form of flow control. While it is possible to use global flow control, this is normally undesirable, for performance reasons.

The normal and optimal way to use RoCE is to use Priority Flow Control (PFC). To use PFC, it must be enabled on all endpoints and switches in the flow path.

5.6.2.3.2.5 Installing and Loading the Driver

To install and load the driver:

1. Install MLNX_OFED (See Installation section for further details).
   RoCE is installed as part of mlx5 and other modules upon driver's installation.

   > ⚠ The list of the modules that will be loaded automatically upon boot can be found in the configuration file **/etc/infiniband/openib.conf** .

2. Query for the device's information. Example:

```
ibv_devinfo MLNX_OFED_LINUX-5.0-2.1.8.0:
```

3. Display the existing MLNX_OFED version.

```
ofed_info -s
hca_id: mlx5_0
        transport:                      InfiniBand (0)
        fw_ver:                         16.28.0578
        node_guid:                      ec0d:9a03:0044:3764
        sys_image_guid:                 ec0d:9a03:0044:3764
        vendor_id:                      0x02c9
        vendor_part_id:                 4121
        hw_ver:                         0x0
        board_id:                       MT_0000000009
        phys_port_cnt:                  1
                port:   1
                        state:                  PORT_ACTIVE (4)
                        max_mtu:                4096 (5)
                        active_mtu:             1024 (3)
                        sm_lid:                 0
                        port_lid:               0
                        port_lmc:               0x00
                        link_layer:             Ethernet
```

Output notes:

The port's state is: Ethernet is in PORT_ACTIVE state	The port state can also be obtained by running the following command: **# cat /sys/class/infiniband/mlx5_0/ports/1/state: ACTIVE**

link_layer parameter shows that port 1 is Ethernet	The link_layer can also be obtained by running the following command: **# cat /sys/class/infiniband/mlx5_0/ports/1/link_layer** **Ethernet**
The fw_ver parameter shows that the firmware version is 16.28.0578.	The firmware version can also be obtained by running the following command: **# cat /sys/class/infiniband/mlx5_0/fw_ver 16.28.0578**

Associating InfiniBand Ports to Ethernet Ports

The mlx5_ib driver holds a reference to the net device for getting notifications about the state of the port, as well as using the mlx5_core driver to resolve IP addresses to MAC that are required for address vector creation. However, RoCE traffic does not go through the mlx5_core driver; it is completely offloaded by the hardware.

```
# ibdev2netdev
mlx5_0 port 1 <===> eth2
#
```

Configuring an IP Address to the netdev Interface

To configure an IP address to the netdev interface:

1. Configure an IP address to the netdev interface on both sides of the link.

```
# ifconfig eth2 20.4.3.220
# ifconfig eth2
eth2    Link encap:Ethernet HWaddr 00:02:C9:08:E8:11
    inet addr:20.4.3.220 Bcast:20.255.255.255 Mask:255.0.0.0
    UP BROADCAST MULTICAST MTU:1500 Metric:1
    RX packets:0 errors:0 dropped:0 overruns:0 frame:0
    TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
    collisions:0 txqueuelen:1000
    RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)
```

2. Make sure that ping is working.

```
ping 20.4.3.219
PING 20.4.3.219 (20.4.3.219) 56(84) bytes of data.
64 bytes from 20.4.3.219: icmp_seq=1 ttl=64 time=0.873 ms
64 bytes from 20.4.3.219: icmp_seq=2 ttl=64 time=0.198 ms
64 bytes from 20.4.3.219: icmp_seq=3 ttl=64 time=0.167 ms
20.4.3.219 ping statistics —
3 packets transmitted, 3 received, 0% packet loss, time 2000ms rtt min/avg/max/mdev = 0.167/0.412/0.873/
0.326 ms
```

Adding VLANs

To add VLANs:

1. Make sure that the 8021.q module is loaded.

```
modprobe 8021q
```

2. Add VLAN.

```
# vconfig add eth2 7
Added VLAN with VID == 7 to IF -:eth2:-
#
```

3. Configure an IP address.

```
ifconfig eth2.7 7.4.3.220
```

Defining Ethernet Priority (PCP in 802.1q Headers)

1. Define Ethernet priority on the server.

```
# ibv_rc_pingpong -g 1 -i 2 -l 4
local address: LID 0x0000, QPN 0x1c004f, PSN 0x9daf6c, GID fe80::202:c900:708:e799
remote address: LID 0x0000, QPN 0x1c004f, PSN 0xb0a49b, GID fe80::202:c900:708:e811
8192000 bytes in 0.01 seconds = 4840.89 Mbit/sec
1000 iters in 0.01 seconds = 13.54 usec/iter
```

2. Define Ethernet priority on the client.

```
# ibv_rc_pingpong -g 1 -i 2 -l 4 sw419
local address: LID 0x0000, QPN 0x1c004f, PSN 0xb0a49b, GID fe80::202:c900:708:e811
remote address: LID 0x0000, QPN 0x1c004f, PSN 0x9daf6c, GID fe80::202:c900:708:e799
8192000 bytes in 0.01 seconds = 4855.96 Mbit/sec
1000 iters in 0.01 seconds = 13.50 usec/iter
```

Using rdma_cm Tests

1. Use rdma_cm test on the server.

```
# ucmatose
cmatose: starting server
initiating data transfers
completing sends
receiving data transfers
data transfers complete
cmatose: disconnecting
disconnected
test complete
return status 0
#
```

2. Use rdma_cm test on the client.

```
# ucmatose -s 20.4.3.219
cmatose: starting client
cmatose: connecting
receiving data transfers
sending replies
data transfers complete
test complete
return status 0
#
```

This server-client run is without PCP or VLAN because the IP address used does not belong to a VLAN interface. If you specify a VLAN IP address, then the traffic should go over VLAN.

5.6.2.3.2.6 Type Of Service (ToS)

The TOS field for **rdma_cm** sockets can be set using the **rdma_set_option()** API, just as it is set for regular sockets. If a TOS is not set, the default value (0) is used.

ToS Kernel Implementation

Modern Linux kernels (v2.6.39+) no longer use hard-coded bit shifting to determine service levels. Instead, the kernel uses a two-step mapping process:

1. TOS to Linux Priority: The TOS byte is interpreted using RFC 1349 service bits to derive a Linux Socket Priority (**sk_prio**).

2. Linux Priority to Hardware Resources: The Linux Priority is then mapped to the InfiniBand SL and VLAN Priority Code Point (PCP).

Step 1: TOS to Linux Priority Mapping

The bit pattern extracted from the TOS value is mapped to a Linux priority as shown below:

TOS Value	Linux Priority	Traffic Class
0x0–0x6	0	Best Effort
0x8–0xE	2	Bulk Data
0x10–0x16	6	Interactive
0x18–0x1E	4	Interactive Bulk

Step 2: Priority to SL and PCP

Once the Linux Priority is determined, the hardware driver maps it to specific network headers based on the interface configuration:

- Service level (SL): The Linux Priority is mapped to an SL based on the **tc mqprio** configuration. If **mqprio** is not configured, the SL is derived from the VLAN device. If neither mapping is present, the SL defaults to 0. Note that this is independent of the hardware's priority-to-TC mapping.
- VLAN PCP: For traffic on a tagged VLAN interface, the PCP is determined by the **tc mqprio** configuration on the upper network device. If **mqprio** is not configured, the PCP is derived from the VLAN interface's egress map (configured via **ip link** or **vconfig**).

DSCP

A configuration entry exists in the RDMA-CM **configfs** that allows users to select a default TOS for RDMA-CM QPs. This is useful for users who want to control the TOS field without modifying their source code.

Applications that set the TOS explicitly using the **rdma_set_option** API will override the **configfs** value and continue to work as expected. For further information about DSCP marking, refer to the "HowTo Set Egress ToS/DSCP on RDMA CM QPs" community post.

5.6.2.3.2.7 RoCE LAG

RoCE LAG is a feature meant for mimicking Ethernet bonding for IB devices and is available for dual port cards only.
This feature is supported on kernel versions 4.9 and above.

RoCE LAG mode is entered when both Ethernet interfaces are configured as a bond in one of the following modes:

- active-backup (mode 1)
- balance-xor (mode 2)
- 802.3ad (LACP) (mode 4)

Any change of bonding configuration that negates one of the above rules (i.e, bonding mode is not 1, 2 or 4, or both Ethernet interfaces that belong to the same card are not the only slaves

of the bond interface), will result in exiting RoCE LAG mode and the return to normal IB device per port configuration.
Once RoCE LAG is enabled, instead of having two IB devices; mlx5_0 and mlx5_1, there will be one device named mlx5_bond_0.
For information on how to configure RoCE LAG, refer to HowTo Configure RoCE over LAG (ConnectX-4/ConnectX-5/ConnectX-6) Community post.

5.6.2.3.2.8 Disabling RoCE

By default, RoCE is enabled on all mlx5 devices. When RoCE is enabled, all traffic to UDP port 4791 is treated as RoCE traffic by the device.

In case you are only interested in Ethernet (no RDMA) and wish to enable forwarding of traffic to this port, you can disable RoCE through sysfs:

```
echo <0|1> > /sys/devices/{pci-bus-address}/roce_enable
```

Once RoCE is disabled, only Ethernet traffic will be supported. Therefore, there will be no GID tables and only Raw Ethernet QPs will be supported.

The current RoCE state can be queried by sysfs:

```
cat /sys/devices/{pci-bus-address}/roce_enable
```

5.6.2.3.2.9 Enabling/Disabling RoCE on VMs via VFs

By default, when configuring VFs on the hypervisor, all VFs will be enabled with RoCE. This means they require more OS memory (from the VM). In case you are only interested in Ethernet (no RDMA) on the VM, and you wish to save the VM memory, you can disable RoCE on the VF from the hypervisor. In addition, by disabling RoCE, a VM can have the capability of utilizing the RoCE UDP port (4791) for standard UDP traffic.

For details on how to enable/disable RoCE on a VF, refer to HowTo Enable/Disable RoCE on VMs via VFs Community post.

5.6.2.3.2.10 Force DSCP

This feature enables setting a global traffic_class value for all RC QPs, or setting a specific traffic class based on several matching criteria.

Usage

- To set a single global traffic class to be applied to all QPs, write the desired global traffic_class value to /sys/class/infiniband/<dev>/tc/<port>/traffic_class.
  Note the following:
  - Negative values indicate that the feature is disabled. traffic_class value can be set using ibv_modify_qp()
  - Valid values range between 0 - 255

The ToS field is 8 bits, while the DSCP field is 6 bits. To set a DSCP value of X, you need to multiply this value by 4 (SHIFT 2). For example, to set DSCP value of 24, set the ToS bit to 96 (24x4=96).

- To set multiple traffic class values based on source and/or destination IPs, write the desired rule to /sys/class/infiniband/<dev>/tc/<port>/traffic_class. For example:

```
echo "tclass=16,src_ip=1.1.1.2,dst_ip=1.1.1.0/24" > /sys/class/infiniband/mlx5_0/tc/1/traffic_class
```

Note: Adding "tclass" prefix to tclass value is optional.

In the example above, traffic class 16 will be set to any QP with source IP 1.1.1.2 and destination IP 1.1.1.0/24.
Note that when setting a specific traffic class, the following rule precedence will apply:

- If a global traffic class value is set, it will be applied to all QPs
- If no global traffic class value is set, and there is a rule with matching source and destination IPs applicable to at least one QP, it will be applied
- Rules only with matching source and/or destination IPs have no defined precedence over other rules with matching source and/or destination IPs

Notes:

- A mask can be provided when using destination IPv4 addresses
- The rule precedence is not affected by the order in which rules are inserted
- Overlapping rules are entirely up to the administrator.
- "tclass=-1" will remove the rule from the database

5.6.2.3.2.11 Force Time to Live (TTL)

This feature enables setting a global TTL value for all RC QPs.
Write the desired TTL value to /sys/class/infiniband/<dev>/tc/<port>/ttl. Valid values range between 0 - 255

5.6.2.3.3 Adaptive Retransmission: Parameters Control

5.6.2.3.3.1 1. Introduction

Adaptive Retransmission (ADP) Algorithm timeout adjustments is introduced in the April 2025 GA firmware for all NICs supporting the standard ADP algorithm. It addresses advanced customer needs for greater flexibility, enabling multiple timeout values and configurable retry counts per timeout. This enhancement allows for faster reactions to dropped packet reports while giving the transport layer the ability to defer flow errors. It assumes that QP users understand their fabric's characteristics and can select appropriate parameter values to optimize performance.

5.6.2.3.3.2 2. ROCE_ACCL Layout and Relevant Field Description

2.1. ROCE_ACCL Register

Offset	Bits	Name	Description	Access
00h	28	adp_retx_profile_select	In query, indicates that adp_retx_profile_id and adp_retx_profile are supported and valid. In set, indicates that the profile indicated adp_retx_profile_id is modified with the parameters in adp_retx_profile.	RW
	0	roce_adp_retrans_field_select	In Query, indicates if roce_adp_retrans mode is configurable. In set, indicates roce_adp_retrans_en field is modified.	RW

Offset	Bits	Name	Description	Access
04h	30:28	`adp_retx_profile_id`	The index of the adaptive-retransmission profile to query or to modify. Value 0 is reserved. Valid if `adp_retrans_timeouts_profile_select` is set. Must comply with `adp_retx_profile_max_id`.	Index
	0	`roce_adp_retrans_en`	If set, the adaptive retransmission window functionality is enabled. Valid only when `roce_adp_retrans_field_select` == 1.	RW
08h	30:28	`adp_retx_profile_max_range_num`	Maximal number of timeout ranges in `adp_retx_profile`.	RO
	26:24	`adp_retx_profile_max_id`	Maximal index supported for `adp_retx_profile_id`.	RO
	19:0	`adp_retx_base_timeout_min`	The minimal base-timeout value for `adp_retx_profile`, given in nano second units.	RO
10h-3Ch	384	`adp_retx_profile`	Adaptive Retransmission Profile.	RW

2.2. Adaptive Retransmission Profile

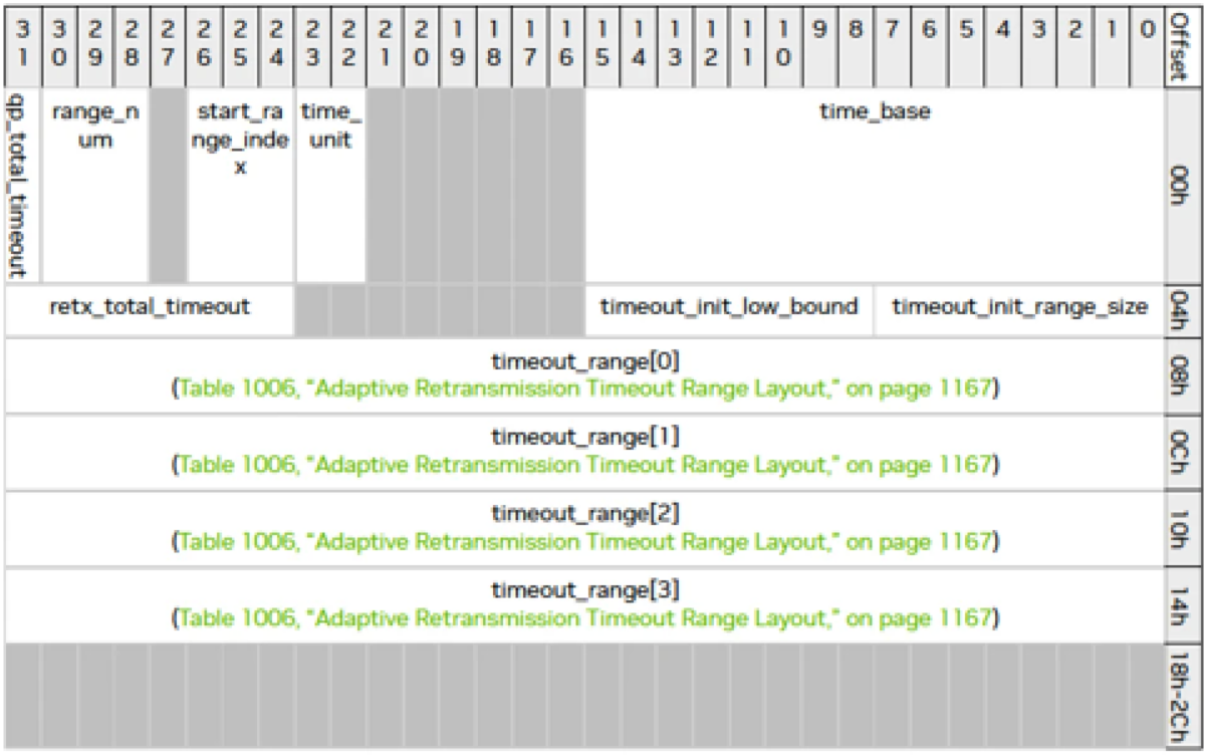
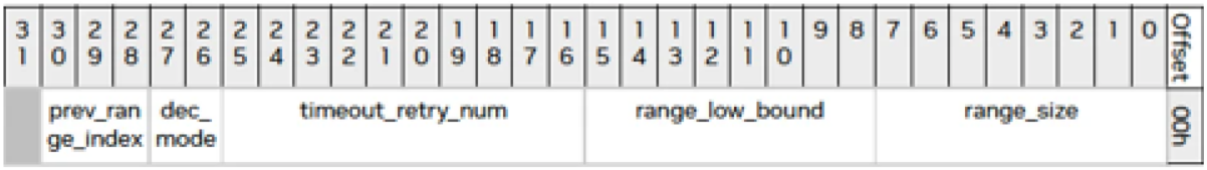


Table 1006 - Adaptive Retransmission Timeout Range Layout



Offset	Bits	Name	Description	Access
00h	31	**qp_total_timeout**	If set, Total Retransmission Timeout will use QP parameters as timeout value, given by: • **QP.retry_count** * **QP.timeout** or **QP.extended_retry_count** * **QP.timeout** • If clr, **retx_total_timeout** is used	
	30:28	**range_num**	Number of valid ranges in **timeout_range[]** array.	
	26:24	**start_range_index**	Index of **timeout_range** used as the first range following QP creation.	
	23:22	**time_unit**	Time granularity of **time_base** • **0x1** : **TIME_USEC** – µsecs • Other values are reserved	
	15:0	**time_base**	Base value for all time parameter, units defined by **time_unit** .	

Offset	Bits	Name	Description	Access
04h	31:24	`retx_total_timeout`	Total Retransmission Timeout, given by `base_time` << `retx_total_timeout`. Valid if `qp_total_timeout` is `0x0`.	
	15:8	`timeout_init_low_bound`	Lower bound for QP initial timeout randomization, given by `base_time` << `timeout_init_low_bound`	
	7:0	`timeout_init_range_size`	Upper bound for QP initial timeout randomization, given by `base_time` << (`timeout_init_low_bound` + `timeout_init_range_size`)	
08h-14h	31:0	`timeout_range[4]`	Array of timeout ranges. First `range_num` entries are valid.	

2.3. Adaptive Retransmission Timeout Range

Offset	Bits	Name	Description	Access
00h	30:28	`prev_range_index`	Index of the timeout range to used when the current timeout is the range's lower-bound, and timeout should be decreased.	
	27:26	`dec_mode`	Defines the behavior of timeout decrease until lower bound is reached. • `0x0` : `TO_DIV_4` • `0x1` : `TO_DIV_2` • `0x2` : `TO_LOW_BOUND` • Other values are reserved	
	25:16	`timeout_retry_num`	Number of consecutive retransmissions attempts with the same timeout value, before timeout is increased.	
	15:8	`range_low_bound`	Lower bound of the timeout range, given by `base_time` << `range_low_bound`	
	7:0	`range_size`	Upper bound of the timeout range given by `base_time` << (`range_low_bound` + `range_size`)	

5.6.2.3.3.3 3. Enablement and Capabilities

The following are ADP Retransmission configuration details:

- `roce_adp_retrans_field_select` – Indicates that ADP Retransmission functionality is supported by the device.
- `adp_retx_profile_select` – Indicates support for advanced ADP Retransmission profile configuration.
- `roce_adp_retrans_en` – Enables ADP Retransmission.
- `adp_retx_profile_id` – Activates an advanced ADP Retransmission profile with the specified ID. Currently, only profile #1 is available for users; profile #0 is reserved as the default.

- **adp_retx_base_timeout_min** – Minimum base timeout enforced by firmware: **0xFA0** (4000 nanoseconds, i.e., 4 microseconds).
- **adp_retx_profile_max_id** – Maximum of 1 profile supported by the firmware.
- **adp_retx_profile_max_range_num** – Maximum of 4 timeout ranges supported per profile.

5.6.2.3.3.4 4. Algorithm & Profile Description and Formulas

The following is the adaptive retransmission (ADP) timeout algorithm flow.

4.1. Initialization

- Initial timeout value:

```
rand(profile.timeout_init_low_bound, profile.timeout_init_low_bound + profile.timeout_init_range_size - 1)
```

- Hardware timer:

```
profile.time_base * (2 ^ granularity)
```

  Granularity = initial timeout.
- On the first timeout event:
  - If **current_timeout** falls within any defined range (**range_low_bound** → **range_low_bound + range_size**), the algorithm continues within that range.
  - Otherwise, it starts from **profile.timeout_range[profile.start_range_index].range_low_bound**.

Notes:

- Initial timeout is randomized to prevent multiple QPs from starting with the same value.
- A good configuration ensures the initial timeout falls within a defined range.
- The initial timeout value is retried only once.
- **time_base** must be a power of 2, with the minimum allowed value per firmware capability (4 µs).

4.2. On Consecutive Timeout Events

- **current_timeout** is used **profile.timeout_range[current_range].timeout_retry_num** times.
- Once exhausted, **current_timeout** is doubled until it reaches

```
profile.timeout_range[current_range].range_low_bound + profile.timeout_range[current_range].range_size
```

- The algorithm then continues with the next defined range.

Notes:

- Each range defines one or more timeout values.
- Timeout ranges must be sorted by **range_low_bound** (**range_low_bound[i] < range_low_bound[j]** for **i < j**).
- Maximum timeout is capped by **QP.ack_timeout**, regardless of configured ranges.
- The overall timeout is determined by:

- `profile.qp_total_timeout ? (QP.ack_timeout * QP.retry_num) : profile.retx_total_timeout`
- If no forward progress is made within the total timeout, the QP fails and transitions to ERR state with error code `IBV_WC_RETRY_EXC_ERR (Transport Retry Counter Exceeded)`.

4.3. On Forward Progress

- `current_timeout` is decremented based on `profile.timeout_range[current_range].dec_mode`:
  - Decrement by 2
  - Decrement by 4
  - Or reset to `profile.timeout_range[current_range].range_low_bound` of the current range.
- If `current_timeout` reaches the lowest timeout in the range, move to the previous range defined by `profile.timeout_range[current_range].prev_range_index`.

Notes:

- The minimum enforced timeout is `adp_retx_base_timeout_min` (4 µs).
- `prev_range_index` must always be less than `current_range`.

4.4. General Notes

- If no profile is selected, the ADP algorithm defaults to firmware-defined timeouts.
- When a profile is selected, it applies to all QPs of the PF and its associated VFs.
- Reverting to firmware-defined timeouts requires a fwreset or system reboot.
- To check which mode is active:
  - `adp_retx_profile_id = 0` → firmware-defined timeouts
  - `adp_retx_profile_id > 0` → profile-based timeouts

5.6.2.3.3.5 5. Sample Usage

The following examples show how to review and update the Adaptive Retransmission (ADP) profile using both:

- The standard `ROCE_ACCL` register
- The advanced mechanism (`reg_id 0x402c`)

> ⚠ The examples below use `mlxreg`, but the same usage applies when using the `mstreg` tool.

```
[]$ sudo mlxreg -d 1a:00.0 --yes --reg_name ROCE_ACCL --get
Sending access register...

Field Name                                  | Data
============================================================
roce_adp_retrans_field_select               | 0x00000001
roce_tx_window_field_select                 | 0x00000001
roce_slow_restart_field_select              | 0x00000001
roce_slow_restart_idle_field_select         | 0x00000001
min_ack_timeout_limit_disabled_field_select | 0x00000001
adaptive_routing_forced_en_field_select     | 0x00000000
selective_repeat_forced_en_field_select     | 0x00000001
dc_half_handshake_en_field_select           | 0x00000000
cc_per_plane_en_field_select                | 0x00000000
ack_dscp_force_field_select                 | 0x00000001
cc_burst_size_select                        | 0x00000001
ps_metadata_reg_id_select                   | 0x00000000
ifa2_turn_around_select                     | 0x00000001
```

```
adp_retx_profile_select                     | 0x00000001
roce_adp_retrans_en                         | 0x00000001
roce_tx_window_en                           | 0x00000000
roce_slow_restart_en                        | 0x00000001
roce_slow_restart_idle_en                   | 0x00000000
min_ack_timeout_limit_disabled              | 0x00000000
adaptive_routing_forced_en                  | 0x00000000
selective_repeat_forced_en                  | 0x00000000
dc_half_handshake_en                        | 0x00000000
cc_per_plane_en                             | 0x00000000
ack_dscp_force                              | 0x00000000
ack_dscp                                    | 0x00000000
ps_metadata_reg_id                          | 0x00000000
cc_burst_size                               | 0x00000000
Ifa2_turn_around_en                         | 0x00000000
adp_retx_profile_id                         | 0x00000000
adp_retx_base_timeout_min                   | 0x00000fa0
adp_retx_profile_max_id                     | 0x00000001
adp_retx_profile_max_range_num              | 0x00000004
time_base                                   | 0x00000000
time_unit                                   | 0x00000000
start_range_index                           | 0x00000000
range_num                                   | 0x00000000
qp_total_timeout                            | 0x00000000
timeout_init_range_size                     | 0x00000000
timeout_init_low_bound                      | 0x00000000
retx_total_timeout                          | 0x00000000
range_size_0                                | 0x00000000
range_low_bound_0                           | 0x00000000
timeout_retry_num_0                         | 0x00000000
dec_mode_0                                  | 0x00000000
prev_range_index_0                          | 0x00000000
range_size_1                                | 0x00000000
range_low_bound_1                           | 0x00000000
timeout_retry_num_1                         | 0x00000000
dec_mode_1                                  | 0x00000000
prev_range_index_1                          | 0x00000000
range_size_2                                | 0x00000000
range_low_bound_2                           | 0x00000000
timeout_retry_num_2                         | 0x00000000
dec_mode_2                                  | 0x00000000
prev_range_index_2                          | 0x00000000
range_size_3                                | 0x00000000
range_low_bound_3                           | 0x00000000
timeout_retry_num_3                         | 0x00000000
dec_mode_3                                  | 0x00000000
prev_range_index_3                          | 0x00000000
============================================================

[]$ sudo mlxreg -d 1a:00.0 --yes --reg_id 0x402c --reg_len 0x40 --get
Sending access register...

Address | Data
======================
0x00000000 | 0x1800065f
0x00000004 | 0x00000005
0x00000008 | 0x41000fa0
0x0000000c | 0x00000000
0x00000010 | 0x00000000
0x00000014 | 0x00000000
0x00000018 | 0x00000000
0x0000001c | 0x00000000
0x00000020 | 0x00000000
0x00000024 | 0x00000000
0x00000028 | 0x00000000
0x0000002c | 0x00000000
0x00000030 | 0x00000000
0x00000034 | 0x00000000
0x00000038 | 0x00000000
0x0000003c | 0x00000000
======================

// Equivalent usage with ROCE_ACCL reg_name commented out for clarity and future usage
[]$ ##sudo mlxreg -d 1a:00.0 --reg_name ROCE_ACCL --set
roce_adp_retrans_en=0x1,adp_retx_profile_id=0x1,time_base=4,time_unit=1,start_range_index=0,range_num=1,qp_total_ti
meout=1,timeout_init_range_size=2,timeout_init_low_bound=0,range_size_0=0x2,range_low_bound_0=0,timeout_retry_num_0
=0x1,dec_mode_0=1,prev_range_index_0=0

[]$ sudo mlxreg -d 1a:00.0 --reg_id 0x402c --reg_len 0x40 --set
"0x4.0:1=1,0x4.28:3=1,0x10.0:16=4,0x10.22:2=1,0x10.24:3=0,0x10.28:3=1,0x10.31:1=1,0x14.0:8=2,0x14.8:8=0,0x18.0:8=2,
0x18.8:8=0,0x18.16:10=1,0x18.26:2=1"
You are about to send access register id: 0x402c with the following data:
Address | Data
======================
0x00000000 | 0x1800065f
0x00000004 | 0x10000005
0x00000008 | 0x41000fa0
0x0000000c | 0x00000000
0x00000010 | 0x90400004
0x00000014 | 0x00000002
0x00000018 | 0x04010002
0x0000001c | 0x00000000
0x00000020 | 0x00000000
0x00000024 | 0x00000000
0x00000028 | 0x00000000
0x0000002c | 0x00000000
0x00000030 | 0x00000000
0x00000034 | 0x00000000
0x00000038 | 0x00000000
0x0000003c | 0x00000000
======================

 Do you want to continue ? (y/n) [n] : y
 Sending access register...
[]$ sudo mlxreg -d 1a:00.0 --yes --reg_name ROCE_ACCL --get
```

```
Sending access register...

Field Name                                  | Data
============================================================
roce_adp_retrans_field_select               | 0x00000001
roce_tx_window_field_select                 | 0x00000001
roce_slow_restart_field_select              | 0x00000001
roce_slow_restart_idle_field_select         | 0x00000001
min_ack_timeout_limit_disabled_field_select | 0x00000001
adaptive_routing_forced_en_field_select     | 0x00000000
selective_repeat_forced_en_field_select     | 0x00000001
dc_half_handshake_en_field_select           | 0x00000000
cc_per_plane_en_field_select                | 0x00000000
ack_dscp_force_field_select                 | 0x00000001
cc_burst_size_select                        | 0x00000001
ps_metadata_reg_id_select                   | 0x00000000
ifa2_turn_around_select                     | 0x00000001
adp_retx_profile_select                     | 0x00000001
roce_adp_retrans_en                         | 0x00000001
roce_tx_window_en                           | 0x00000000
roce_slow_restart_en                        | 0x00000001
roce_slow_restart_idle_en                   | 0x00000000
min_ack_timeout_limit_disabled              | 0x00000000
adaptive_routing_forced_en                  | 0x00000000
selective_repeat_forced_en                  | 0x00000000
dc_half_handshake_en                        | 0x00000000
cc_per_plane_en                             | 0x00000000
ack_dscp_force                              | 0x00000000
ack_dscp                                    | 0x00000000
ps_metadata_reg_id                          | 0x00000000
cc_burst_size                               | 0x00000000
Ifa2_turn_around_en                         | 0x00000000
adp_retx_profile_id                         | 0x00000001
adp_retx_base_timeout_min                   | 0x00000fa0
adp_retx_profile_max_id                     | 0x00000001
adp_retx_profile_max_range_num              | 0x00000004
time_base                                   | 0x00000004
time_unit                                   | 0x00000001
start_range_index                           | 0x00000000
range_num                                   | 0x00000001
qp_total_timeout                            | 0x00000001
timeout_init_range_size                     | 0x00000002
timeout_init_low_bound                      | 0x00000000
retx_total_timeout                          | 0x00000000
range_size_0                                | 0x00000002
range_low_bound_0                           | 0x00000000
timeout_retry_num_0                         | 0x00000001
dec_mode_0                                  | 0x00000001
prev_range_index_0                          | 0x00000000
range_size_1                                | 0x00000000
range_low_bound_1                           | 0x00000000
timeout_retry_num_1                         | 0x00000000
dec_mode_1                                  | 0x00000000
prev_range_index_1                          | 0x00000000
range_size_2                                | 0x00000000
range_low_bound_2                           | 0x00000000
timeout_retry_num_2                         | 0x00000000
dec_mode_2                                  | 0x00000000
prev_range_index_2                          | 0x00000000
range_size_3                                | 0x00000000
range_low_bound_3                           | 0x00000000
timeout_retry_num_3                         | 0x00000000
dec_mode_3                                  | 0x00000000
prev_range_index_3                          | 0x00000000
============================================================
```

5.6.2.3.4 Migration to RDMA-Core

As a leading RDMA-based interconnect provider, NVIDIA strives to bring all its software drivers and features to the Linux Upstream in order to guarantee best out-of-the box experience.

The NVIDIA Linux driver development methodology is “Upstream first”. Therefore, NVIDIA contributes all its developed features to the RDMA subsystem before porting them to its own Linux driver, that is, MLNX_OFED.

Each generation of the ConnectX® network adapters includes dedicated hardware that unleashes cutting-edge RDMA features, delivering compute and data intensive applications with highest performance and scalability. The RDMA subsystem verbs library is a generic API library serving all vendors and all RDMA applications.

To enable the usage of new cutting-edge technologies as soon as they became available, NVIDIA came up with an interim solution to work around the long wait periods before these features become supported and incorporated in Linux Upstream. As such, and until a faster acceptance process for the verbs API was implemented for the RDMA subsystem, NVIDIA has developed and maintained a private API with the prefix **ibv_exp** (known as “experimental verbs”). This private API

enabled NVIDIA to expose early on the latest hardware offloads to its customers' applications via the NVIDIA Linux driver (MLNX_OFED).

Nowadays, the acceptance process has become faster. Therefore, the private verbs API has become redundant. NVIDIA has set new APIs in Linux Upstream, which can be used to achieve most of the legacy experimental verbs' functionalities and more. Going forward, future RDMA-related features will only be available through the Upstream RDMA subsystem.

The userspace RDMA subsystem (known as RDMA-Core) library introduces the following new libmlx5 API channels:

- **mlx5dv** – a channel for configuring hardware objects with NVIDIA specific attributes.
- **mlx5dv_devx** – a channel for exposing raw NVIDIA hardware objects, enabling the use of hardware features with minimal Kernel changes (some of the mlx5dv APIs use **mlx5dv_devx** at userspace level).

Application owners who wish to use the latest RDMA-based in-network computing acceleration engines and hardware offloads are kindly asked to use the RDMA-Core APIs and avoid using the experimental verbs APIs.

Application owners currently using the experimental verbs (**ibv_exp**) APIs are highly encouraged to begin porting their applications to use the RDMA-core APIs instead. The porting process is for the most part straight-forward.

NVIDIA plans to deprecate the experimental verbs API in the near future, following an advance twelve-month notice.

As of MLNX_OFED v4.7:

- No new features will be added to the experimental verbs API.
- **mlx5dv** and **mlx5dv_devx** APIs will become default APIs. However, the experimental verbs API will not be exposed.
- Application owners can still choose to install the experimental verbs API and use the verbs (using a special installation flag).

5.6.2.3.4.1 MLNX_OFED Features Verbs and Capabilities

- Accelerated Verbs
- AS Notify
- Atomic Operations
- Packet Checksum Offload
- Contiguous Pages
- CQE Compression
- CQ Moderation
- Cross Channel
- Device Memory
- Dynamically Connected QPs
- Flow Entropy
- Flow Steering Support

- Hardware Offload for Erasure Code
- Inline Receive
- Memory Window
- Multi-packet RQ
- No Operation
- Out-of-order
- PeerDirect Sync
- NVIDIA PeerDirect Async
- Physical Address Memory Allocation
- Prefetch Memory Region
- Query GID Attributes
- Raw Ethernet
- Registration and Re-registration of Memory Region
- Resource Domain
- RoCE Time-stamping
- Shared Memory Region
- Tag Matching
- TCP Segmentation Offload
- Tunneled Atomic
- User-mode Memory Registration
- VLAN Offload

Accelerated Verbs

The accelerated verbs should be used to Post-Receive to the created WQs and to poll for completions.

Experimental Accelerated Verbs

- **ibv_exp_query_intf**
- **ibv_exp_release_intf**

Accelerated Verbs RDMA-Core Support

Accelerated verbs are replaced in RDMA-Core by the mlx5dv API in **libmlx5**.

AS Notify

AS Notify is a low-latency hardware-based thread wakeup mechanism. Instead of actively polling a Completion Queue (CQ), the user application can arm the CQ and issue a “wait” instruction to put the user thread to sleep. The user application will be woken up by AS_notify interrupt once a completion event takes place. Note that when AS_notify interrupt cannot be triggered, firmware will fall back into the traditional MSI interrupt.

AS Notify Experimental Verbs

- **ibv_exp_create_cq**
  - **IBV_EXP_CQ_AS_NOTIFY**

AS Notify Experimental Capabilities

`IBV_EXP_CQ_AS_NOTIFY`

AS Notify RDMA-Core Support

> ⚠ This feature is currently not supported by RDMA-Core. For further information, please contact Support.

Atomic Operations

Atomic Operations execute a 64-bit operation at a specified address on a remote node. The operations atomically read, modify and write the destination address and guarantee that operations on this address by other QPs on the same CA do not occur between the Read and Write. The scope of the atomicity guarantee may optionally extend to other CPUs and HCAs.

Atomic Operations Experimental vs. RDMA-Core Verbs and Device Attributes

<table>
<tr><th colspan="2">Experimental</th><th colspan="2">RDMA-Core</th></tr>
<tr><td colspan="4">Verbs</td></tr>
<tr><td rowspan="2">ibv_exp_query_device</td><td>exp_atomic_cap
• IBV_EXP_ATOMIC_NONE
• IBV_EXP_ATOMIC_HCA
• IBV_EXP_ATOMIC_GLOB</td><td>ibv_query_device_ex</td><td>• IBV_ATOMIC_NONE
• IBV_ATOMIC_HCA
• IBV_ATOMIC_GLOB</td></tr>
<tr><td>pci_atomic_caps</td><td></td><td>pci_atomic_caps</td></tr>
<tr><td rowspan="2">ibv_exp_reg_mr</td><td></td><td rowspan="2">ibv_reg_mr</td><td></td></tr>
<tr><td>IBV_EXP_ACCESS_REMOTE_ATOMIC</td><td>IBV_ACCESS_REMOTE_ATOMIC</td></tr>
<tr><td colspan="4">Device Attributes</td></tr>
<tr><td colspan="2">IBV_EXP_DEVICE_ATTR_PCI_ATOMIC_CAPS</td><td colspan="2"></td></tr>
</table>

Extended Atomics

Extended atomic operations are a set of operations beyond those defined by the IB Spec. Atomicity guarantees, Atomic Ack generation, ordering rules and error behavior for this set of extended atomic operations is the same as that for IB standard Atomic operations.

Extended Atomics Experimental Verbs

- ibv_exp_post_send
  - IBV_EXP_WR_EXT_MASKED_ATOMIC_CMP_AND_SWP
  - IBV_EXP_WR_EXT_MASKED_ATOMIC_FETCH_AND_ADD
  - IBV_EXP_SEND_EXT_ATOMIC_INLINE
  - ext_op.masked_atomics
- ibv_exp_create_qp
  - max_atomic_arg
- ibv_exp_poll_cq

    - IBV_EXP_WC_MASKED_COMP_SWAP
    - IBV_EXP_WC_MASKED_FETCH_ADD
- ibv_exp_query_device
    - ext_atom

Extended Atomics Experimental Capabilities and Device Attributes

- IBV_EXP_DEVICE_EXT_ATOMICS
- IBV_EXP_DEVICE_EXT_MASKED_ATOMICS
- IBV_EXP_DEVICE_ATTR_EXP_CAP_FLAGS
- IBV_EXP_DEVICE_ATTR_EXT_ATOMIC_ARGS
- IBV_EXP_DEVICE_ATTR_MASKED_ATOMICS

RDMA-Core Support

⚠ This feature is currently not supported by RDMA-Core. For further information, please contact Support.

Atomic Response Endianness

Atomic Operation in Connect-IB (MT27600) are fully supported on big endian (BE) machines (e.g. PPC). Their support is limited to little-endian machines (e.g. x86).

When using ibv_exp_query_device on little-endian machines with Connect-IB, the attr.exp_atomic_cap is set to IBV_EXP_ATOMIC_HCA_REPLY_BE, which indicates that if enabled, the atomic operation replied value is big-endian and contradicts the host endianness.

Enabling atomic operation with this endianness contradiction is by setting the IBV_EXP_QP_CREATE_QTOMIC_BE_REPLY flag in exp_create_flag in ibv_exp_create_qp.

RDMA-Core libmlx5 will not verify any atomic QP creation flag upon secnifn Atomic operations.

If an application swaps data between BE and host endianness over legacy library, it shall continue doing so.

For further information, please contact Support.

Packet Checksum Offload

The device supports calculation of checksum on transmitted packets and validation of received packets checksum. The adapter device offloads IPv4 checksum (L3) and TCP/UDP checksum (L4).

Checksum calculation is supported for TCP/UDP running over IPv4 and IPv6.

Checksum Offload Experimental Verbs

- ibv_exp_post_send
    - IBV_EXP_SEND_IP_CSUM

Checksum Offload Experimental Capabilities

- IBV_EXP_DEVICE_RX_CSUM_TCP_UDP_PKT

- IBV_EXP_DEVICE_RX_CSUM_IP_PKT
- IBV_EXP_SW_PARSING_CSUM

Checksum Offload RDMA-Core Verbs

- ibv_post_send
  - IBV_SEND_IP_CSUM
- ibv_query_device_ex
  - IBV_RAW_PACKET_CAP_IP_CSUM
- ibv_wr_post
  - IBV_SEND_IP_CSUM

Checksum Offload RDMA-Core Experimental Capabilities

- IBV_DEVICE_UD_IP_CSUM
- IBV_DEVICE_RC_IP_CSUM
- IBV_DEVICE_RAW_IP_CSUM

Contiguous Pages

Contiguous Pages improves performance by allocating user memory regions over physical contiguous pages. It enables a user application to ask low level drivers to allocate contiguous memory for it as part of ibv_reg_mr .

Contiguous Pages Environmental Variables

Environmental Variables	Value
MLX_QP_ALLOC_TYPE	CONTIG, PREFER_CONTIG
MLX_CQ_ALLOC_TYPE	CONTIG, PREFER_CONTIG
MLX_MR_MAX_LOG2_CONTIG_BSIZE	_MAX_LOG2_CONTIG_BLOCK_SIZE (23)
MLX_MR_MIN_LOG2_CONTIG_BSIZE	_MIN_LOG2_CONTIG_BLOCK_SIZE (12)
HUGE_CQ	
qp_huge_key	

Contiguous Pages Experimental Verbs

- ibv_exp_reg_mr
  - IBV_EXP_ACCESS_ALLOCATE_MR
- ibv_exp_rereg_mr
  - IBV_EXP_ACCESS_ALLOCATE_MR

Contiguous Pages RDMA-Core Support

Contiguous memory in RDMA-Core is achieved using huge pages.

CQE Compression

CQE compression saves PCIe bandwidth by compressing a few CQEs into a smaller amount of bytes on the PCIe.

CQE Compression Environmental Variables

- `MLX5_ENABLE_CQE_COMPRESSION`

CQE Compression Experimental Verbs

- `ibv_exp_create_cq`
  - `IBV_EXP_CQ_COMPRESSED_CQE`

CQE Compression Capabilities

- `IBV_EXP_CQ_COMPRESSED_CQE`

CQE Compression RDMA-Core Verbs

- `mlx5dv_create_cq`
  - `cqe_comp_res_format`
  - `MLX5DV_CQ_INIT_ATTR_MASK_COMPRESSED_CQE`

CQE Compression RDMA-Core Capabilities

- `mlx5dv_query_device`
  - `mlx5dv_cqe_comp_caps`

Relevant Man Pages

- mlx5dv_create_cq

CQ Moderation

CQ moderation enables the user to moderate completion events by specifying the number of completions that cause an event and the timeout in micro seconds to cause the event even if the number of completions was not reached.

CQ Moderation Experimental Verbs

- `ibv_exp_modify_cq`
  - `cq_count`
  - `cq_period`

CQ Moderation Experimental Capabilities

`IBV_EXP_CQ_ATTR_MODERATION`

CQ Moderation RDMA-Core Verbs

- `ibv_modify_cq`
  - `ibv_moderate_cq`

Relevant Man Pages

- ibv_modify_cq

Cross Channel

Cross Channel is a Verbs API that enables one to define a list of communication tasks and synchronization points and post this list as a single WQE which the HCA progresses entirely. Once posted, only completion of the list is polled for by the CPU. In this way, one can create and schedule complex communication and coordination patterns among all nodes in a cluster.

Cross Channel Experimental Verbs

- **ibv_exp_post_task**
- **ibv_exp_create_qp**
  - **IBV_EXP_QP_CREATE_CROSS_CHANNEL**
  - **IBV_EXP_QP_CREATE_MANAGED_SEND**
  - **IBV_EXP_QP_CREATE_MANAGED_RECV**
  - **IBV_EXP_QP_CREATE_IGNORE_SQ_OVERFLOW**
  - **IBV_EXP_QP_CREATE_IGNORE_RQ_OVERFLOW**
- **ibv_exp_create_cq**
  - **IBV_EXP_CQ_CREATE_CROSS_CHANNEL**
- **ibv_exp_modify_cq**
  - **IBV_EXP_CQ_IGNORE_OVERRUN**
- **ibv_exp_post_send**
  - union task
  - **ibv_exp_calc_op**
  - **ibv_exp_calc_data_type**
  - **ibv_exp_calc_data_size**
  - **IBV_EXP_WR_SEND_ENABLE**
  - **IBV_EXP_WR_RECV_ENABLE**
  - **IBV_EXP_WR_CQE_WAIT**
- **ibv_exp_query_device**
  - **calc_cap**

Cross Channel Experimental Capabilities and Device Attributes

- **IBV_EXP_DEVICE_CROSS_CHANNEL**
- **IBV_EXP_DEVICE_ATTR_CALC_CAP**

Cross Channel RDMA-Core Support

This feature is currently not supported by RDMA-Core. For further information, please contact Support.

Device Memory

Device Memory is a verbs API that allows using on-chip memory, located on the device, as a data buffer for send/receive and RDMA operations. The device memory can be mapped and accessed directly by user and kernel applications, and can be allocated in various sizes, registered as memory regions with local and remote access keys for performing the send/ receive and RDMA operations. Using the device memory to store packets for transmission can significantly reduce transmission latency compared to the host memory.

Device Memory Experimental Verbs

- **ibv_exp_alloc_dm**
- **ibv_exp_memcpy_dm**
- **ibv_exp_free_dm**
- **ibv_exp_reg_mr**
    - **ibv_exp_reg_mr_dm**

Device Memory RDMA-Core Verbs

- **ibv_alloc_dm**
- **ibv_reg_dm_mr**
- **ibv_memcpy_to_dm**
- **ibv_memcpy_from_dm**
- **ibv_free_dm**

Relevant Man Pages

- ibv_alloc_dm

Example

See example in **libibverbs/examples/rc_pingpong.c** .

Dynamically Connected QPs

A dynamically connected transport service is an extension to transport services that enables a higher degree of scalability while maintaining high performance for sparse traffic. Utilization of Dynamically Connected (DC) transport reduces the total number of QPs required system-wide, by having QPs of reliable type dynamically connect and disconnect from any remote node.

DC QP is only supported in mlx5 driver.

DC QPs Experimental vs. RDMA-Core Verbs and Capabilities

Experimental		RDMA-Core	
Verbs			
ibv_exp_create_dct		**mlx5dv_create_qp** • qp_type = IBV_QPT_DRIVER • dv_init_attr.dc_init_attr.dc_type = MLX5DV_DCTYPE_DCT	
	dc_key		dct_access_key

	access flags		access flags
	flow_label		ah_attr.grh.flow_label
	Inline_size		Not supported by RDMA-Core
	IBV_EXP_DCT_OOO_RW_DATA_PLACEMENT		Supported through opensm
ibv_exp_destroy_dct		**ibv_destroy_qp**	
ibv_exp_query_dct		**ibv_query_qp**	
	dc_key		Not supported by RDMA-Core
	port		Port
	access_flags		access_flags
	min_rnr_timer		min_rnr_timer
	tclass		tclass
	flow_label		flow_label
	mtu		mtu
	pkey_index		pkey_index
	gid_index		gid_index
	hop_limit		hop_limit
	key_violations		Not supported by RDMA-Core
	state		Not supported by RDMA-Core
ibv_exp_post_send	dct_access_key, dct_number	**mlx5dv_wr_post**	
ibv_exp_poll_cq	IBV_EXP_WC_DCT	**mlx5dv_wr_set_dc_addr**	
ibv_exp_modify_qp	dct_key, IBV_EXP_QP_DC_KEY		Not supported by RDMA-Core
Capabilities and Device Attributes			
	IBV_EXP_DEVICE_DC_TRANSPORT	No DC capabilities flags needed in RDMA-Core	
	IBV_EXP_DEVICE_DC_RD_REQ, IBV_EXP_DEVICE_DC_RD_RES		
	IBV_EXP_DEVICE_DC_INFO		
	IBV_EXP_DEVICE_ATTR_MAX_DCT		
	IBV_EXP_TM_CAP_DC		

Relevant Man Pages

- mlx5dv_create_qp
- mlx5dv_wr_post

Example

```
/**Create DC QP**/
struct mlx5dv_qp_init_attr dv_init_attr;
struct ibv_qp_init_attr_ex init_attr;
```

```
memset(&dv_init_attr, 0, sizeof(dv_init_attr));
memset(&init_attr, 0, sizeof(init_attr));


init_attr.qp_type = IBV_QPT_DRIVER;
init_attr.send_cq = send_cq;
init_attr.recv_cq = recv_cq;
init_attr.pd = pd;


if (initiator) {                                                    /** DCI **/
                init_attr.comp_mask |= IBV_QP_INIT_ATTR_SEND_OPS_FLAGS | IBV_QP_INIT_ATTR_PD;
                init_attr.send_ops_flags |= IBV_QP_EX_WITH_SEND;


                dv_init_attr.comp_mask |=
                                        MLX5DV_QP_INIT_ATTR_MASK_DC |
                                        MLX5DV_QP_INIT_ATTR_MASK_QP_CREATE_FLAGS;
                dv_init_attr.create_flags |=
                                        MLX5DV_QP_CREATE_DISABLE_SCATTER_TO_CQE;
                dv_init_attr.dc_init_attr.dc_type = MLX5DV_DCTYPE_DCI;
} else {                                                            /** DCT **/
                init_attr.comp_mask |= IBV_QP_INIT_ATTR_PD;
                init_attr.srq = srq;
                dv_init_attr.comp_mask = MLX5DV_QP_INIT_ATTR_MASK_DC;
                dv_init_attr.dc_init_attr.dc_type = MLX5DV_DCTYPE_DCT;
                dv_init_attr.dc_init_attr.dct_access_key = DC_KEY;
}


qp = mlx5dv_create_qp(context, &init_attr, &dv_init_attr);


if (initiator) {
                ex_qp = ibv_qp_to_qp_ex(qp);
                dv_qp = mlx5dv_qp_ex_from_ibv_qp_ex(ex_qp);
}
```

```
/**DCI post send**/
struct ibv_ah_attr ah_attr;
ah_attr.dlid = rem_dest->lid;
ah_attr.port_num = ib_port;

ah = ibv_create_ah(pd, &ah_attr);
if (ah) {
    return -1;
}

ibv_wr_start(ex_qp);
ex_qp->wr_id = SEND_WRID;
ex_qp->wr_flags = IBV_SEND_SIGNALED;
ibv_wr_send(ex_qp);
mlx5dv_wr_set_dc_addr(dv_qp, ah, rem_dest->dctn, DC_KEY);
ibv_wr_set_sge(ex_qp, mr->lkey, (uint64_t)mr->addr, size);
ibv_wr_complete(ex_qp);
return 0;
```

DC RDMA Atomic

DC RDMA Atomic Experimental Verbs

- **ibv_exp_query_device**
  - **max_dc_req_rd_atom**
  - **max_dc_res_rd_atom**

DC RDMA Atomic RDMA-Core Support

This feature is currently not supported by RDMA-Core. For further information, please contact Support.

Dynamically Connected On-Demand-Paging

DC ODP Experimental Query Capabilities

- **ibv_exp_query_device**

- `dc_odp_caps`

DC ODP RDMA-Core Query Capabilities

- `mlx5dv_query_device`
  - `comp_mask` : `MLX5DV_CONTEXT_MASK_DC_ODP_CAPS`
  - `dc_odp_caps` with `enum ibv_odp_transport_cap_bits`

Enabling ODP for DC is done, as in all other transports, by setting the `IBV_ACCESS_ON_DEMAND` flag in `ibv_reg_mr` access parameter.

Example

```
struct mlx5dv_context attr_out = {};
attr_out.comp_mask |= MLX5DV_CONTEXT_MASK_DC_ODP_CAPS;
mlx5dv_query_device(context, &attr_out);
if (attr_out.dc_odp_caps & IBV_ODP_SUPPORT_SEND)
    printf("support DC ODP send operation");
```

Key Violations

Key Violations Experimental Verbs

- `ibv_exp_arm_dct`
- `ibv_exp_poll_dc_info`

Key Violations RDMA-Core Support

⚠ This feature is currently not supported by RDMA-Core. For further information, please contact Support.

Flow Entropy

Flow Entropy Experimental Verbs

- `ibv_exp_modify_qp`
  - `flow_entropy`

Flow Entropy Experimental Capabilities

`IBV_EXP_QP_ATTR_FLOW_ENTROPY`

Flow Entropy RDMA-Core Support

⚠ This feature is currently not supported by RDMA-Core. For further information, please contact Support.

Flow Steering Support

Flow steering is a model which steers network flows based on flow specifications to specific QPs. Those flows can be either unicast or multicast network flows. In order to maintain flexibility,

domains and priorities are used. Flow steering uses a methodology of flow attribute, which is a combination of L2-L4 flow specifications, a destination QP and a priority. Flow steering rules may be inserted either by using ethtool or by using InfiniBand verbs. The verbs abstraction uses a different terminology from the flow attribute (**ibv_flow_attr**), defined by a combination of specifications (**struct ibv_flow_spec_***).

Flow Steering Experimental vs. RDMA-Core Verbs and Capabilities

Experimental Verbs		RDMA-Core	
Verbs			
ibv_exp_create_flow		**ibv_create_flow**	
	IBV_EXP_FLOW_ATTR_*		IBV_FLOW_ATTR
	IBV_EXP_FLOW_ATTR_FLAGS_ALLOW_LOOP_BACK		IBV_FLOW_ATTR_FLAGS_ALLOW_LOOP_BACK
	IBV_EXP_FLOW_SPEC_ETH		IBV_FLOW_SPEC_ETH
	IBV_EXP_FLOW_SPEC_IB		Not supported by RDMA-CORE
	IBV_EXP_FLOW_SPEC_IPV4		IBV_FLOW_SPEC_IPV4
	IBV_EXP_FLOW_SPEC_IPV6		IBV_FLOW_SPEC_IPV6
	IBV_EXP_FLOW_SPEC_IPV4_EXT		IBV_FLOW_SPEC_IPV4_EXT
	IBV_EXP_FLOW_SPEC_IPV6_EXT		IBV_FLOW_SPEC_IPV6
	IBV_EXP_FLOW_SPEC_TCP		IBV_EXP_FLOW_SPEC_TCP
	IBV_EXP_FLOW_SPEC_UDP		IBV_FLOW_SPEC_UDP
	IBV_EXP_FLOW_SPEC_VXLAN_TUNNEL		IBV_FLOW_SPEC_VXLAN_TUNNEL
	IBV_EXP_FLOW_SPEC_INNER		IBV_FLOW_SPEC_INNER
	IBV_EXP_FLOW_SPEC_ACTION_TAG		IBV_FLOW_SPEC_ACTION_TAG
	IBV_EXP_FLOW_SPEC_ACTION_DROP		IBV_FLOW_SPEC_ACTION_DROP
	Ibv_exp_flow_spec_*		Ibv_flow_spec_*
	Ibv_exp_destroy_flow		Ibv_destroy_flow
Capabilities			
IBV_EXP_DEVICE_MANAGED_FLOW_STEERING		IBV_DEVICE_MANAGED_FLOW_STEERING	

Relevant Man Pages

- ibv_create_flow

Example

- https://github.com/linux-rdma/rdma-core/blob/master/libibverbs/man/ibv_create_flow.3

Hardware Offload for Erasure Code

HW Offload for Erasure Code Experimental Verbs

- `ibv_exp_alloc_ec_calc`
- `ibv_exp_dealloc_ec_calc`
- `ibv_exp_ec_encode_async`
- `ibv_exp_ec_encode_sync`
- `ibv_exp_ec_decode_async`
- `ibv_exp_ec_decode_sync`
- `intibv_exp_ec_update_async`
- `intibv_exp_ec_update_sync`
- `ibv_exp_ec_poll`
- `ibv_exp_ec_encode_send`

HW Offload for Erasure Code Experimental Capabilities and Device Attributes

- `IBV_EXP_DEVICE_EC_OFFLOAD`
- `IBV_EXP_DEVICE_ATTR_EC_CAPS`
- `IBV_EXP_DEVICE_ATTR_EC_GF_BASE`

HW Offload for Erasure Code RDMA-Core Support

This feature is currently not supported by RDMA-Core. For further information, please contact Support.

Inline Receive

When Inline-Receive is active, the HCA may write received data in to the receive WQE or CQE. Using Inline-Receive saves PCIe read transaction since the HCA does not need to read the scatter list, therefore it improves performance in case of short receive –messages. On poll CQ, the driver copies the received data from WQE/CQE to the user's buffers. Therefore, apart from querying Inline-Receive capability and Inline-Receive activation the feature is transparent to user application.

Inline Receive Experimental Verbs

- `ibv_exp_query_device`
  - `inline_recv_size`
- `ibv_exp_create_qp`
  - `max_inl_recv`
- `ibv_exp_create_dct`
  - `inline_size`

Inline Receive Experimental Device Attributes

`IBV_EXP_DEVICE_ATTR_INLINE_RECV_SZ`

Inline Receive RDMA-Core Verbs

- `mlx5dv_create_qp`

- `MLX5DV_QP_CREATE_ALLOW_SCATTER_TO_CQE`

Inline Receive RDMA-Core Environmental Variables

`MLX5_SCATTER_TO_CQE`

Relevant Man Pages

- mlx5dv_create_qp

Memory Window

Memory Window (MW) allows the application to have more flexible control over remote access to its memory. It is available only on physical functions or native machines. The two types of Memory Windows supported are: type 1 and type 2B.

Memory Window is intended for situations where the application wants to:

- Grant and revoke remote access rights to a registered region in a dynamic fashion with less of a performance penalty.
- Grant different remote access rights to different remote agents and/or grant those rights over different ranges within registered region.

MW Experimental vs. RDMA-Core Verbs, Variables and Capabilities

Experimental Verbs		RDMA-Core	
Verbs			
`ibv_exp_bind_mw`		`ibv_bind_mw`	
`ibv_exp_post_send`	`bind_mw`		`ibv_wr_bind_mw`
`ibv_exp_reg_mr`	`IBV_EXP_ACCESS_MW_BIND`	`ibv_reg_mr`	`IBV_ACCESS_MW_BIND`
Capabilities and Device Attributes			
`IBV_EXP_DEVICE_MEM_WINDOW`		`IBV_DEVICE_MEM_WINDOW`	
`IBV_EXP_DEVICE_MW_TYPE_2A`		`IBV_DEVICE_MEM_WINDOW_TYPE_2A`	
`IBV_EXP_DEVICE_MW_TYPE_2B`		`IBV_DEVICE_MEM_WINDOW_TYPE_2B`	
		`IBV_DEVICE_MEM_MGT_EXTENSIONS`	
Environmental Variables			
`MLX5_SHUT_UP_MW`		In RDMA-Core, it is possible to not use MW feature with the new post send API (`ibv_posr_wr`)	

Relevant Man Pages

- ibv_wr_post

- ibv_bind_mw

Multi-packet RQ

Multi-Packet RQ is a receive queue where multiple packets are written to the same WR data buffer. The feature improves performance and reduces memory footprint.

Multi-Packet RQ Experimental vs. RDMA-Core Verbs and Capabilities

Experimental		RDMA-Core	
Verbs			
`ibv_exp_query_device`	`supported_qps`: • `IBV_EXP_MP_RQ_SUP_TYPE_SRQ_TM` • `IBV_EXP_MP_RQ_SUP_TYPE_WQ_RQ`	Not supported by RDMA-Core.	
	`allowed_shifts`: • `IBV_EXP_MP_RQ_NO_SHIFT` • `IBV_EXP_MP_RQ_2BYTES_SHIFT`		
`ibv_exp_query_device` • `mp_rq_caps`	`min_single_wqe_log_num_of_strides`	`mlx5dv_query_device` • `mlx5dv_striding_rq_caps`	`min_single_wqe_log_num_of_strides`
	`max_single_wqe_log_num_of_strides`		`max_single_wqe_log_num_of_strides`
	`min_single_stride_log_num_of_bytes`		`min_single_stride_log_num_of_bytes`
	`max_single_stride_log_num_of_bytes`		`max_single_stride_log_num_of_bytes`
`ibv_exp_create_wq`	`mp_rq`	`mlx5dv_create_wq`	• `striding_rq_attrs` • `MLX5DV_WQ_INIT_ATTR_MASK_STRIDING_RQ`

`ibv_exp_poll_cq`	• `mp_wr`
	• `exp_wc_flags`
	  • `IBV_EXP_WC_MP_WR_MORE_IN_MSG`
	  • `IBV_EXP_WC_MP_WR_CONSUMED`
	  • `IBV_EXP_WC_RECV_NOP`
		Not supported by RDMA-Core.	

Capabilities and Device Attributes			
`IBV_EXP_DEVICE_ATTR_MP_RQ`		Not supported by RDMA-Core.	
`IBV_EXP_MP_RQ_SUP_TYPE_SRQ_TM`			

No Operation

No operation (NOP) feature is used for cache optimization.

NOP Experimental Verbs

- `ibv_exp_post_send`
  - `IBV_EXP_WR_NOP`

NOP Experimental Capabilities

`IBV_EXP_DEVICE_NOP`

NOP RDMA-Core Support

> ⚠ This feature is currently not supported by RDMA-Core. For further support, please contact Support.

Out-of-order

In certain fabric configurations, InfiniBand packets for a given QP may take up different paths in a network from source to destination. This results into packets being received in an out-of-order (OOO) manner. These packets can now be handled instead of being dropped, in order to avoid retransmission. Data will be placed into host memory in an out-of-order manner when out of order messages are received.

OOO Experimental Verbs

- `ibv_exp_create_dct`
  - `IBV_EXP_DCT_OOO_RW_DATA_PLACEMENT`

OOO Environmental Variables

`MLX5_RELAXED_PACKET_ORDERING_ON`

OOO Device Attributes

`IBV_EXP_DEVICE_ATTR_OOO_CAPS`

OOO Rdma-Core Support

OOO feature is enabled by default in RDMA-Core.

PeerDirect Sync

PeerDirect sync allows for attaching context to a peer client. The verb receives an attribute in addition to the ibv device, then it calls the original open device to create a context.

PeerDirect Sync Experimental Verbs

- `ibv_exp_open_device`
  - `peer_info`

RDMA-Core Support

> ⚠ This feature is currently not supported by RDMA-Core. For further information, please contact Support.

NVIDIA PeerDirect Async

NVIDIA PeerDirect™ Async sub-system gives PeerDirect hardware devices, such as GPU cards, dedicated AS accelerators, and so forth, the ability to take control over HCA in critical path offloading CPU. To achieve this, there is a set of verb calls and structures providing application with abstract description of operation sequences intended to be executed by peer device.

PeerDirect Async Experimental Verbs

- **ibv_exp_create_cq**
  - **ibv_exp_peer_direct_attr**
- **ibv_exp_create_qp**
  - **ibv_exp_peer_direct_attr**
- **ibv_exp_peer_commit_qp**
- **ibv_exp_rollback_qp**
- **ibv_exp_peer_peek_cq**
- **ibv_exp_peer_abort_peek_cq**
- Any verb in the header file **peer_ops.h**

RDMA-Core Support

This feature is currently not supported by RDMA-Core. For further information, please contact Support.

Physical Address Memory Allocation

Physical address memory region (**PA_MR**) allows for managing physical memory used for posting Send and Receive requests. This can benefit the performance of applications that register large memory regions with random access.

PA_MR Experimental Verbs

- **ibv_exp_reg_mr**
  - **IBV_EXP_ACCESS_PHYSICAL_ADDR**

PA_MR Device Capabilities

- **IBV_EXP_DEVICE_PHYSICAL_RANGE_MR**

PA_MR RDMA-Core Support

This feature is currently not supported by RDMA-Core. For further information, please contact Support.

Prefetch Memory Region

Prefetch memory region (MR) enables prefetch pages of an on-demand paging (ODP) memory region.

Prefetch Memory Region Experimental vs. RDMA-Core Verbs

Experimental Verbs	RDMA-Core Verbs
`ibv_exp_prefetch_mr` • `IBV_EXP_PREFETCH_WRITE_ACCESS`	`ibv_advise_mr` • `IBV_ADVISE_MR_ADVICE_PREFETCH` • `IBV_ADVISE_MR_ADVICE_PREFETCH_WRITE`

Relevant Man Page

- ibv_advise_mr

Example

See example in `libibverbs/examples/rc_pingpong.c` .

Query GID Attributes

Query GID attributes, such as GID type.

Query GID Attributes Experimental Verbs

- `ibv_exp_query_gid_attr`

Query GID Attributes RDMA-Core Support

GID type can be extracted from the `ibsysfs` .

Example

Review `ibv_query_gid_type` , `ibv_read_sysfs_file` from rdma-core: https://github.com/linux-rdma/rdma-core/blob/master/libibverbs/verbs.c.

Raw Ethernet

Raw Ethernet programming enables writing an application that bypasses the kernel stack. To achieve this, packet headers and offload options need to be provided by the application.

Raw Ethernet Experimental vs. RDMA-Core Verbs and Capabilities

Experimental		RDMA-Core	
Verbs			
`ibv_exp_create_q`p`		`ibv_create_qp` / `ibv_create_qp_ex`	
	`IBV_QPT_RAW_ETH`		`IBV_QPT_RAW_PACKET`
Capabilities			
	`IBV_EXP_DEVICE_WQ_DELAY_DROP`		`IBV_RAW_PACKET_CAP_DELAY_DROP`

Relevant Man Pages

- ibv_create_qp_ex

Registration and Re-registration of Memory Region

Re-registration memory region (MR) allows the user to change attributes of the MR. The user may change the PD, access flag or the address and length of the memory region.

MR Registration/Re-Registration Experimental vs. RDMA-Core Verbs

Experimental Verbs		RDMA-Core Verbs	
`ibv_exp_reg_mr`		`ibv_reg_mr`	
	`IBV_EXP_ACCESS_*`		`IBV_ACCESS_*`
	`IBV_EXP_ACCESS_ALLOCATE_MR`		Please refer to the Contiguous Pages section
	`IBV_EXP_ACCESS_SHARED_MR_*` `IBV_EXP_ACCESS_NO_RDMA`		Shared MR functionality is deprecated in OFED and not supported in RDMA-Core.
	`IBV_EXP_ACCESS_RELAXED`		*ODP relaxed* Replaced with *implicit ODP*
	`IBV_EXP_ACCESS_PHYSICAL_ADDR`		Please refer to the Physical Address Memory Allocation section.
	`IBV_EXP_ACCESS_TUNNELED_ATOMIC`		Not supported in RDMA-Core
	`IBV_EXP_REG_MR_DM`	`ibv_reg_dm_mr`	
`ibv_exp_rereg_mr`		`ibv_rereg_mr`	
	`IBV_EXP_REREG_MR_CHANGE_TRAN`SLATION`		`IBV_REREG_MR_CHANGE_TRANSLATION`
	`IBV_EXP_REREG_MR_CHANGE_PD`		`IBV_REREG_MR_CHANGE_PD`
	`IBV_EXP_REREG_MR_CHANGE_ACCESS`		`IBV_REREG_MR_CHANGE_ACCESS`
	`IBV_EXP_ACCESS_*`		`IBV_ACCESS_*`

Relevant Man Pages

- ibv_reg_mr
- ibv_rereg_mr

Example

See example in `libibverbs/examples/rc_pingpong.c` .

Resource Domain

Resource domain is a verb object that may be associated with a QP and CQ objects upon creation to enhance data-path performance.

Resource Domain Experimental Verbs

- ibv_exp_create_res_domain
- ibv_exp_destroy_res_domain
- ibv_exp_create_qp
  - ibv_exp_res_domain
- ibv_exp_query_device
  - max_ctx_res_domain
- ibv_exp_create_cq
  - ibv_exp_res_domain
- ibv_exp_create_wq
  - ibv_exp_res_domain

Resource Domain Experimental Capabilities

- IBV_EXP_DEVICE_ATTR_MAX_CTX_RES_DOMAIN

Resource Domain RDMA-Core Verbs

- ibv_alloc_td
- ibv_alloc_parent_domain
- ibv_dealloc_pd

Relevant Man Pages

- ibv_alloc_parent_domain
- ibv_alloc_td

RoCE Time-stamping

RoCE time-stamping allows you to stamp packets when they are sent to the wire or when they are received from the wire. The time stamp is given in raw hardware cycles, but can easily be converted into hardware-referenced nanoseconds-based-time. Additionally, it enables you to query the hardware for the hardware time, thus stamp other application's event and compare time.

RoCE Time-Stamping Experimental vs. RDMA-Core Verbs and Device Attributes

<table>
<tr><th colspan="2">Experimental</th><th colspan="2">RDMA-Core</th></tr>
<tr><th colspan="4">Verbs</th></tr>
<tr><td rowspan="2">ibv_exp_query_values</td><td></td><td rowspan="2">ibv_query_rt_values_ex</td><td></td></tr>
<tr><td>IBV_WXP_VALUES_HW_CLOCK_NS</td><td>ibv_query_rt_values_ex</td></tr>
</table>

	IBV_EXP_VALUES_HW_CLOCK	ibv_query_rt_values_ex	IBV_VALUES_MASK_RAW_CLOCK
	IBV_EXP_VALUES_CLOCK_INFO	mlx5dv_get_clock_info	
ibv_exp_cqe_ts_to_ns		mlx5dv_ts_to_ns	
ibv_exp_create_cq		ibv_create_cq_ex	
	IBV_EXP_CQ_TIMESTAMP		IBV_WC_EX_WITH_COMPLETION_TIMESTAM
	IBV_EXP_CQ_TIMESTAMP_TO_SYS_TIME		IBV_WC_EX_WITH_COMPLETION_TIMESTAMP_WALLCLOCK
ibv_exp_poll_cq		ibv_start_poll	
	• Timestamp • IBV_EXP_WC_WITH_TIMESTAMP		• ibv_wc_read_completion_ts • ibv_wc_read_completion_wallclock_ns
ibv_exp_query_device		ibv_query_device_ex	
	timestamp_mask		completion_timestamp_mask
	hca_core_clock		hca_core_clock
Device Attributes			
IBV_EXP_DEVICE_ATTR_WITH_TIMESTAMP_MASK	Device attributes are not needed in RDMA-Core.		
IBV_EXP_DEVICE_ATTR_WITH_HCA_CORE_CLOC			

Relevant Man Pages

- ibv_query_rt_values_ex
- mlx5dv_get_clock_info
- mlx5dv_ts_to_ns
- ibv_create_cq_ex

Example

See example in **libibverbs/examples/rc_pingpong.c** .

Shared Memory Region

This feature helps share a memory region (MR) between processes.

Shared MR Experimental Verbs

- **ibv_exp_reg_shared_mr**
- **ibv_exp_reg_mr**

- `IBV_EXP_ACCESS_SHARED_MR_*`
- `IBV_EXP_ACCESS_NO_RDMA`

Shared MR Experimental Capabilities

- `IBV_EXP_DEVICE_SHARED_MR`

Shared MR RDMA-Core Support

> ⚠ This feature is deprecated in MLNX_OFED driver and is not supported in RDMA-Core.

Tag Matching

Tag matching and rendezvous offload is a technology employed by NVIDIA to offload the processing of MPI messages from the host machine onto the network card. Employing this technology enables a zero copy of MPI messages, i.e. messages are scattered directly to the user's buffer without intermediate buffering and copies. It also provides a complete rendezvous progress by NVIDIA devices. Such overlap capability enables the CPU to perform the application's computational tasks while the remote data is gathered by the adapter.

Tag Matching Experimental Verbs

`ibv_exp_create_dct`

- `IB_EXP_SRQT_TAG_MATCHING`
- `MLX5_DCTC_OFFLOAD_TYPE_RNDV`

Tag Matching Experimental Capabilities

- `IBV_EXP_DEVICE_ATTR_TM_CAPS`
- `IBV_EXP_TM_CAP_DC`

Tag Matching RDMA-Core Support

> ⚠ This feature is currently not supported by RDMA-Core. For further information, please contact Support.

TCP Segmentation Offload

TCP segmentation offload (TSO) enables the adapter cards to accept a large amount of data with a size greater than the MTU size. The TSO engine splits the data into separate packets and inserts the user-specified L2/L3/L4 headers automatically per packet. With the usage of TSO, CPU is offloaded from dealing with a large throughput of data.

TSO Experimental vs. RDMA-Core Verbs and Capabilities

Experimental Verbs	RDMA-Core
Verbs	

ibv_exp_create_qp		ibv_create_qp_ex	
	max_tso_header		max_tso_header
ibv_exp_query_device		ibv_query_device_ex	
	tso_caps		tso_caps
ibv_exp_post_send	tso	• ibv_post_send • ibv_wr_send_tso	tso
Capabilities and Device Attributes			
IBV_EXP_DEVICE_ATTR_TSO_CAPS			

Relevant Man Pages

- ibv_create_qp_ex
- ibv_query_device_ex

Tunneled Atomic

Tunneled atomic updates RDMA write operations so that when an RDMA Write operation is issued, the payload indicates which atomic operation to perform, instead of being written to the memory region (MR).

Tunneled Atomic Experimental Verbs

- ibv_exp_reg_mr
  - IBV_EXP_ACCESS_TUNNELED_ATOMIC
- ibv_exp_query_device
  - tunneled_atomic_caps

Tunneled Atomic Experimental Capabilities

IBV_EXP_DEVICE_ATTR_TUNNELED_ATOMIC

Tunneled Atomic RDMA-Core Support

⚠ This feature is currently not supported by RDMA-Core. For further information, please contact Support.

User-mode Memory Registration

User-mode memory registration (UMR) is a fast registration mode which uses send queues (SQs). This feature enables the usage of RDMA operations and scatters data through appropriate memory keys on the remote side.

UMR Experimental Verbs

The following UMR and non-inline UMR experimental verbs are no longer the default verbs used for configuration of neither feature.

UMR Verbs

- `Ibv_exp_create_qp`
  - `exp_create_flags` : `IBV_EXP_QP_CREATE_UMR`
  - `max_inl_send_klms`
- `ibv_exp_query_device`
  - `umr_cap`
  - `umr_fixed_size_caps`
- `ibv_exp_post_send`
  - `exp_opcode` : `IBV_EXP_WR_UMR_FILL` , `IBV_EXP_WR_UMR_INVALIDATE`
  - `ext_op` : `umr (umr type, memory_objects, exp_access, modified_mr, base_addr, num_mrs, mem_reg_list, mem_repeat_block_list, repeat_count, stride_dim)`
- `ibv_exp_poll_cq`
  - `exp_opcode` : `IBV_EXP_WC_UMR`
- `ibv_exp_create_mr`
- `ibv_exp_query_mkey`

Non-inline UMR Verbs

- Environmental variable: `MLX*_POST_SEND_PREFER_BF`
- `ibv_exp_dealloc_mkey_list_memory`
- `ibv_exp_alloc_mkey_list_memory`

UMR Experimental Capabilities and Device Attributes

- `IBV_EXP_DEVICE_UMR` , `IBV_EXP_DEVICE_UMR_FIXED_SIZE` , `IBV_EXP_DEVICE_MR_ALLOCATE`
- `IBV_EXP_DEVICE_ATTR_UMR` , `IBV_EXP_DEVICE_ATTR_UMR_FIXED_SIZE_CAPS`

UMR RDMA-Core Verbs

UMR Verbs

- `ibv_create_qp_ex`
  - `send_ops_flags` : `IBV_QP_EX_WITH_BIND_MW` , `IBV_QP_EX_WITH_LOCAL_INV`
- `mlx5dv_create_qp`
  - `send_ops_flags` : `MLX5DV_QP_EX_WITH_MR_INTERLEAVED` , `MLX5DV_QP_EX_WITH_MR_LIST`
- `mlx5dv_wr_post`
  - `mlx5dv_wr_mr_interleaved`
  - `mlx5dv_wr_mr_list`
- `mlx5dv_wc_opcode`
  - `MLX5DV_WC_UMR`

Non-inline UMR Verbs

⚠ This feature is currently not supported by RDMA-Core. For further information, please contact Support.

UMR RDMA-Core Capabilities and Device Attributes

No UMR capabilities flags are needed in RDMA-Core.

Relevant Man Pages

- mlx5dv_create_qp
- mlx5dv_wr_post

Example

```
struct mlx5dv_qp_init_attr mlx5_qp_attr = {};
struct ibv_qp_init_attr_ex init_attr_ex = {};
struct ibv_qp     *qp;
struct ibv_qp_ex *qpx;
struct mlx5dv_qp_ex *dv_qp;
struct mlx5dv_mkey *dv_mkey;
struct mlx5dv_mkey_init_attr mkey_init_attr = {};
struct mlx5dv_mr_interleaved *array_interleaved;
int num_interleaved = 1;
int repeat_count = 2;
int message_size = 4096;
int skip_bytes_interleaved = 20;

mlx5_qp_attr.comp_mask = MLX5DV_QP_INIT_ATTR_MASK_SEND_OPS_FLAGS;
mlx5_qp_attr.send_ops_flags = MLX5DV_QP_EX_WITH_MR_INTERLEAVED;
init_attr_ex.cap.max_inline_data = 128;
init_attr_ex.send_ops_flags = IBV_QP_EX_WITH_SEND;

qp = mlx5dv_create_qp(context, &init_attr_ex, &mlx5_qp_attr);
qpx = ibv_qp_to_qp_ex(qp);
dv_qp = mlx5dv_qp_ex_from_ibv_qp_ex(qpx);
mkey_init_attr.create_flags = MLX5DV_MKEY_INIT_ATTR_FLAGS_INDIRECT;
mkey_init_attr.max_entries = 4;
mkey_init_attr.pd = pd;

dv_mkey = mlx5dv_create_mkey(&mkey_init_attr);
array_interleaved = calloc(1, num_interleaved * sizeof(struct mlx5dv_mr_interleaved));
qpx->wr_flags = IBV_SEND_INLINE | IBV_SEND_SIGNALED;
array_interleaved[0].addr = (uintptr_t)mr->addr;
array_interleaved[0].bytes_count = (message_size - skip_bytes_interleaved) / 2;
array_interleaved[0].bytes_skip = skip_bytes_interleaved / 2;
array_interleaved[0].lkey
= mr->lkey;
ibv_wr_start(qpx);
mlx5dv_wr_mr_interleaved (dv_qp, dv_mkey, access_flags,
                          repeat_count, num_interleaved, array_interleaved);
ret = ibv_wr_complete(ctx->qpx);
```

VLAN Offload

The device offloads VLAN insertion and stripping for raw Ethernet frames.

VLAN insertion is performed by the driver, inlining the VLAN tag into the Ethernet frame headers in the WQE Eth Segment.

VLAN Stripping is configured in RQ through the VLAN Stripping Disable (**vsd**) bit. When configured to perform VLAN Stripping, the device removes the VLAN tag from the incoming frames and reports it in the CQE fields.

VLAN Offload Experimental vs. RDMA-Core Verbs and Capabilities

	Experimental		RDMA-Core
Verbs			

ibv_exp_query_qp	wq_vlan_offloads_cap		Check caps through ibv_query_device_ex
ibv_exp_create_wq		ibv_create_wq	
	vlan_offloads • IBV_EXP_RECEIVE_WQ_CVLAN_STRIP • IBV_EXP_RECEIVE_WQ_CVLAN_INSERTION		• IBV_WQ_FLAGS_CVLAN_STRIPPING • VLAN insertion not supported by RDMA core
ibv_exp_modify_wq	vlan_offloads	ibv_modify_wq	flags IBV_WQ_FLAGS_CVLAN_STRIPPING
Capabilities and Device Attributes			
	IBV_EXP_DEVICE_ATTR_VLAN_OFFLOADS		IBV_RAW_PACKET_CAP_CVLAN_STRIPPING

Relevant Man Pages

- ibv_create_wq
- ibv_modify_wq
- ibv_query_device_ex

5.6.2.3.4.2 Additional Capabilities in MLNX_OFED

The following table lists additional experimental and RDMA-Core capabilities in MLNX_OFED.

Experimental	RDMA-Core
IBV_EXP_DEVICE_CAPI	NOT supported by RDMA-Core
IBV_EXP_DEVICE_VXLAN_SUPPORT	NOT supported by RDMA-Core
IBV_EXP_DEVICE_ATTR_PACKET_PACING_CAPS	ibv_packet_pacing_caps , ibv_modify_qp_rate_limit
IBV_EXP_DEVICE_ATTR_TUNNEL_OFFLOADS_CAPS	MLX5DV_CONTEXT_MASK_TUNNEL_OFFLOADS
IBV_EXP_DEVICE_ATTR_PCI_ATOMIC_CAPS	Not supported by RDMA-Core
IBV_EXP_DEVICE_ATTR_RX_PAD_END_ALIGN	IBV_DEVICE_PCI_WRITE_END_PADDING

5.6.2.3.4.3 Applications Support in RDMA-Core

Rping RTT Measurements

RTT measurements in examples/rping.c are not supported by RDMA-Core. This functionality is available only in the experimental legacy rping.c example using rping -m .

For further information, please contact Support.

5.6.2.3.5 Congestion Control Infrastructure

NVIDIA® BlueField® DPUs and NVIDIA® ConnectX® SuperNICs provide a set of production-ready congestion control algorithms that execute natively on the device's DPA-based congestion control engine—the same underlying infrastructure used by the programmable PCC framework. Spanning reactive and proactive congestion-avoidance approaches, these algorithms address a broad range of RoCE fabric conditions, using both explicit congestion notifications and delay-based detection to make real-time, per-flow rate decisions on the device.

NVIDIA provides two deployable native CC algorithms:

- DCQCN algorithm (legacy interoperability) – a closed-loop, ECN-based algorithm that coordinates Congestion Points, Notification Points, and Reaction Points.
- ZTR-RTT CC algorithm (GA) – augments the DCQCN feedback model with hardware-timestamped round-trip time (RTT) measurements, enabling Zero Touch RoCE deployments without specialized switch configuration.

Native and user-programmed CC are separate, mutually exclusive execution paths. The **USER_PROGRAMMABLE_CC** firmware setting determines which path is active. Within the selected path, the PPCC (Port Programmable Congestion Control) register provides per-port control for enabling algorithms and managing their parameters and counters.

The native algorithms expose tunable parameters that allow deployments to adjust congestion sensitivity, rate-recovery behavior, and latency objectives.

The same infrastructure underpins NVIDIA DOCA Programmable Congestion Control (PCC), enabling users to design and deploy custom CC algorithms tailored to their fabric and traffic characteristics.

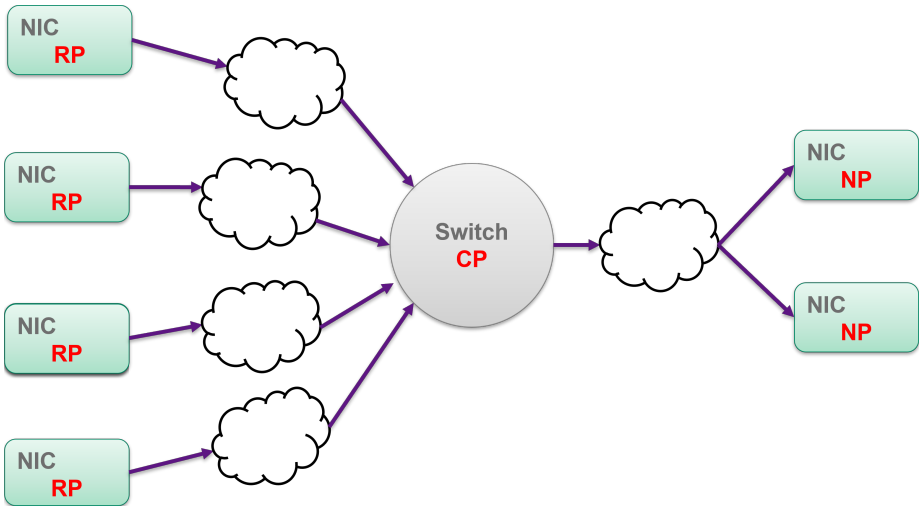
5.6.2.3.5.1 DCQCN CC Algorithm

1. Introduction

The Data Center Quantized Congestion Notification (DCQCN) algorithm operates in a closed-loop system consisting of three primary roles:

- The Congestion Point (CP)
- The Notification Point (NP)
- The Reaction Point (RP)

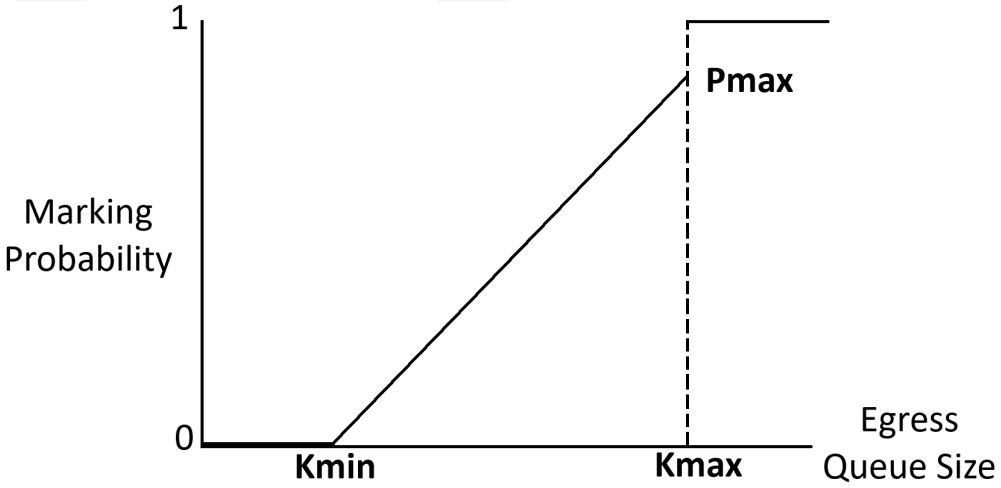
Each NIC can have multiple flows transmitting to a CP, and each flow must react to congestion independently.



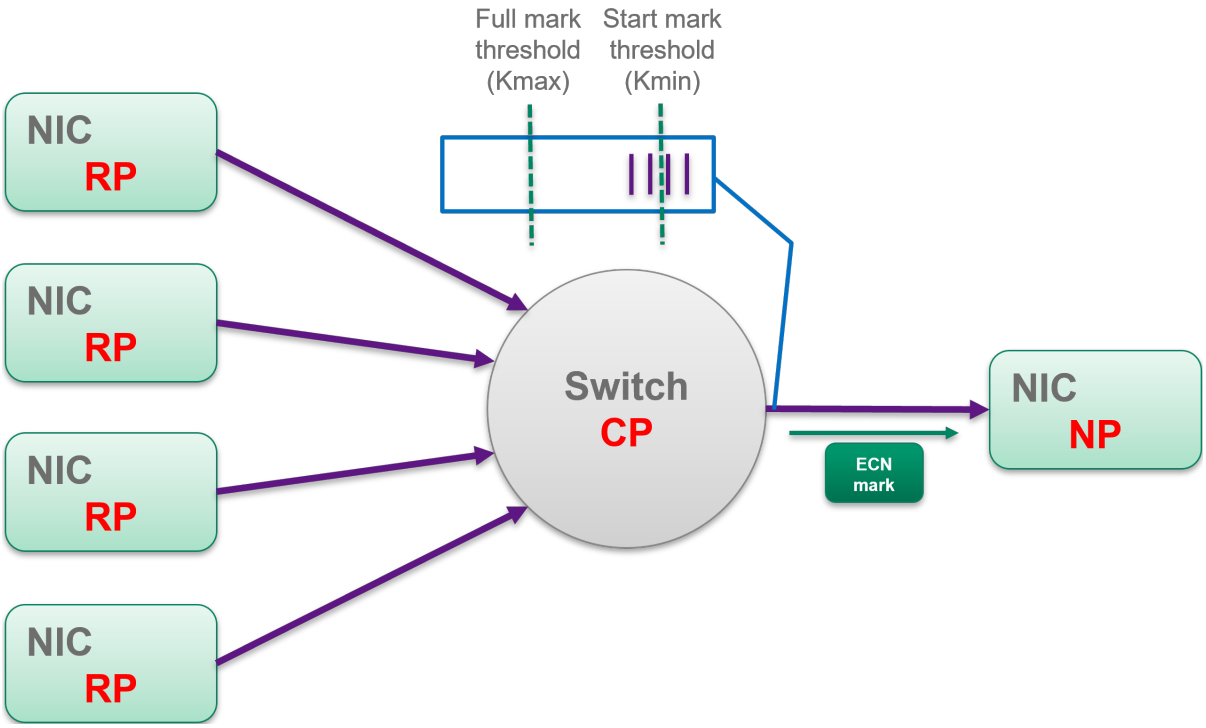
2. Congestion Point – The Switch

The CP is typically the network switch. It monitors its egress queue length and marks passing packets to signal congestion.

- ECN marking – The CP marks the Explicit Congestion Notification (ECN) bits in the packet header based on the egress queue size.
- Marking probability – As the queue size grows past a minimum threshold (**Kmin**), the probability of marking a packet increases linearly until it reaches a maximum probability (**Pmax**) at the full mark threshold (**Kmax**).



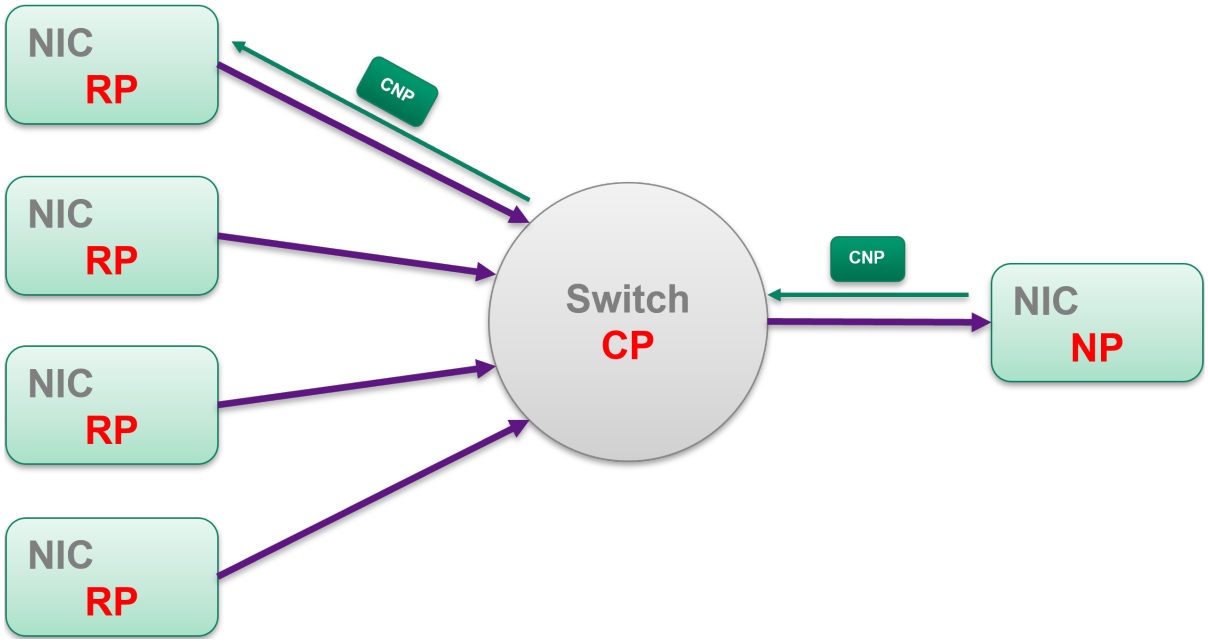
- Compatibility – This is a standard functionality supported by commodity Ethernet switches and is also utilized for DCTCP.



3. Notification Point – The Receiver

The NP is the receiving NIC. Its job is to alert the sender that congestion was encountered along the path.

- CNP generation – When an ECN-marked packet arrives at the NP, a Congestion Notification Packet (CNP) is generated and sent back to the sender.
- Hardware acceleration – This CNP generation is implemented directly in the NIC's hardware, providing a highly fast response.
- Flow identification – The CNP specifically identifies the flow (Queue Pair) experiencing congestion.
- QoS delivery – The CNP can be delivered via a guaranteed low-latency path to ensure the sender is notified as quickly as possible.



4. Reaction Point – The Sender

The RP is the sending NIC. It uses the incoming CNPs (or lack thereof) to dynamically calculate and adjust its transmission rate. The RP manages its rate through three mechanisms: Alpha Update, Rate Decrease, and Rate Increase.

4.1. Alpha (α) Update

Alpha is a weighting factor that determines how aggressively the sending rate should be modified. It is updated every `DCE_TCP_RTT` microseconds.

- Incrementing alpha – If a CNP was received during the update period, alpha is incremented using the following formula
  We don't have a way to export this inline extension.
  .

- Decrementing alpha – If no CNP was received during the update period, alpha is decremented
  We don't have a way to export this inline extension.
  .

- The
  We don't have a way to export this inline extension.
  parameter –
  We don't have a way to export this inline extension.
  controls the aggressiveness of alpha's changes. The lower
  We don't have a way to export this inline extension.
  is, the more aggressive the changes will be.

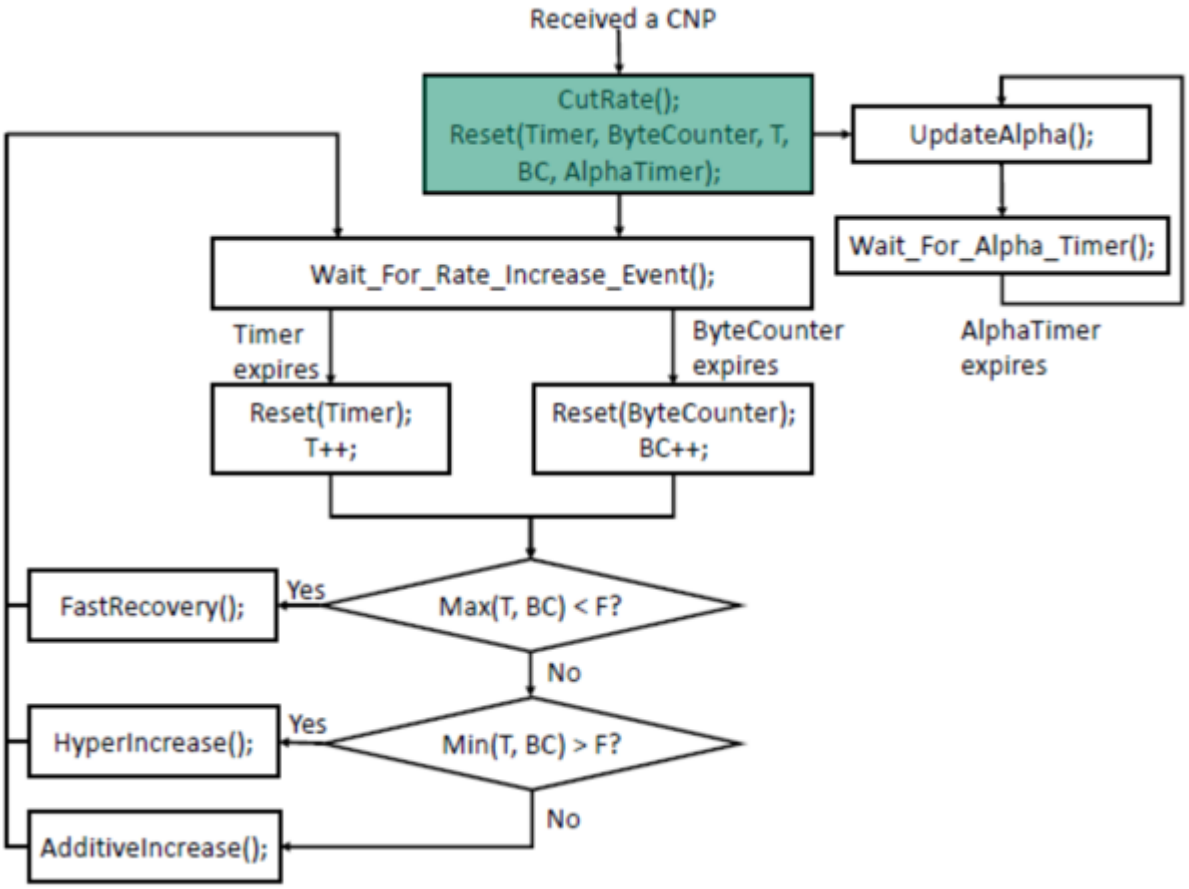
4.2. Rate Decrease Logic

The RP monitors incoming CNPs over a defined window called **RATE_REDUCE_MONITOR_PERIOD** (in microseconds). After this time elapses, if CNPs were detected, the rate is cut.

1. Remember target rate – The algorithm saves the current rate as the target rate recovery point (
   We don't have a way to export this inline extension.
). If **CLAMP_TARGET_RATE** is enabled, the algorithm only remembers this on the *first* decrement.

2. Decrease current rate – The rate is mathematically reduced using the current alpha value and the **RPG_GD** parameter:
   We don't have a way to export this inline extension.

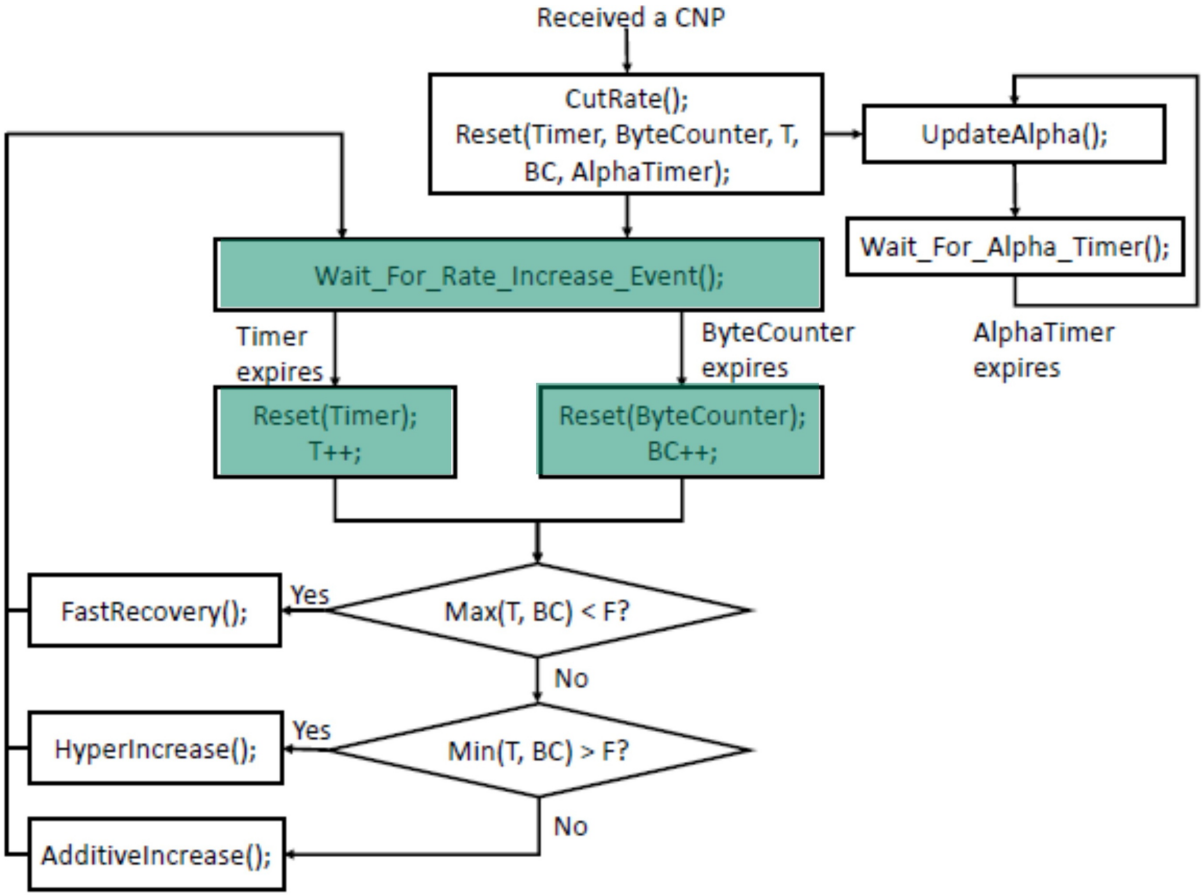
3. Reset counters – After a rate decrease, all increment timers and counters (Time, Byte Counter, etc.) are immediately reset.



4.3. Rate Increase Logic

If no CNPs are received, the RP will systematically increase its transmission rate based on two primary triggers:

- Timer – Rate increments after **TIME_RESET** microseconds.
- Byte counter – Rate increments after **BYTE_RESET** bytes are transmitted.



The rate increase happens across three distinct stages:

4.3.1. Fast Recovery

This is the initial stage of rate recovery.

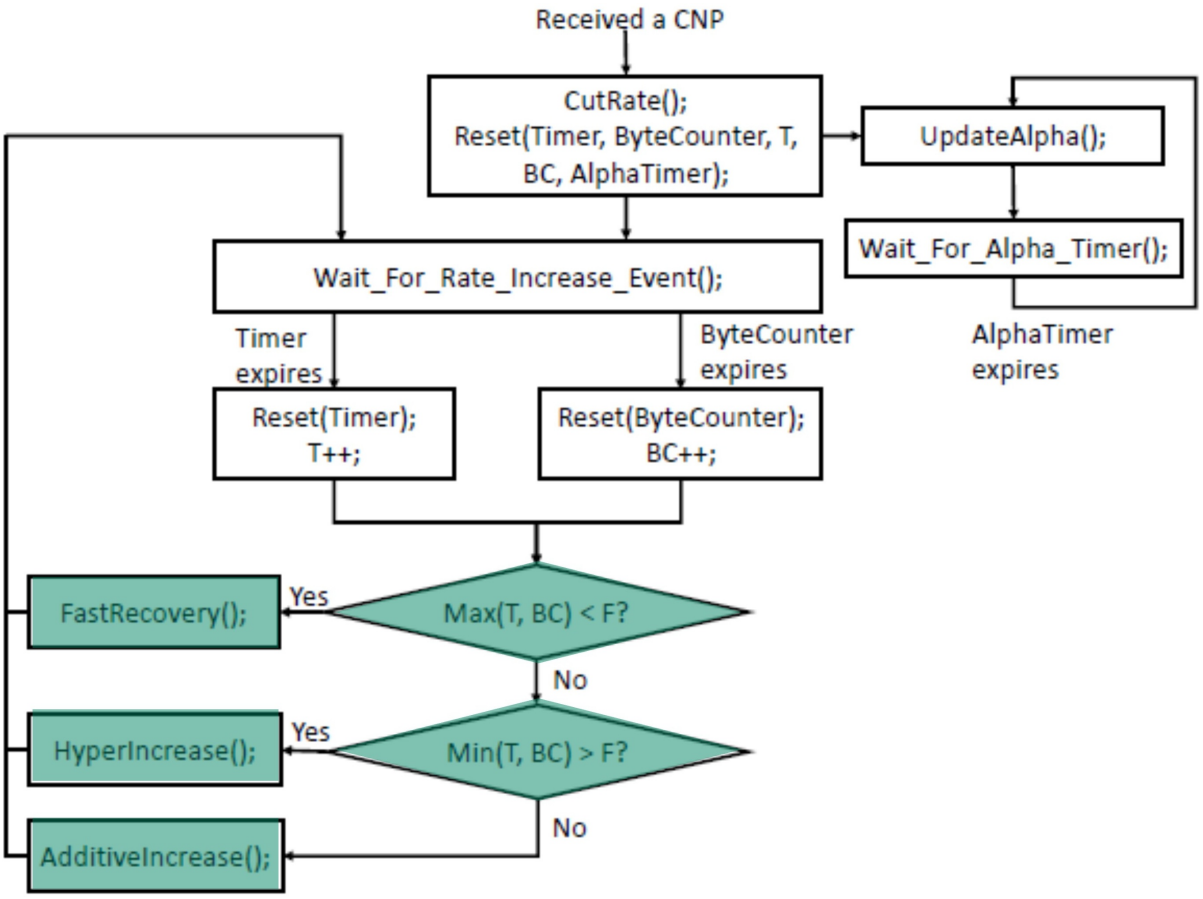
We don't have a way to export this inline extension.

4.3.2. Additive Increase (AI)

This stage begins after `RPG_THRESHOLD` consecutive time resets or `RPG_THRESHOLD` consecutive byte resets.

- The Target Rate is increased by a fixed Additive Increase (AI) value:
  We don't have a way to export this inline extension.

- The Current Rate is then recalculated:
  We don't have a way to export this inline extension.



4.3.3. Hyper Increase (HAI)

This aggressive stage begins only after **RPG_THRESHOLD** consecutive time resets and **RPG_THRESHOLD** consecutive byte resets.

- The Target Rate is increased by a fixed Hyper Additive Increase (HAI) value:
  We don't have a way to export this inline extension.

- The Current Rate is then recalculated:
  We don't have a way to export this inline extension.

5.6.2.3.5.2 DCQCN CC Parameters

This page provides the DCQCN configuration parameters supported by NVIDIA adapters. It details specific settings governing general operation, alpha updates, rate adjustments, and notification points for RoCE congestion control.

The parameters in the following sections are supported by NVIDIA® ConnectX®-4 and later.

1. General

WinOF	MLNX_OFED	NVConfig (Firmware)	Description	Values	Default
DcQcnEnablePrio0 ... 7	Enable	ROCE_CC_PRIO_MASK	Enables RoCE Congestion Control (DCQCN) on a specific priority	• 1 – Enable • 0 – Disable	Enable

2. Alpha Update

WinOF	MLNX_OFED	NVConfig (Firmware)	Description	Unit	Values	Default
DcQcnDceAlphaG	dce_tcp_g	DCE_TCP_G	Every DcQcnDceAlphaUpdatePeriod α is updated. If the CNP is received during this period, alpha is incremented. Otherwise, it is decremented. • Increment formula: We don't have a way to export this inline extension. • Decrement formula: We don't have a way to export this inline extension. • The rate decrement formula is: We don't have a way to export this inline extension.	Fixed point with 10 bits in the fraction part	0...1023	1019
DcQcnDceAlphaUpdatePeriod	dce_tcp_rtt	DCE_TCP_RTT	See above.	µsecs	1...131071	1
DcQcnInitialAlphaValue	initial_alpha_value	INITIAL_ALPHA_VALUE	This parameter sets the initial value of alpha that should be used when receiving the first CNP for a flow.	Fixed point with 10 bits in the fraction part	1...1023	1023

3. Rate Decrease

WinOF	MLNX_OFED	NVConfig (Firmware)	Description	Unit	Values	Default
DcQcnRateToSetOnFirstCnp	rate_to_set_on_first_cnp	RATE_TO_SET_ON_FIRST_CNP	Rates (current, target) on first CNP (0–85% of line rate).	Mb/s	0, 1... line rate [Mb/s]	0
DcQcnMinDecFac	rpg_min_dec_fac	RPG_MIN_DEC_FAC	This parameter defines the maximal ratio of rate decrease in a single event.	%	0...100	50
DcQcnMinRate	rpg_min_rate	RPG_MIN_RATE	This parameter defines the minimal rate limit of the QP.	Mb/s	1...line rate	1
DcQcnAlphaToRateShift	rpg_gd	RPG_GD	The coefficient between α and the rate reduction factor. The rate decrement formula is: We don't have a way to export this inline extension.	Base-2 logarithm represented as a fixed-point number with 10 fractional bits. For example, a mathematical result of 2.0 is stored as 2048 (i.e., 2^11).	10..11	11
DcQcnRateReduceMonitorPeriod	rate_reduce_monitor_period	RATE_REDUCE_MONITOR_PERIOD	The time period between rate reductions.	µsecs	1... UINT32	4

4. Rate Increase

WinOF	MLNX_OFED	NVConfig (Firmware)	Description	Unit	Values	Default
DcQcnClampTgtRate	clamp_tgt_rate	CLAMP_TGT_RATE	If set, every rate decreases. The target rate is updated to the current rate. Otherwise, the target rate is updated to the current rate only on the first decrement after the increment event.	None	• Enable • Disable	Disable
DcQcnTimeReset	rpg_time_reset	RPG_TIME_RESET	The time period between rate increase events.	µsecs	1... 131071	300

WinOF	MLNX_OFED	NVConfig (Firmware)	Description	Unit	Values	Default
DcQcnByteReset	rpg_byte_reset	RPG_BYTE_RESET	The sent bytes counter between rate increase events.	64 bytes	1...32767	32767
DcQcnStageThreshold	rpg_threshold	RPG_THRESHOLD	The threshold of rate increase events for moving to next rate increase phase.	None	1...31	1
DcQcnAiRate	rpg_ai_rate	RPG_AI_RATE	The rate increase value in the Additive Increase phase.	Mb/s	1... line rate	5
DcQcnHaiRate	rpg_hai_rate	RPG_HAI_RATE	The rate increase value in the Hyper Increase phase.	Mb/s	1... line rate	50

5. Notification Point

WinOF	MLNX_OFED	NVConfig (Firmware)	Description	Values	Default
DcQcnCnpDscp	cnp_dscp	CNP_DSCP	This parameter sets the DSCP value for CNPs.	0...63	48
DcQcnCnp802pPrio	cnp_802p_prio	CNP_802P_PRIO	This parameter sets the PCP value for CNPs. Set DcQcnCnpPrioMode to 0 to use this priority value.	0...7	6
DcQcnCnpPrioMode	Not required. This parameter is set automatically if value of cnp_dscp and cnp_802p_prio is set.	N/A	If this parameter is set to 0, then use the configured priority for CNPs. Otherwise, the priority value of CNPs is taken from received data packets.	0/1	0
DcQcnMinTimeBetweenCnps	min_time_between_cnps	MIN_TIME_BETWEEN_CNPS	Minimum time between sending CNPs from the port, in µsecs.	0..4095	4

Refer to this page for more information on RDMA/RoCE Solutions.

5.6.2.3.5.3 ZTR-RTT CC Algorithm

This page introduces NVIDIA Zero Touch RoCE (ZTR) and its hardware-based congestion control algorithm (ZTR-RTT CC), explaining how they enable the seamless, large-scale deployment of high-performance RoCE without requiring specialized switch configurations.

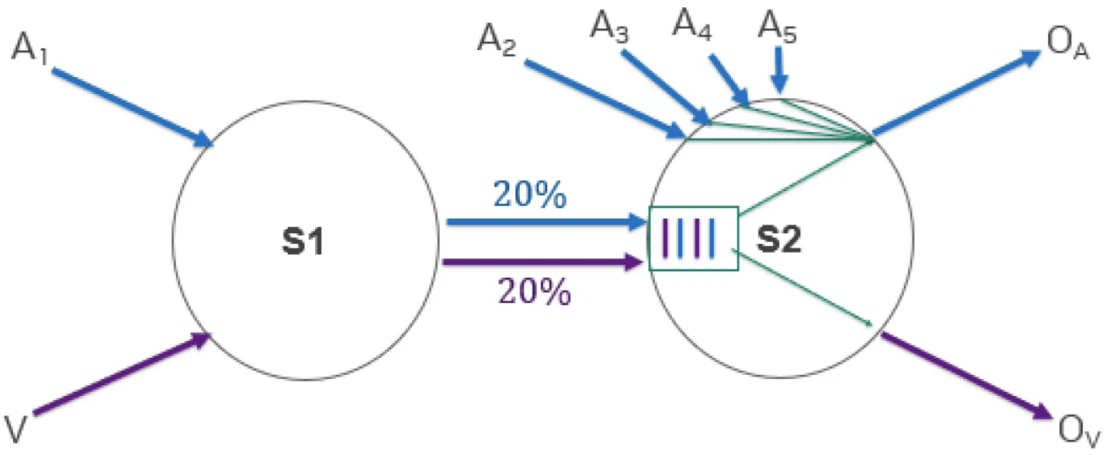
1. Introduction

The ZTR-RTT algorithm is a congestion control (CC) mechanism within the DOCA Programmable Congestion Control (PCC) framework. It relies on measuring network delay alongside traditional explicit notifications. By closely monitoring Round Trip Time (RTT), the algorithm dynamically adjusts transmission rates to maintain high throughput while avoiding queue buildup and packet drops in the network fabric.

2. Congestion Control & Head-of-the-Line Blocking

To understand the necessity of an algorithm like ZTR-RTT, we must look at how congestion spreads in a network, specifically through a phenomenon called Head-of-the-Line (HOL) Blocking.

Consider a scenario with two switches, `S1` and `S2`:



- The shared link – `S1` is forwarding two different traffic flows to `S2`: a blue flow (
  We don't have a way to export this inline extension.
) destined for egress port
  We don't have a way to export this inline extension.
  , and a purple flow (
  We don't have a way to export this inline extension.
) destined for egress port
  We don't have a way to export this inline extension.
  . They share the physical link between `S1` and `S2`.

- The congestion point – At switch `S2`, the blue flow (
  We don't have a way to export this inline extension.
) must compete with several other incoming blue flows (
  We don't have a way to export this inline extension.
  ,

We don't have a way to export this inline extension.
,
We don't have a way to export this inline extension.
,
We don't have a way to export this inline extension.
) that are all trying to exit out of the same port (
We don't have a way to export this inline extension.
). This aggregation overwhelms
We don't have a way to export this inline extension.
, causing blue packets to back up into `S2` 's shared ingress queue.

- The HOL blocking effect – Because that shared queue fills up with congested blue packets, the purple packets (
  We don't have a way to export this inline extension.
) get stuck waiting behind them; even though their destination port (
  We don't have a way to export this inline extension.
) is completely uncongested and ready to transmit.

A highly responsive CC algorithm like ZTR-RTT prevents HOL blocking by detecting the delay on the blue flow and throttling it before it can fill the shared queue, allowing the purple flow to pass through uninterrupted.

3. ZTR-RTT Congestion Indications

To determine the health of the network path and avoid the buffering scenarios described above, the algorithm monitors two primary congestion indicators:

- RTT (Round Trip Time) – Real-time measurement of the delay between sending a packet and receiving its acknowledgment.
- CNP (Congestion Notification Packet) – Explicit notifications sent by the network or receiver when congestion is encountered.

4. Rate Update Scheme

The algorithm continuously evaluates the current congestion state by comparing the measured RTT against the expected delay based on the current transmission rate.

- Congestion evaluation – The congestion state is determined mathematically by comparing the measured RTT to
  We don't have a way to export this inline extension.
  .

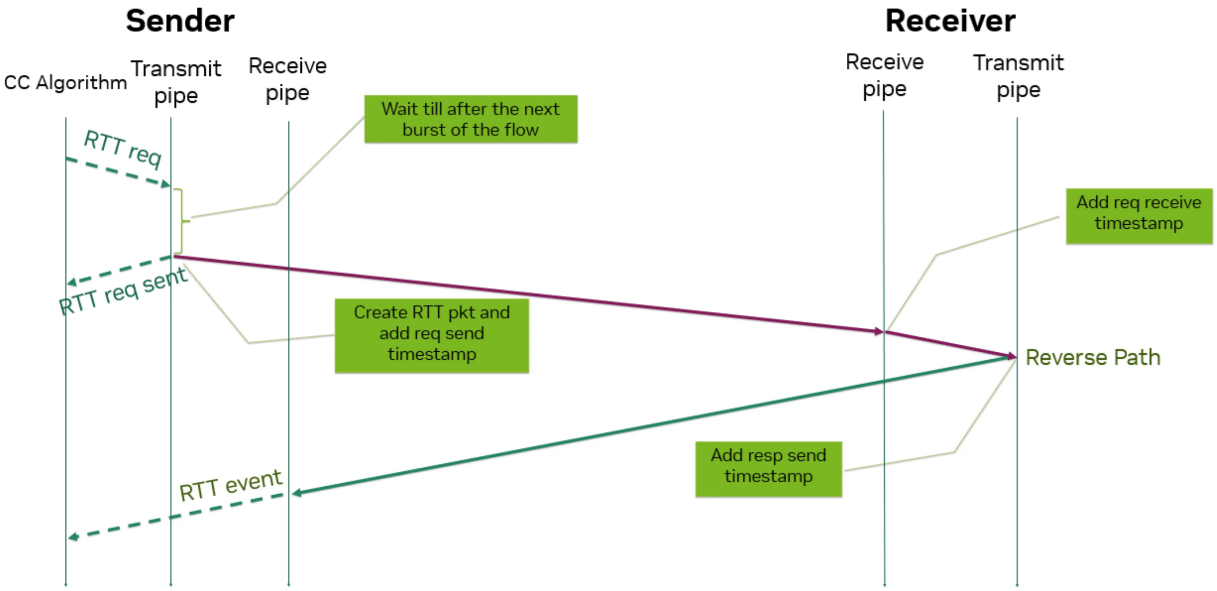
- Adjustment method – Based on this evaluation, the algorithm employs an Additive Increase (gradually adding to the rate when the network is clear) and a Multiplicative Decrease (cutting the rate by a percentage when congestion is detected) approach to maintain an optimal flow.

5. Algorithmic Windowing & Fast Reaction

- Algorithmic windowing – The algorithm mimics traditional window-based CC behavior (similar to standard TCP) by adjusting the transmission rate dynamically whenever the measured RTT changes.
- Fast reaction on first congestion – The algorithm is designed to implement an immediate, fast rate reaction upon the very first indication of network congestion, preventing severe packet drops and maintaining a stable fabric.

6. RTT Measurement Flow

To accurately calculate the RTT without introducing excessive processing overhead, the Sender (Reaction Point) and Receiver (Notification Point) execute a highly synchronized, hardware-assisted timestamping sequence:



1. Initiation (Sender) – The CC algorithm initiates an **RTT req** (Request).
2. Sender Transmit Pipe – The transmit pipe waits until after the next burst of the flow. It then creates the RTT packet and stamps it with a req send timestamp before transmitting it across the network (**RTT req sent**).
3. Receiver Receive Pipe – The packet arrives at the Receiver, where the receive pipe immediately adds a req receive timestamp.
4. Receiver Transmit Pipe (Reverse Path) – As the acknowledgment packet is routed back to the sender, the transmit pipe adds a resp send timestamp.
5. Completion (Sender) – The packet arrives back at the Sender's receive pipe, which extracts all the timestamp data and triggers an RTT event back to the CC algorithm to calculate the final delay and update the rate accordingly.

5.6.2.3.5.4 ZTR-RTT CC Parameter and Counter Configuration

This page guides network administrators and developers in fine-tuning the ZTR-RTT CC algorithm on NVIDIA network adapters. It details the specific configuration variables, hardware counters, and debug tools required for optimization.

1. Introduction

The Zero Touch RoCE Round-Trip Time Congestion Control (ZTR-RTT CC) algorithm utilizes hardware-based feedback loops to proactively manage network congestion. To ensure optimal performance across diverse workloads, the algorithm offers a highly configurable parameter set. This document provides a comprehensive reference for tuning ZTR-RTT CC, detailing its configuration parameters, fixed-point datatypes (such as `fxpN`), and the specific `mlxreg` commands used to apply these tunings. Additionally, it outlines the available performance counters and debugging modes necessary for monitoring congestion states and troubleshooting network flows.

2. ZTR-RTT CC Parameters

2.1. Datatypes

- All parameters are saved as integers in the algorithm. Therefore, integer values need to be used for tuning.
- Some parameters have a different datatype than integer.
  - In the parameters table we summarize the available integer values and the other datatype values.
- Datatype fixed point `N` (`fxpN`):
  - A 32-bit integer represents fixed point number with N bits in the fraction part

32-N bits	**.**	**N bits**

  - To cast between integer representation of `fxpN` to its real value need to divide the integer by
    - For example, 3932 in `fxp16` represents the number

2.2. Parameter Table

Index	Parameter Name	Description	Units	Datatype	Range	Default – RoCE	Recommended Tuning Range
0	`BW_G`	NIC port bandwidth	Gb/s	Integer	100–800	Auto config by device bandwidth	Auto config by device bandwidth

Index	Parameter Name	Description	Units	Datatype	Range	Default – RoCE	Recommended Tuning Range
1	ALPHA	ZTR_RTTCC evaluate the congestion state by comparing the RTT to target RTT which is We don't have a way to export this inline extension. . ALPHA represents the linear connection between target RTT and We don't have a way to export this inline extension. . The formula is: We don't have a way to export this inline extension.	None	fxp16	As integer: 0–2^{16}	6553	As integer: 0–2^{16}
					As fxp16: 0–1	0.1	As fxp16: 0–1
2	MAX_DEC	The maximal multiplicative decrement allowed in one update. The algorithm updates a congestion window (cwnd) in bytes and converges it to rate by the formula: We don't have a way to export this inline extension. For better stability the algorithm limit the rate decrement by MAX_DEC .	None	fxp16	As integer: 1–2^{16}	63570	As integer: 45875–64880
					As fxp16: We don't have a way to export this inline extension. –1	0.97	As fxp16: 0.7–0.99
3	MAX_INC	The maximal multiplicative decrement allowed in one update. The algorithm updates a congestion window (cwnd) in bytes and converges it to rate by the formula: We don't have a way to export this inline extension. For better stability the algorithm limit the rate increment by MAX_INC .	None	fxp16	As integer: 2^{16}–2^{20}	69468	As integer: 2^{16}–2^{18}
					As fxp16: 1–16	1.06	As fxp16: 1–4

Index	Parameter Name	Description	Units	Datatype	Range	Default – RoCE	Recommended Tuning Range
4	`AI`	Additive increase value in Bytes per 100Gb/s. That is, config X will give the value We don't have a way to export this inline extension. .	Bytes per 100Gb/s	Integer	1–5000	9	5–100
5	`HAI`	Hyper additive increase value per 100Gb/s. That is, config X will give the value We don't have a way to export this inline extension. .	Bytes per 100Gb/s	Integer	1–5000	300	200–2000
6	`HAI_PERIOD_NS`	After this period without any decrement, moving from additive increase to hyper additive increase	nsec	Integer	1–`UINT32_MAX`	7000000 (7ms)	100µs–20ms
7	`CONGESTION_DELAY_THRESHOLD`	`ZTR_RTTCC` will react only when the MIN RTT is above this value. The minimal value of `targetRTT` is `CONGESTION_DELAY_THRESHOLD`. See ALPHA.	nsec	Integer	1–`UINT32_MAX`	15000 (15µs)	2µs–15µs
8	`MAX_DELAY`	When RTT is above this value `ZTR_RTTCC` will react more aggressively.	nsec	Integer	1–2^{20}	250000 (250µs)	30µs–1ms
9	`RATE_ON_FIRST_CONGESTION`	At the first time RTT passed `MAX_DELAY`, `ZTR_RTTCC` will set the rate to this value.	None	`fxp20`	As integer: 1–2^{20}	65536	As integer: 10485–2^{20}
					As fxp20: We don't have a way to export this inline extension. –1	0.0625	As fxp20: 0.01–1

Index	Parameter Name	Description	Units	Datatype	Range	Default – RoCE	Recommended Tuning Range
10	`DELAY_ONLY`	Use only RTT as congestion indication	None	Boolean	0,1	0	0
11	`CNP_VLD_RTT`	CNP to validate RTT. When this parameter is set we ignore RTT measurement if there was no CNP during it.	None	Boolean	0,1	0	0,1
12	`TX_DEC`	When set, `ZTR_RTTCC` does not wait for RTT measurement to end. the rate in TX events based on the delay measured until this time.	None	Boolean	0,1	1	0,1
13	`FIXED_RATE`	Available only in debug version of the algorithm. Disable the algorithm rate updates and set a fix rate. When value is 0 the algorithm works as usual.	None	`fxp20`	As integer: $1–2^{23}$	0	0
					As fxp20: We don't have a way to export this inline extension. –8	0	0
14	`FAST_SCHED`	The maximal rate of the NIC scheduler. Can be more than line rate to improve tx pipelining.	None	`fxp20`	As integer: $1–2^{23}$	2097152	As integer: $2^{20}–2^{22}$
					As fxp20: We don't have a way to export this inline extension. –8	2	As fxp20: 1 to 4
15	`TOPOLOGY_AWARE`	The value 1 is available only with `ADVANCED_FEATURES_EN=1`. Evaluate the congestion state by comparing the RTT to the minimum measured RTT.	None	Boolean	0,1	0	0,1

Index	Parameter Name	Description	Units	Data type	Range	Default – RoCE	Recommended Tuning Range
16	ADVANCED_FEATURES_EN	Param to enable advance feature in the algorithm that was not fully tested.	None	Boolean	0,1	0	0,1

2.3. Parameter Tunning Command

```
mlxreg -d <dev> -y --set "cmd_type=8,value=<parameter value>" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=1,algo_param_index=<parameter index>"
```

3. ZTR_RTT CC Counters

Index	Name	Description
0	ZTR_CC_CNP_HANDLE_COUNTER	Number of CNPs handled by the algorithm. Active only if CNP_DEC or CNP_VLD_RTT parameters are set.
1	ZTR_CC_NACK_HANDLE_COUNTER	Number of NACKs handled by the algorithm.
2	ZTR_CC_AI_INC_COUNTER	Number of additive increments.
3	ZTR_CC_HAI_INC_COUNTER	Number of hyper additive increments.
4	ZTR_CC_DEC_COUNTER	Number of decrements.
5	ZTR_CC_HYPER_DEC_COUNTER	Number of hyper decrements.
6	ZTR_CC_TX_DEC_COUNTER	Number of decrements in TX event. Active only if TX_DEC parameter is set.
7	ZTR_CC_MAX_RTT	Maximal RTT measured by the algorithm.
8	ZTR_CC_MIN_RTT	Minimal RTT measured by the algorithm.
9	ZTR_CC_SUM_RTT	Sum of RTTs measured by the algorithm.
10	ZTR_CC_NUM_RTT	Number of RTTs measured by the algorithm. With ZTR_CC_SUM_RTT and ZTR_CC_NUM_RTT, the average RTT can be calculated.
11	ZTR_CC_NOT_VLD_RTT_COUNTER	Number of time RTT measurement was not validate by CNP. Active only when CNP_VLD_RTT parameter is set.
12	ZTR_CC_MAX_RATE	The maximal output rate determined by the algorithm.
13	ZTR_CC_MIN_RATE	The minimal output rate determined by the algorithm.

Index	Name	Description
14	ZTR_CC_EMPTY_SYS_RTT_COUNTER	Number of times the algorithm detected global minimum RTT.
15	ZTR_CC_RTT_TIMEOUT_COUNTER	Number of RTT probe timeouts.

4. Debug Tools

4.1. Debug Mode

ZTR RTTCC algorithm has two working modes:

- Default in **algo_slot 0**.
- Debug in **algo_slot 1**.

Counters are available only in debug mode.

Moving to debug mode requires the following steps:

1. Disable deployment mode in **algo_slot 0**.
2. Enable debug mode in **alg_slot 1** and enable counters.

Example:

- Disable deployment mode:

```
sudo mlxreg -d /dev/mst/mt4129_pciconf0 -y --set "cmd_type=2" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=0,algo_param_index=0"
```

- Enable debug mode and counters:

```
sudo mlxreg -d /dev/mst/mt4129_pciconf0 -y --set "cmd_type=1,counter_en=1" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=1,algo_param_index=0"
```

4.2. Query Counters

Example:

- Reset counters:

```
sudo mlxreg -d /dev/mst/mt4129_pciconf0  -y --set "cmd_type=13" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=1,algo_param_index=0"
```

- Query counters:

```
 mlxreg -d /dev/mst/mt4129_pciconf0 -y --set "cmd_type=12" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=1,algo_param_index=0"
```

In the output

We don't have a way to export this inline extension.
We don't have a way to export this inline extension.
will indicate the value of counter number .

5.6.2.3.5.5 Enhanced Connection Establishment

Introduction

Enhanced Connection Establishment (ECE) extends the IBTA-defined RDMA Connection Manager (RDMA-CM) layer. Introduced in IBTA version 1.4, ECE allows two Host Channel Adapters (HCAs) to negotiate optional transport features when establishing a queue pair (QP), on a per-flow basis. Because ECE operates at the connection-manager layer rather than the InfiniBand transport layer, it applies to both InfiniBand and RoCE fabrics.

ECE is supported on NVIDIA ConnectX-6 Dx and later devices.

ECE solves a fundamental problem in heterogeneous RDMA deployments. Features such as Selective Repeat and Congestion Control require both endpoints to agree before they are activated. Enabling a feature on only one endpoint, without the peer's knowledge, can cause packet drops, delays, or performance degradation. ECE provides a bilateral handshake through which the endpoints advertise and agree on supported features for each QP.

Negotiated Features

ECE supports negotiation of the following features:

Feature	Abbreviation	Result When Negotiated
Selective Repeat	SR	Retransmits only missing packets instead of using go-back-N retransmission.
Congestion Control	CC	Negotiates the congestion-control algorithms enabled on the QP, which can include ZTR-RTT CC, DCQCN, or both.

ECE activates only the capabilities that both endpoints advertise. If one endpoint does not support a feature, ECE clears that feature from the agreed options and establishes the connection without it.

ECE Wire Format

Within RDMA-CM, ECE information is carried in the following connection-management messages:

- **REQ**
- **REP**
- **SIDR_REQ**
- **SIDR_REP**

The messages use the following fields:

- **VendorID**: Contains the IEEE organizationally unique identifier (OUI) for the vendor capability set.
- **AttributeModifier**: Contains the vendor-specific options supported by the endpoint.

At the ibverbs API level, these fields map to the **ibv_ece** structure defined in **<infiniband/verbs.h>**:

```
struct ibv_ece {
    uint32_t vendor_id;  /* IEEE OUI. NVIDIA = 0x15b3 */
    uint32_t options;    /* 4 MSB = ECE version; 28 LSB = feature bits */
    uint32_t comp_mask;
```

```
};
```

In the **options** field:

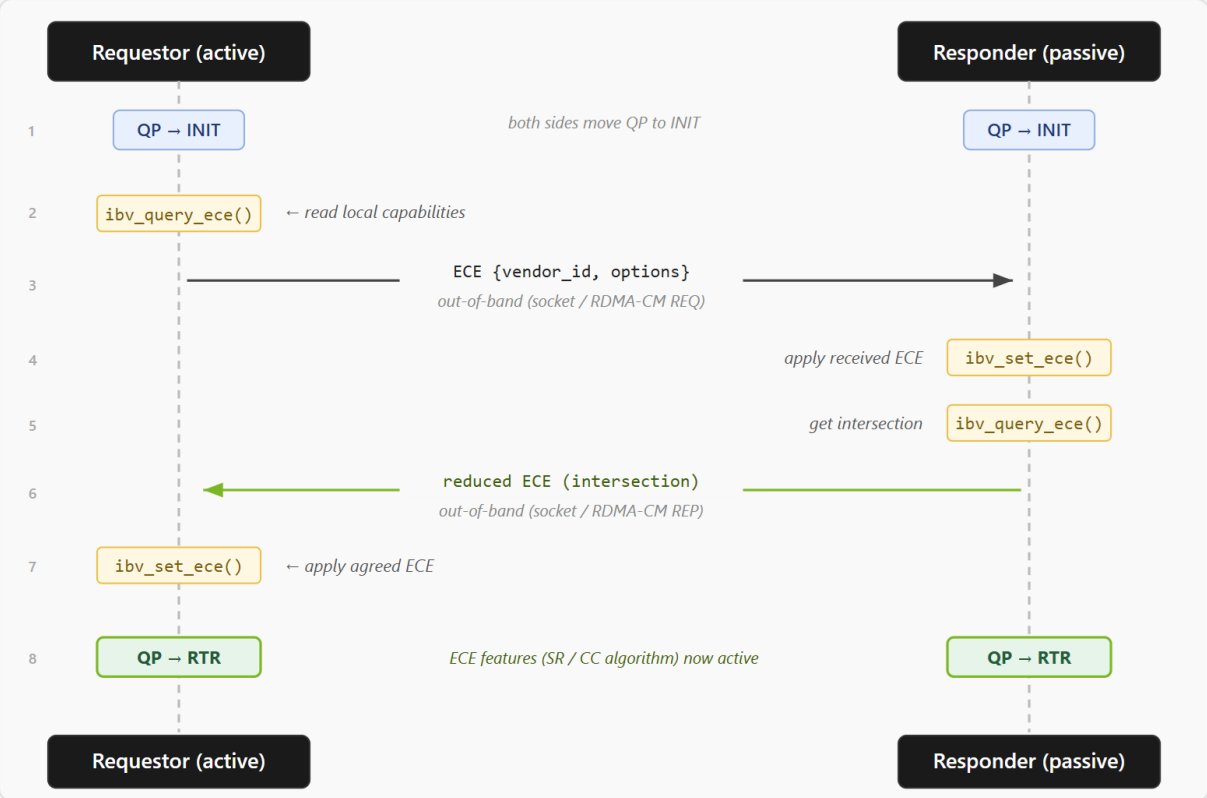
- The four most significant bits identify the ECE version.
- The remaining 28 bits represent negotiable capabilities as individual flags.

The version field allows ECE to support future extensions without breaking backward compatibility.

Connection Establishment Flow (RC QP)

For a reliable-connected QP, ECE negotiation occurs around the transition from **INIT** to **RTR**.

Both endpoints must call **ibv_set_ece()** before calling **ibv_modify_qp()** to transition the QP to **RTR**.



For applications that manage QP connection setup directly through ibverbs, the application must exchange the ECE values between endpoints through an out-of-band channel, such as a socket or shared memory.

When an application uses RDMA-CM, the CM stack carries the ECE values in its connection-management messages.

Features Negotiated via ECE

Selective Repeat

By default, InfiniBand transport uses go-back-N retransmission. When a packet is lost, the sender retransmits that packet and all subsequent packets, including packets that the receiver has already received.

Selective Repeat changes this behavior so that the sender retransmits only the missing packets. This reduces retransmission overhead in environments where packet loss occurs intermittently.

Selective Repeat can be enabled in either of the following ways:

- ECE negotiation – Enables Selective Repeat per QP after both endpoints advertise and agree on the capability. This approach is recommended for deployments containing different NIC generations.
- Device-wide configuration – Enables Selective Repeat for all QPs on the device by setting `selective_repeat_forced_en` in the `ROCE_ACCL` register. This method bypasses ECE negotiation.

Congestion Control

The CC feature bits in the ECE options negotiate which congestion-control algorithms are enabled on the QP, not merely whether congestion control is enabled. The negotiated result can include ZTR-RTT CC, DCQCN, or both.

Both endpoints must agree on each algorithm. An algorithm that is not supported by both endpoints is not activated for the QP.

NVIDIA NICs support the following built-in congestion-control algorithms:

- ZTR-RTT CC – Uses hardware-timestamped round-trip-time measurements and does not require specialized switch configuration. Both endpoints must negotiate ZTR-RTT CC before either endpoint begins sending RTT probe packets.
- DCQCN – Uses Explicit Congestion Notification (ECN) marking from ECN-capable switches.

For configuration information, see:

- ZTR-RTT CC Algorithm
- ZTR-RTT CC Parameter and Counter Configuration
- DCQCN CC Algorithm
- DCQCN CC Parameters

> Note: An incorrect ECE call order in NCCL versions earlier than 2.24.x could cause one endpoint to activate ZTR-RTT CC while the other endpoint did not. This asymmetric state caused one endpoint to send RTT probes that the peer did not handle and resulted in a reported 20–22% performance degradation in H100 all-to-all workloads.

Integrating ECE

API Summary

Function	Caller	Call Point	Purpose
`ibv_query_ece(qp, ece, supported)`	Requestor	After transitioning the QP to `INIT`	Retrieves the local ECE capabilities.

Function	Caller	Call Point	Purpose
`ibv_set_ece(qp, ece, supported)`	Both endpoints	Before transitioning the QP to `RTR`	Applies the received or reduced ECE options.
`ibv_query_ece(qp, ece, supported)`	Responder	After calling `ibv_set_ece()`	Retrieves the reduced set of mutually supported ECE options.

The `supported` output parameter indicates whether the underlying driver supports ECE.

When `supported` is `0`, the QP does not support ECE. The application can continue establishing the connection without activating ECE-negotiated features, and the feature set defaults to the device configuration.

Raw ibverbs Integration

Applications that manage QP connection setup directly must exchange ECE values through the same side channel used to exchange information such as the local identifier, QP number, and packet sequence number.

The following example shows the ECE negotiation sequence using an out-of-band socket.

Requestor

```
#include <infiniband/verbs.h>

/* Helper: send/receive ECE over an established socket fd */
static int send_ece(int fd, struct ibv_ece *ece) {
    uint32_t buf[2] = { htonl(ece->vendor_id), htonl(ece->options) };
    return write(fd, buf, sizeof(buf)) == sizeof(buf) ? 0 : -1;
}

static int recv_ece(int fd, struct ibv_ece *ece) {
    uint32_t buf[2];
    if (read(fd, buf, sizeof(buf)) != sizeof(buf))
        return -1;

    ece->vendor_id = ntohl(buf[0]);
    ece->options = ntohl(buf[1]);
    return 0;
}

int requestor_connect(struct ibv_qp *qp, int sockfd)
{
    struct ibv_ece local_ece = {};
    struct ibv_ece remote_ece = {};
    int supported = 0;
    int ret;

    /*
     * Step 1: After the QP is in INIT, retrieve the local ECE
     * capabilities.
     */
    ret = ibv_query_ece(qp, &local_ece, &supported);
    if (ret || !supported) {
        /* ECE is not supported. Continue without ECE negotiation. */
        goto move_to_rtr;
    }

    /*
     * Step 2: Send the local ECE capabilities to the responder through
     * the out-of-band socket.
     */
    if (send_ece(sockfd, &local_ece) < 0)
        return -1;

    /*
     * Step 3: Receive the reduced set of mutually supported ECE options
     * from the responder.
     */
    if (recv_ece(sockfd, &remote_ece) < 0)
        return -1;

    /*
     * Step 4: Apply the agreed ECE options before transitioning the QP
     * to RTR.
     */
    ret = ibv_set_ece(qp, &remote_ece, &supported);
    if (ret) {
        fprintf(stderr, "ibv_set_ece failed: %s
", strerror(ret));
```

```
        return -1;
    }

move_to_rtr:
    /* Step 5: Transition the QP to RTR. */
    struct ibv_qp_attr attr = {
        .qp_state = IBV_QPS_RTR,
        .path_mtu = IBV_MTU_4096,
        .dest_qp_num = remote_qpn, /* Received from the peer */
        .rq_psn = remote_psn,
        .max_dest_rd_atomic = 1,
        .min_rnr_timer = 12,
        /* Populate ah_attr from the remote LID or GID. */
    };

    return ibv_modify_qp(
        qp,
        &attr,
        IBV_QP_STATE |
        IBV_QP_AV |
        IBV_QP_PATH_MTU |
        IBV_QP_DEST_QPN |
        IBV_QP_RQ_PSN |
        IBV_QP_MAX_DEST_RD_ATOMIC |
        IBV_QP_MIN_RNR_TIMER);
}
```

Responder

```
int responder_accept(struct ibv_qp *qp, int sockfd)
{
    struct ibv_ece remote_ece = {};
    struct ibv_ece reduced_ece = {};
    int supported = 0;
    int ret;

    /* Step 1: Receive the requestor's ECE capabilities. */
    if (recv_ece(sockfd, &remote_ece) < 0)
        return -1;

    /*
     * Step 2: Apply the received ECE options. The driver calculates
     * the intersection with the responder's locally supported options.
     */
    ret = ibv_set_ece(qp, &remote_ece, &supported);
    if (ret || !supported) {
        /*
         * ECE is not supported on this endpoint. Return an empty ECE
         * result and continue without ECE-negotiated features.
         */
        memset(&reduced_ece, 0, sizeof(reduced_ece));
        goto send_reduced;
    }

    /*
     * Step 3: Retrieve the reduced set of mutually supported ECE
     * options.
     */
    ret = ibv_query_ece(qp, &reduced_ece, &supported);
    if (ret) {
        fprintf(stderr, "ibv_query_ece failed: %s
", strerror(ret));
        return -1;
    }

send_reduced:
    /* Step 4: Return the reduced ECE options to the requestor. */
    if (send_ece(sockfd, &reduced_ece) < 0)
        return -1;

    /*
     * Step 5: Transition the QP to RTR with the agreed features
     * active.
     */
    struct ibv_qp_attr attr = {
        .qp_state = IBV_QPS_RTR,
        /* Populate the remaining fields from the requestor's QP data. */
    };

    return ibv_modify_qp(
        qp,
        &attr,
        IBV_QP_STATE |
        IBV_QP_AV |
        IBV_QP_PATH_MTU |
        IBV_QP_DEST_QPN |
        IBV_QP_RQ_PSN |
        IBV_QP_MAX_DEST_RD_ATOMIC |
        IBV_QP_MIN_RNR_TIMER);
}
```

NCCL Integration

NCCL wraps these functions through `wrap_ibv_query_ece()` and `wrap_ibv_set_ece()` in `src/include/ibvwrap.h`. The wrappers add a `supported` check and NCCL-specific error handling, but use the same negotiation sequence.

RDMA-CM Integration

When an application uses RDMA-CM, the CM stack manages the `REQ` and `REP` messages.

Set ECE on the CM-managed QP before calling `rdma_connect()` or `rdma_accept()`. RDMA-CM then carries the ECE options as part of its internal InfiniBand connection-management handshake.

Client

```
#include <rdma/rdma_cma.h>
#include <infiniband/verbs.h>

int client_connect(
    struct rdma_cm_id *id,
    struct rdma_conn_param *conn_param)
{
    struct ibv_ece ece = {};
    int supported = 0;

    /*
     * The QP must already exist. Call rdma_create_qp() before this
     * function.
     */

    /*
     * Step 1: Retrieve the local ECE capabilities from the CM-managed
     * QP.
     */
    if (ibv_query_ece(id->qp, &ece, &supported) == 0 && supported) {
        /*
         * Optionally restrict the options to the features that the
         * application wants to activate.
         *
         * Leaving the options unchanged advertises all locally
         * supported features.
         */
    }

    /*
     * Step 2: rdma_connect() sends the IB REQ. The CM includes the QP's
     * current ECE options in the request.
     */
    return rdma_connect(id, conn_param);
}
```

Server

Handle ECE negotiation after receiving the `CONNECT_REQUEST` event and before calling `rdma_accept()`.

```
int server_on_connect_request(struct rdma_cm_event *event)
{
    struct rdma_cm_id *child_id = event->id;
    struct ibv_ece remote_ece = {};
    struct ibv_ece reduced_ece = {};
    int supported = 0;

    /* Create a QP on the child CM ID before negotiating ECE. */
    struct ibv_qp_init_attr qp_attr = {
        /* Populate the QP initialization attributes. */
    };

    if (rdma_create_qp(child_id, pd, &qp_attr))
        return -1;

    /*
     * Step 1: The CM exposes the requestor's ECE through ibv_query_ece()
     * on the child QP after the QP is created and before rdma_accept().
     * Apply it with ibv_set_ece() to calculate the intersection.
     */
    if (ibv_query_ece(
            child_id->qp,
            &remote_ece,
            &supported) == 0 && supported) {
        ibv_set_ece(child_id->qp, &remote_ece, &supported);

        /*
```

```
            * Retrieve the reduced set of mutually supported ECE options.
            */
           ibv_query_ece(
               child_id->qp,
               &reduced_ece,
               &supported);
       }

       /*
        * Step 2: rdma_accept() sends the IB REP containing the reduced
        * ECE options.
        */
       struct rdma_conn_param accept_param = {
           /* Populate the connection parameters. */
       };

       return rdma_accept(child_id, &accept_param);
   }
```

When the client receives the **REP**, RDMA-CM applies the reduced ECE options to the client QP automatically. The client does not need to call **ibv_set_ece()** explicitly after receiving the response.

Backward Compatibility

ECE is designed to allow connections to proceed when the endpoints support different capabilities.

The following fallback behavior applies:

- Peer without ECE support – If **ibv_query_ece()** sets **supported** to **0**, or the remote endpoint sends a **vendor_id** value of **0**, the connection proceeds without ECE-negotiated features, and the feature set defaults to the device configuration.
- Partial feature mismatch: ECE clears any feature that one endpoint does not support. The connection proceeds with only the capabilities supported by both endpoints.
- ECE mismatch – NVIDIA NICs do not send a **REJ** response because of an ECE mismatch. The connection falls back to the mutually supported feature set. This is graceful degradation, not connection failure.

Platform Support

ECE is supported on NVIDIA ConnectX-6 Dx and later devices.

Firmware and InfiniBand device-verbs support for ECE predates adoption by many applications. An application that does not call **ibv_query_ece()** and **ibv_set_ece()** does not activate ECE-gated features on its QPs.

Version History

RDMA-CM and Linux Kernel

Version	Date	Change
Linux 5.8	August 2020	Introduced ECE through a nine-patch series for **RDMA/cm**, **RDMA/ucma**, and **RDMA/cma**. Added ECE information to **REQ**, **REP**, **SIDR_REQ**, and **SIDR_REP** messages, and extended **ucma_connect** and **rdma_accept** to carry ECE parameters.
Linux 5.8	August 2020	Added mlx5 driver support through **RDMA/mlx5: Set ECE options during modify QP**. Added the **ece_options** field to **mlx5_ib_modify_qp**, propagated ECE through the **INIT2RTR**, **RTR2RTS**, and **RTS2RTS** firmware commands, and added the **MLX5_IB_ALLOC_UCONTEXT_RESP_MASK_ECE** capability flag.

Version	Date	Change
After Linux 5.8	2020–2021	Restored Dynamic Connected QP support through `RDMA/mlx5: Return ECE DC support`. ECE options established by the first connection are inherited by later connections to the same many-to-one DC QP.

rdma-core

Version	Date	Change
rdma-core 30	Approximately 2020	Added `ibv_query_ece()` and `ibv_set_ece()` to libibverbs. Added `rdma_set_local_ece()` and `rdma_get_remote_ece()` to librdmacm.

NCCL

Version	Change
2.19.4	Added RoCE ECE support through `NCCL_IB_ECE_ENABLE`. ECE was enabled by default. An incorrect `ibv_set_ece()` call order caused ZTR-RTT CC negotiation to become unsynchronized between the requestor and responder.
2.23.4	Added `NCCL_IB_ECE_ENABLE=0` as a mechanism for disabling ECE. The call-order issue remained.
2.24.x	Corrected the ECE verbs call order so that the requestor and responder synchronize ZTR-RTT CC activation.

NCCL Compatibility Issue

With the incorrect call order, one endpoint could activate ZTR-RTT CC while the other endpoint did not. This produced RTT probe packets with opcode `0x64`, `UD SEND ONLY`, and destination QP `0x10`. The issue caused a reported 20–22% performance degradation in H100 all-to-all workloads and was root-caused in March 2025.

Related Documentation

- ZTR-RTT CC Algorithm
- ZTR-RTT CC Parameter and Counter Configuration
- DCQCN CC Algorithm
- DCQCN CC Parameters

5.6.2.4 DPU

This section contains the following pages:

- QoS Configuration
- Shared RQ Mode

5.6.2.4.1 QoS Configuration

To learn more about port QoS configuration, refer to this community post.

When working in Embedded Host mode, using `mlnx_qos` on both the host and Arm will result with undefined behavior. Users must only use `mlnx_qos` from the Arm. After changing the QoS settings from Arm, users must restart the mlx5 driver on host.

When configuring QoS using DCBX, the `lldpad` service from the NVIDIA® BlueField® networking platform's (DPU or SuperNIC) side must be disabled if the configurations are not done using tools other than `lldpad`.

This section explains how to configure QoS group and settings using devlink located under `/opt/mellanox/iproute2/sbin/`. It is applicable to host PF/VF and Arm side SFs. The following uses VF as example.

The settings of a QoS group include creating/deleting a QoS group and modifying its `tx_max` and `tx_share` values. The settings of VF QoS include modifying its `tx_max` and `tx_share` values, assigning a VF to a QoS group, and unassigning a VF from a QoS group. This section focuses on the configuration syntax.

Please refer to section "Limit and Bandwidth Share Per VF" in the MLNX_OFED User Manual for detailed explanation on vPort QoS behaviors.

5.6.2.4.1.1 devlink port function rate add

	`devlink port function rate add <DEV>/<GROUP_NAME>` Adds a QoS group.	
Syntax Description	DEV/GROUP_NAME	Specifies group name in string format
Example	This command adds a new QoS group named `12_group` under device `pci/0000:03:00.0`: `devlink port function rate add pci/0000:03:00.0/12_group`	
Notes		

5.6.2.4.1.2 devlink port function rate del

	`devlink port function rate del <DEV>/<GROUP_NAME>` Deletes a QoS group.	
Syntax Description	DEV/GROUP_NAME	Specifies group name in string format

Example	This command deletes QoS group **12_group** from device **pci/0000:03:00.0**: `devlink port function rate del pci/0000:03:00.0/12_group`
Notes	

5.6.2.4.1.3 devlink port function rate set tx_max tx_share

	devlink port function rate set {<DEV>/<GROUP_NAME> | <DEV>/<PORT_INDEX>} tx_max <TX_MAX> [tx_share <TX_SHARE>] Sets **tx_max** and **tx_share** for QoS group or devlink port.	
Syntax Description	DEV/GROUP_NAME	Specifies the group name to operate on
	DEV/PORT_INDEX	Specifies the devlink port to operate on
	TX_MAX	**tx_max** bandwidth in MB/s
	TX_SHARE	tx_share bandwidth in MB/s

Example	This command sets **tx_max** to 2000MB/s and **tx_share** to 500MB/s for the **12_group** QoS group:
	`devlink port function rate set pci/0000:03:00.0/12_group tx_max 2000MBps tx_share 500MBps`
	This command sets **tx_max** to 2000MB/s and **tx_share** to 500MB/s for the VF represented by port index 196609:
	`devlink port function rate set pci/0000:03:00.0/196609 tx_max 200MBps tx_share 50MBps`
	This command displays a mapping between VF devlink ports and netdev names:
	`$ devlink port`
	In the output of this command, VFs are indicated by **flavour pcivf**.	

Notes		

5.6.2.4.1.4 devlink port function rate set parent

	devlink port function rate set <DEV>/<PORT_INDEX> {parent <PARENT_GROUP_NAME>} Assigns devlink port to a QoS group.	
Syntax Description	DEV/PORT_INDEX	Specifies the devlink port to operate on
	PARENT_GROUP_NAME	parent group name in string format
Example	This command assigns this function to the QoS group **12_group**: `devlink port function rate set pci/0000:03:00.0/196609 parent 12_group`	
Notes		

5.6.2.4.1.5 devlink port function rate set noparent

	devlink port function rate set <DEV>/<PORT_INDEX> noparent Ungroups a devlink port.	
Syntax Description	DEV/PORT_INDEX	Specifies the devlink port to operate on
Example	This command ungroups this function: `devlink port function rate set pci/0000:03:00.0/196609 noparent`	
Notes		

5.6.2.4.1.6 devlink port function rate show

	devlink port function rate show [<DEV>/<GROUP_NAME> | <DEV>/<PORT_INDEX>] Displays QoS information QoS group or devlink port.	
Syntax Description	DEV/GROUP_NAME	Specifies the group name to display
	DEV/PORT_INDEX	Specifies the devlink port to display

Example	This command displays the QoS info of all QoS groups and devlink ports on the system:
	`devlink port function rate show`
	`pci/0000:03:00.0/12_group type node tx_max 2000MBps tx_share 500MBps`
	`pci/0000:03:00.0/196609 type leaf tx_max 200MBps tx_share 50MBps parent 12_group`
	This command displays QoS info of **12_group**:
	`devlink port function rate show pci/0000:03:00.0/12_group`
	`pci/0000:03:00.0/12_group type node tx_max 2000MBps tx_share 500MBps`	

Notes	If a QoS group name or devlink port are not specified, all QoS groups and devlink ports are displayed.	

5.6.2.4.2 Shared RQ Mode

When creating 1 send queue (SQ) and 1 receive queue (RQ), each representor consumes ~3MB memory per single channel. Scaling this to the desired 1024 representors (SFs and/or VFs) would require ~3GB worth of memory for single channel. A major chunk of the 3MB is contributed by RQ allocation (receive buffers and SKBs). Therefore, to make efficient use of memory, shared RQ mode is implemented so PF/VF/SF representors share receive queues owned by the uplink representor.

The feature is enabled by default. To disable it:

1. Edit the field **ALLOW_SHARED_RQ** in **/etc/mellanox/mlnx-bf.conf** as follows:

   ```
   ALLOW_SHARED_RQ="no"
   ```

2. Restart the driver. Run:

```
/etc/init.d/openibd restart
```

To connect from the host to NVIDIA® BlueField® networking platform (DPU or SuperNIC) in shared RQ mode, please refer to section Verifying Connection from Host to BlueField.

⚠ PF/VF representor to PF/VF communication on the host is not possible.

The following behavior is observed in shared RQ mode:

- It is expected to see a **0** in the **rx_bytes** and **rx_packets** and valid **vport_rx_packets** and **vport_rx_bytes** after running traffic. Example output:

```
# ethtool -S pf0hpf
NIC statistics:
     rx_packets: 0
     rx_bytes: 0
     tx_packets: 66946
     tx_bytes: 8786869
     vport_rx_packets: 546093
     vport_rx_bytes: 321100036
     vport_tx_packets: 549449
     vport_tx_bytes: 321679548
```

- Ethtool usage – in this mode, it is not possible to change/set the ring or coalesce parameters for the RX side using ethtool. Changing channels also only affects the TX side.

5.6.2.5 Storage Protocols

There are several storage protocols that use the advantage of InfiniBand and RDMA for performance reasons (high throughput, low latency and low CPU utilization). In this chapter we will discuss the following protocols:

- SCSI RDMA Protocol (SRP) is designed to take full advantage of the protocol off-load and RDMA features provided by the InfiniBand architecture.
- iSCSI Extensions for RDMA (iSER) is an extension of the data transfer model of iSCSI, a storage networking standard for TCP/IP. It uses the iSCSI components while taking the advantage of the RDMA protocol suite. ISER is implemented on various stor- age targets such as TGT, LIO, SCST and out of scope of this manual.
  For various ISER targets configuration steps, troubleshooting and debugging, as well as other implementation of storage protocols over RDMA (such as Ceph over RDMA, nbdX and more) refer to Storage Solutions on the Community website.
- Lustre is an open-source, parallel distributed file system, generally used for large-scale cluster computing that supports many requirements of leadership class HPC simulation environments.
- NVM Express™ over Fabrics (NVME-oF)
  - NVME-oF is a technology specification for networking storage designed to enable NVMe message-based commands to transfer data between a host computer and a target solid-state storage device or system over a network such as Ethernet, Fibre Channel, and InfiniBand. Tunneling NVMe commands through an RDMA fabric provides a high throughput and a low latency. This is an alternative to the SCSi based storage networking protocols.
  - NVME-oF Target Offload is an implementation of the new NVME-oF standard Target (server) side in hardware. Starting from ConnectX-5 family cards, all regular IO

requests can be processed by the HCA, with the HCA sending IO requests directly to a real NVMe PCI device, using peer-to-peer PCI communications. This means that excluding connection management and error flows, no CPU utilization will be observed during NVME-oF traffic.
For further information, please refer to Storage Solutions on the Community website (enterprise-support.nvidia.com/s/).

5.6.2.5.1 SRP - SCSI RDMA Protocol

The SCSI RDMA Protocol (SRP) is designed to take full advantage of the protocol offload and RDMA features provided by the InfiniBand architecture. SRP allows a large body of SCSI software to be readily used on InfiniBand architecture. The SRP Initiator controls the connection to an SRP Target in order to provide access to remote storage devices across an InfiniBand fabric. The kSRP Target resides in an IO unit and provides storage services.

5.6.2.5.1.1 SRP Initiator

This SRP Initiator is based on open source from OpenFabrics (www.openfabrics.org) that implements the SCSI RDMA Protocol-2 (SRP-2). SRP-2 is described in Document # T10/1524-D available from http://www.t10.org.

The SRP Initiator supports

- Basic SCSI Primary Commands -3 (SPC-3) (www.t10.org/ftp/t10/drafts/spc3/spc3r21b.pdf)
- Basic SCSI Block Commands -2 (SBC-2) (www.t10.org/ftp/t10/drafts/sbc2/sbc2r16.pdf)
- Basic functionality, task management and limited error handling

⚠ This package, however, does not include an SRP Target.

Loading SRP Initiator

To load the SRP module either:

- Execute the **modprobe ib_srp** command after the OFED driver is up.

or

1. Change the value of **SRP_LOAD** in **/etc/infiniband/openib.conf** to "**yes**".
2. Run **/etc/init.d/openibd restart** for the changes to take effect.

When loading the `ib_srp` module, it is possible to set the module parameter `srp_sg_tablesize.` This is the maximum number of gather/scatter entries per I/O (default: 12).

SRP Module Parameters

When loading the SRP module, the following parameters can be set (viewable by the "modinfo ib_srp" command):

`cmd_sg_entries`	Default number of gather/scatter entries in the SRP command (default is 12, max 255)
`allow_ext_sg`	Default behavior when there are more than cmd_sg_entries S/G entries after mapping; fails the request when false (default false)
`topspin_workarounds`	Enable workarounds for Topspin/Cisco SRP target bugs
`reconnect_delay`	Time between successive reconnect attempts. Time between successive reconnect attempts of SRP initiator to a disconnected target until dev_loss_tmo timer expires (if enabled), after that the SCSI target will be removed
`fast_io_fail_tmo`	Number of seconds between the observation of a transport layer error and failing all I/O. Increasing this timeout allows more tolerance to transport errors, however, doing so increases the total failover time in case of serious transport failure. Note: fast_io_fail_tmo value must be smaller than the value of recon- nect_delay
`dev_loss_tmo`	Maximum number of seconds that the SRP transport should insulate transport layer errors. After this time has been exceeded the SCSI target is removed. Normally it is advised to set this to -1 (disabled) which will never remove the scsi_host. In deployments where different SRP targets are connected and disconnected frequently, it may be required to enable this timeout in order to clean old scsi_hosts representing targets that no longer exists

Constraints between parameters:

- dev_loss_tmo, fast_io_fail_tmo, reconnect_delay cannot be all disabled or negative values.
- reconnect_delay must be positive number.
- fast_io_fail_tmo must be smaller than SCSI block device timeout.
- fast_io_fail_tmo must be smaller than dev_loss_tmo.

SRP Remote Ports Parameters

Several SRP remote ports parameters are modifiable online on existing connection.

To modify dev_loss_tmo to 600 seconds:

```
echo 600 > /sys/class/srp_remote_ports/port-xxx/dev_loss_tmo
```

To modify fast_io_fail_tmo to 15 seconds:

```
echo 15 > /sys/class/srp_remote_ports/port-xxx/fast_io_fail_tmo
```

To modify reconnect_delay to 10 seconds:

```
echo 20 > /sys/class/srp_remote_ports/port-xxx/reconnect_delay
```

Manually Establishing an SRP Connection

The following steps describe how to manually load an SRP connection between the Initiator and an SRP Target. “Automatic Discovery and Connection to Targets” section explains how to do this automatically.

- Make sure that the ib_srp module is loaded, the SRP Initiator is reachable by the SRP Target, and that an SM is running.
- To establish a connection with an SRP Target and create an SRP (SCSI) device for that target under /dev, use the following command:

```
echo -n id_ext=[GUID value],ioc_guid=[GUID value],dgid=[port GID value],\
pkey=ffff,service_id=[service[0] value] > \
/sys/class/infiniband_srp/srp-mlx[hca number]-[port number]/add_target
```

See “SRP Tools - ibsrpdm, srp_daemon and srpd Service Script” section for instructions on how the parameters in this echo command may be obtained.

Notes:

- Execution of the above “echo” command may take some time
- The SM must be running while the command executes
- It is possible to include additional parameters in the echo command:
  - max_cmd_per_lun - Default: 62
  - max_sect (short for max_sectors) - sets the request size of a command
  - io_class - Default: 0x100 as in rev 16A of the specification (In rev 10 the default was 0xff00)
  - tl_retry_count - a number in the range 2..7 specifying the IB RC retry count. Default: 2
  - comp_vector, a number in the range 0..n-1 specifying the MSI-X completion vector. Some HCA's allocate multiple (n) MSI-X vectors per HCA port. If the IRQ affinity masks of these interrupts have been configured such that each MSI-X interrupt is handled by a different CPU then the comp_vector parameter can be used to spread the SRP completion workload over multiple CPU's.
  - cmd_sg_entries, a number in the range 1..255 that specifies the maximum number of data buffer descriptors stored in the SRP_CMD information unit itself. With allow_ext_sg=0 the parameter cmd_sg_entries defines the maximum S/G list length

for a single SRP_CMD, and commands whose S/G list length exceeds this limit after S/G list collapsing will fail.
    - initiator_ext - see "Multiple Connections from Initiator InfiniBand Port to the Target" section.
- To list the new SCSI devices that have been added by the echo command, you may use either of the following two methods:
    - Execute “fdisk -l”. This command lists all devices; the new devices are included in this listing.
    - Execute “dmesg” or look at /var/log/messages to find messages with the names of the new devices.

SRP sysfs Parameters

Interface for making ib_srp connect to a new target. One can request ib_srp to connect to a new target by writing a comma-separated list of login parameters to this sysfs attribute. The supported parameters are:

id_ext	A 16-digit hexadecimal number specifying the eight byte identifier extension in the 16-byte SRP target port identifier. The target port identifier is sent by ib_srp to the target in the SRP_LOGIN_REQ request.
ioc_guid	A 16-digit hexadecimal number specifying the eight byte I/O controller GUID portion of the 16-byte target port identifier.
dgid	A 32-digit hexadecimal number specifying the destination GID.
pkey	A four-digit hexadecimal number specifying the InfiniBand partition key.
service_id	A 16-digit hexadecimal number specifying the InfiniBand service ID used to establish communication with the SRP target. How to find out the value of the service ID is specified in the documentation of the SRP target.
max_sect	A decimal number specifying the maximum number of 512-byte sectors to be transferred via a single SCSI command.
max_cmd_per_lun	A decimal number specifying the maximum number of outstanding commands for a single LUN.
io_class	A hexadecimal number specifying the SRP I/O class. Must be either 0xff00 (rev 10) or 0x0100 (rev 16a). The I/O class defines the format of the SRP initiator and target port identifiers.
initiator_ext	A 16-digit hexadecimal number specifying the identifier extension portion of the SRP initiator port identifier. This data is sent by the initiator to the target in the SRP_LOGIN_REQ request.
cmd_sg_entries	A number in the range 1..255 that specifies the maximum number of data buffer descriptors stored in the SRP_CMD information unit itself. With allow_ext_sg=0 the parameter cmd_sg_entries defines the maxi- mum S/G list length for a single SRP_CMD, and commands whose S/G list length exceeds this limit after S/G list collapsing will fail.
allow_ext_sg	Whether ib_srp is allowed to include a partial memory descriptor list in an SRP_CMD instead of the entire list. If a partial memory descriptor list has been included in an SRP_CMD the remaining memory descriptors are communicated from initiator to target via an additional RDMA transfer. Setting allow_ext_sg to 1 increases the maximum amount of data that can be transferred between initiator and target via a single SCSI command. Since not all SRP target implementations support partial memory descriptor lists the default value for this option is 0.
sg_tablesize	A number in the range 1..2048 specifying the maximum S/G list length the SCSI layer is allowed to pass to ib_srp. Specifying a value that exceeds cmd_sg_entries is only safe with partial memory descriptor list support enabled (allow_ext_sg=1).

comp_vector	A number in the range 0..n-1 specifying the MSI-X completion vector. Some HCA's allocate multiple (n) MSI-X vectors per HCA port. If the IRQ affinity masks of these interrupts have been configured such that each MSI-X interrupt is handled by a different CPU then the comp_vector parameter can be used to spread the SRP completion workload over multiple CPU's.
tl_retry_count	A number in the range 2..7 specifying the IB RC retry count.

SRP Tools - ibsrpdm, srp_daemon and srpd Service Script

The OFED distribution provides two utilities: ibsrpdm and srp_daemon:

- They detect targets on the fabric reachable by the Initiator (Step 1)
- Output target attributes in a format suitable for use in the above “echo” command (Step 2)
- A service script srpd which may be started at stack startup

The utilities can be found under /usr/sbin/, and are part of the srptools RPM that may be installed using the OFED installation. Detailed information regarding the various options for these utilities are provided by their man pages.
Below, several usage scenarios for these utilities are presented.

ibsrpdm

ibsrpdm has the following tasks:

1. Detecting reachable targets.
    a. To detect all targets reachable by the SRP initiator via the default umad device (/sys/class/infiniband_mad/umad0), execute the following command:

    ```
    ibsrpdm
    ```

    This command will result into readable output information on each SRP Target detected. Sample:

    ```
    IO Unit Info:
         port LID:        0103
         port GID:        fe80000000000000002c90200402bd5
         change ID:       0002
         max controllers: 0x10
      controller[  1]
         GUID:      0002c90200402bd4
         vendor ID: 0002c9
         device ID: 005a44
         IO class : 0100
         ID:        LSI Storage Systems SRP Driver 200400a0b81146a1
         service entries: 1
         service[  0]: 200400a0b81146a1 / SRP.T10:200400A0B81146A1
    ```

    b. To detect all the SRP Targets reachable by the SRP Initiator via another umad device, use the following command:

    ```
    ibsrpdm -d <umad device>
    ```

2. Assisting in SRP connection creation.
    a. To generate an output suitable for utilization in the “echo” command in “Manually Establishing an SRP Connection” section, add the ‘-c’ option to ibsrpdm:

    ```
    ibsrpdm -c
    ```

    Sample output:

```
id_ext=200400A0B81146A1,ioc_guid=0002c90200402bd4,
dgid=fe80000000000000002c90200402bd5,pkey=ffff,service_id=200400a0b81146a1
```

b. To establish a connection with an SRP Target using the output from the 'ibsrpdm -c' example above, execute the following command:

```
echo -n id_ext=200400A0B81146A1,ioc_guid=0002c90200402bd4,
dgid=fe80000000000000002c90200402bd5,pkey=ffff,service_id=200400a0b81146a1 > /sys/
class/infiniband_srp/srp-mlx5_0-1/add_target
```

The SRP connection should now be up; the newly created SCSI devices should appear in the listing obtained from the '**fdisk -l**' command.

3. Discover reachable SRP Targets given an InfiniBand HCA name and port, rather than by just running **/sys/class/infiniband_mad/umad<N>** where **<N>** is a digit.

srpd

The srpd service script allows automatic activation and termination of the srp_daemon utility on all system live InfiniBand ports.

srp_daemon

srp_daemon utility is based on ibsrpdm and extends its functionality. In addition to the ibsrpdm functionality described above, srp_daemon can:

- Establish an SRP connection by itself (without the need to issue the "echo" command described in "Manually Establishing an SRP Connection" section)
- Continue running in background, detecting new targets and establishing SRP connections with them (daemon mode)
- Discover reachable SRP Targets given an infiniband HCA name and port, rather than just by **/dev/umad<N>** where **<N>** is a digit
- Enable High Availability operation (together with Device-Mapper Multipath)
- Have a configuration file that determines the targets to connect to:

1. srp_daemon commands equivalent to ibsrpdm:

```
"srp_daemon -a -o" is equivalent to "ibsrpdm"
"srp_daemon -c -a -o" is equivalent to "ibsrpdm -c"
```

Note: These srp_daemon commands can behave differently than the equivalent ibsrpdm command when /etc/srp_daemon.conf is not empty.

2. srp_daemon extensions to ibsrpdm.

- To discover SRP Targets reachable from the HCA device <InfiniBand HCA name> and the port <port num>, (and to generate output suitable for 'echo'), execute:

```
host1# srp_daemon -c -a -o -i <InfiniBand HCA name> -p <port number>
```

Note: To obtain the list of InfiniBand HCA device names, you can either use the ibstat tool or run 'ls /sys/class/infiniband'.

- To both discover the SRP Targets and establish connections with them, just add the -e option to the above command.
- Executing srp_daemon over a port without the -a option will only display the reachable targets via the port and to which the initiator is not connected. If executing with the -e option it is better to omit -a.

- It is recommended to use the -n option. This option adds the initiator_ext to the connecting string (see "Multiple Connections from Initiator InfiniBand Port to the Target" section).
- srp_daemon has a configuration file that can be set, where the default is /etc/srp_daemon.conf. Use the -f to supply a different configuration file that configures the targets srp_daemon is allowed to connect to. The configuration file can also be used to set values for additional parameters (e.g., max_cmd_per_lun, max_sect).
- A continuous background (daemon) operation, providing an automatic ongoing detection and connection capability. See "Automatic Discovery and Connection to Targets" section.

Automatic Discovery and Connection to Targets

- Make sure the ib_srp module is loaded, the SRP Initiator can reach an SRP Target, and that an SM is running.
- To connect to all the existing Targets in the fabric, run "`srp_daemon -e -o`". This utility will scan the fabric once, connect to every Target it detects, and then exit.

  > ⚠ srp_daemon will follow the configuration it finds in /etc/srp_daemon.conf. Thus, it will ignore a target that is disallowed in the configuration file.

- To connect to all the existing Targets in the fabric and to connect to new targets that will join the fabric, execute srp_daemon -e. This utility continues to execute until it is either killed by the user or encounters connection errors (such as no SM in the fabric).
- To execute SRP daemon as a daemon on all the ports:
  - srp_daemon.sh (found under /usr/sbin/). srp_daemon.sh sends its log to /var/log/srp_daemon.log.
  - Start the srpd service script, run service srpd start

For the changes in openib.conf to take effect, run:

```
/etc/init.d/openibd restart
```

Multiple Connections from Initiator InfiniBand Port to the Target

Some system configurations may need multiple SRP connections from the SRP Initiator to the same SRP Target: to the same Target IB port, or to different IB ports on the same Target HCA. In case of a single Target IB port, i.e., SRP connections use the same path, the configuration is enabled using a different initiator_ext value for each SRP connection. The initiator_ext value is a 16-hexadecimal-digit value specified in the connection command.

Also in case of two physical connections (i.e., network paths) from a single initiator IB port to two different IB ports on the same Target HCA, there is need for a different initiator_ext value on each path. The conventions is to use the Target port GUID as the initiator_ext value for the relevant path.

If you use srp_daemon with -n flag, it automatically assigns initiator_ext values according to this convention. For example:

```
id_ext=200500A0B81146A1,ioc_guid=0002c90200402bec,\
```

```
dgid=fe8000000000000002c90200402bed,pkey=ffff,\ service_id=200500a0b81146a1,initiator_ext=ed2b400002c90200
```

Notes:

- It is recommended to use the -n flag for all srp_daemon invocations.
- ibsrpdm does not have a corresponding option.
- srp_daemon.sh always uses the -n option (whether invoked manually by the user, or automatically at startup by setting SRP_DAEMON_ENABLE to yes).

High Availability (HA)

High Availability works using the Device-Mapper (DM) multipath and the SRP daemon. Each initiator is connected to the same target from several ports/HCAs. The DM multipath is responsible for joining together different paths to the same target and for failover between paths when one of them goes offline. Multipath will be executed on newly joined SCSI devices.
Each initiator should execute several instances of the SRP daemon, one for each port. At startup, each SRP daemon detects the SRP Targets in the fabric and sends requests to the ib_srp module to connect to each of them. These SRP daemons also detect targets that subsequently join the fabric, and send the ib_srp module requests to connect to them as well.

Operation

When a path (from port1) to a target fails, the ib_srp module starts an error recovery process. If this process gets to the reset_host stage and there is no path to the target from this port, ib_srp will remove this scsi_host. After the scsi_host is removed, multipath switches to another path to this target (from another port/HCA).
When the failed path recovers, it will be detected by the SRP daemon. The SRP daemon will then request ib_srp to connect to this target. Once the connection is up, there will be a new scsi_host for this target. Multipath will be executed on the devices of this host, returning to the original state (prior to the failed path).

Manual Activation of High Availability

Initialization - execute after each boot of the driver:

1. Execute **`modprobe dm-multipath`**
2. Execute **`modprobe ib-srp`**
3. Make sure you have created file /etc/udev/rules.d/91-srp.rules as described above
4. Execute for each port and each HCA:

   ```
   srp_daemon -c -e -R 300 -i <InfiniBand HCA name> -p <port number>
   ```

   This step can be performed by executing srp_daemon.sh, which sends its log to /var/log/srp_daemon.log.

Now it is possible to access the SRP LUNs on /dev/mapper/.

> ⚠ It is possible for regular (non-SRP) LUNs to also be present; the SRP LUNs may be identified by their names. You can configure the /etc/multipath.conf file to change multipath behavior.

⚠ It is also possible that the SRP LUNs will not appear under /dev/mapper/. This can occur if the SRP LUNs are in the black-list of multipath. Edit the 'blacklist' section in /etc/multipath.conf and make sure the SRP LUNs are not blacklisted.

Automatic Activation of High Availability

- Start srpd service, run:

```
service srpd start
```

- From the next loading of the driver it will be possible to access the SRP LUNs on /dev/mapper/

  ⚠ It is possible that regular (not SRP) LUNs are also present. SRP LUNs may be identified by their name.

- It is possible to see the output of the SRP daemon in /var/log/srp_daemon.log

5.6.2.5.1.2 Shutting Down SRP

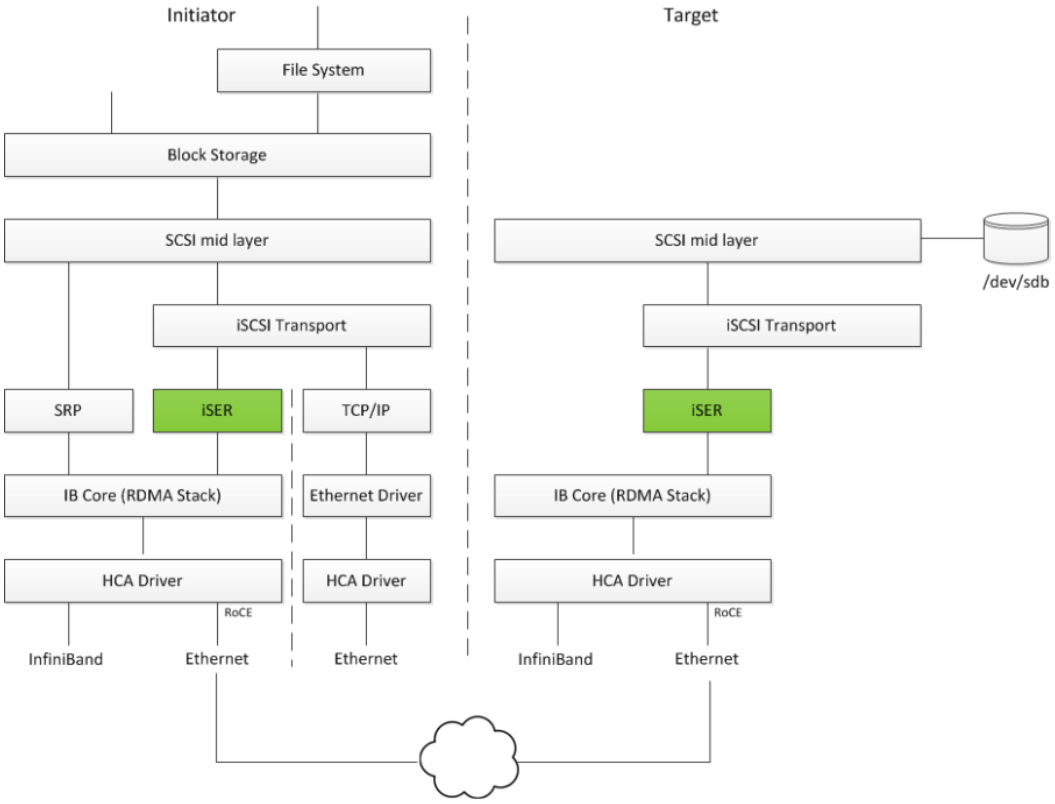
SRP can be shutdown by using "rmmod ib_srp", or by stopping the OFED driver ("/etc/init.d/openibd stop"), or as a by-product of a complete system shutdown.
Prior to shutting down SRP, remove all references to it. The actions you need to take depend on the way SRP was loaded. There are three cases:

1. Without High Availability
   When working without High Availability, you should unmount the SRP partitions that were mounted prior to shutting down SRP.
2. After Manual Activation of High Availability
   If you manually activated SRP High Availability, perform the following steps:
   a. Unmount all SRP partitions that were mounted.
   b. Stop service srpd (Kill the SRP daemon instances).
   c. Make sure there are no multipath instances running. If there are multiple instances, wait for them to end or kill them.
   d. Run: **multipath -F**
3. After Automatic Activation of High Availability
   If SRP High Availability was automatically activated, SRP shutdown must be part of the driver shutdown ("/etc/init.d/openibd stop") which performs Steps 2-4 of case b above. However, you still have to unmount all SRP partitions that were mounted before driver shutdown.

5.6.2.5.2 iSER - iSCSI Extensions for RDMA

iSCSI Extensions for RDMA (iSER) enhances the iSCSI protocol by integrating it with RDMA, enabling direct data transfers to and from SCSI buffers without intermediate copying.

By leveraging the RDMA protocol suite, iSER provides higher bandwidth for block storage transfers with zero-copy efficiency. This approach eliminates the overhead of TCP/IP processing while maintaining full compatibility with the iSCSI protocol.



There are three target implementation of ISER:

- Linux SCSI target framework (tgt)
- Linux-IO target (LIO)
- Generic SCSI target subsystem for Linux (SCST)

Each target can operate in either TCP or iSER transport mode.

iSER also supports RoCE without requiring additional configuration. To bond RoCE interfaces, set the **fail_over_mac** option in the bonding driver (see "Bonding IPoIB").

In the network stack, RDMA/RoCE is positioned below the iSER layer. To run iSER, ensure that the RDMA layer is properly configured and validated, whether over Ethernet or InfiniBand. For guidance on troubleshooting RDMA, refer to "HowTo Enable, Verify and Troubleshoot RDMA" community article.

5.6.2.5.2.1 iSER Initiator

The iSER initiator is controlled through the iSCSI interface available from the iscsi-initiator-utils package.

To discover and log into iSCSI targets, as well as access and manage the open-iscsi database use the **iscasiadm** utility, a command-line tool.

To enable iSER as a transport protocol use " **I iser** " as a parameter of the **iscasiadm** command.

Example for discovering and connecting targets over iSER:

```
iscsiadm -m discovery -o new -o old -t st -I iser -p <ip:port> -l
```

Note that the target implementation (e.g. LIO, SCST, TGT) does not affect he initiator process and configuration.

5.6.2.5.2.2 iSER Targets

> ⚠ Setting the iSER target is out of scope of this manual. For guidelines of how to do so, please refer to the relevant target documentation (e.g. stgt, targetcli).

Targets settings such as timeouts and retries are set the same as any other iSCSI targets.

> ⚠ If targets are set to auto connect on boot, and targets are unreachable, it may take a long time to continue the boot process if timeouts and max retries are set too high.

For various configuration, troubleshooting and debugging examples, refer to Storage Solutions on the Community website.

5.6.2.5.3 Lustre

Lustre is an open-source, parallel distributed file system, generally used for large-scale cluster computing that supports many requirements of leadership class HPC simulation environments.

Lustre Compilation for MLNX_OFED:

> ⚠ This procedure applies to RHEL/SLES OSs supported by Lustre. For further information, please refer to Lustre Release Notes.

➢ *To compile Lustre version 2.4.0 and higher:*

```
$ ./configure --with-o2ib=/usr/src/ofa_kernel/default/
$ make rpms
```

➢ *To compile older Lustre versions:*

```
$ EXTRA_LNET_INCLUDE="-I/usr/src/ofa_kernel/default/include/ -include /usr/src/ofa_kernel/default/include/linux/
compat-2.6.h" ./configure --with-o2ib=/usr/src/ofa_kernel/default/
$ EXTRA_LNET_INCLUDE="-I/usr/src/ofa_kernel/default/include/ -include /usr/src/ofa_kernel/default/include/linux/
compat-2.6.h" make rpms
```

For full installation example, refer to HowTo Install NVIDIA OFED driver for Lustre Community post.

5.6.2.5.4 NVME-oF - NVM Express over Fabrics

5.6.2.5.4.1 NVME-oF

NVME-oF enables NVMe message-based commands to transfer data between a host computer and a target solid-state storage device or system over a network such as Ethernet, Fibre Channel, and InfiniBand. Tunneling NVMe commands through an RDMA fabric provides a high throughput and a low latency.
For information on how to configure NVME-oF, please refer to the HowTo Configure NVMe over Fabrics Community post.

⚠ The --with-nvmf installation option should not be specified, if nvme-tcp kernel module is used. In this case, the native Inbox nvme-tcp kernel module will be loaded.

5.6.2.5.4.2 NVME-oF Target Offload

⚠ This feature is only supported for ConnectX-5 adapter cards family and above.

NVME-oF Target Offload is an implementation of the new NVME-oF standard Target (server) side in hardware. Starting from ConnectX-5 family cards, all regular IO requests can be processed by the HCA, with the HCA sending IO requests directly to a real NVMe PCI device, using peer-to-peer PCI communications. This means that excluding connection management and error flows, no CPU utilization will be observed during NVME-oF traffic.

- For instructions on how to configure NVME-oF target offload, refer to HowTo Configure NVME-oF Target Offload Community post.
- For instructions on how to verify that NVME-oF target offload is working properly, refer to Simple NVMe-oF Target Offload Benchmark Community post.

5.6.2.6 Virtualization

The chapter contains the following sections:

- SR-IOV
- Paravirtualization
- VXLAN Hardware Stateless Offloads
- Q-in-Q Encapsulation per VF in Linux
- VLAN Double Tagging
- Scalable Functions

5.6.2.6.1 SR-IOV

Single Root I/O Virtualization (SR-IOV) enables a single physical PCIe device to expose multiple virtual instances on the PCIe bus. Each instance, known as a virtual function (VF), acts as an independent PCIe device while sharing the physical function (PF)'s resources.

NVIDIA® ConnectX® adapters support up to 127 VFs per port, each of which can be provisioned and managed independently. SR-IOV is typically used with an SR-IOV-enabled hypervisor to provide virtual machines with direct hardware access to network interfaces, improving throughput and reducing CPU overhead.

This section describes how to configure SR-IOV in a Red Hat Enterprise Linux (RHEL) environment using ConnectX VPI adapters.

5.6.2.6.1.1 System Requirements

To configure and use SR-IOV, ensure the following prerequisites are met:

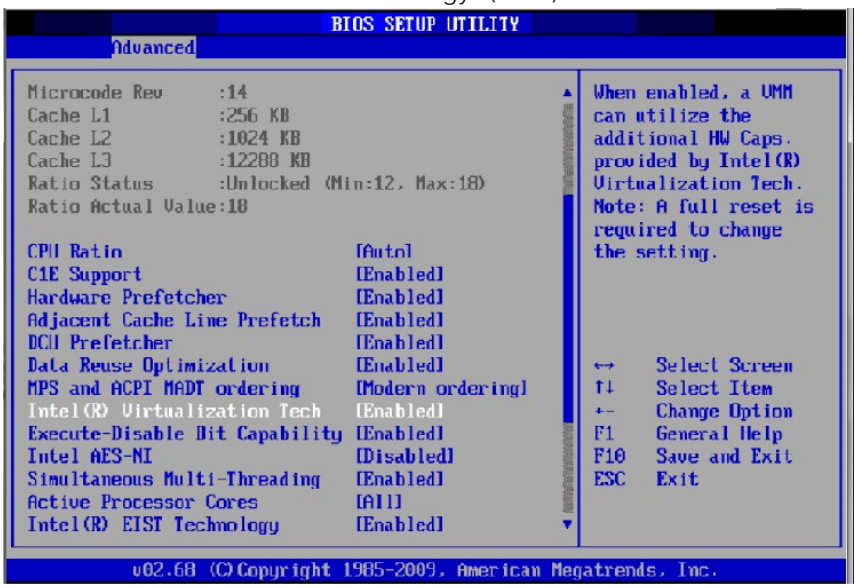
- Installed MLNX_OFED driver
- A server or blade with an SR-IOV-capable BIOS
- A hypervisor that supports SR-IOV (for example, Red Hat Enterprise Linux Server 6 or later)
- An ConnectX VPI adapter supporting SR-IOV

5.6.2.6.1.2 BIOS and Kernel Setup

> The figures used in this section are for illustration purposes only. For further information, refer to your BIOS User Manual.

1. Enable "SR-IOV" in the system BIOS.

2. Enable "Intel Virtualization Technology" (VT-d).



3. Install a hypervisor that supports SR-IOV.
4. Update the GRUB configuration to enable IOMMU:
   Example for Intel systems (**/boot/grub/grub.conf**):

```
default=0
timeout=5
splashimage=(hd0,0)/grub/splash.xpm.gz
hiddenmenu
title Red Hat Enterprise Linux Server (4.x.x)
        root (hd0,0)
        kernel /vmlinuz-4.x.x ro root=/dev/VolGroup00/LogVol00 rhgb quiet
        intel_iommu=on        initrd /initrd-4.x.x.img
```

> Ensure the parameter **intel_iommu=on** is present. On newer systems using **/boot/grub2/grub.cfg** , add the parameter to the line starting with **linux16** .

5.6.2.6.1.3 Configuring SR-IOV (Ethernet)

For configuration details, refer to the community guide HowTo Configure SR-IOV for ConnectX-4/ ConnectX- 5/ConnectX-6 with KVM (Ethernet).

5.6.2.6.1.4 Configuring SR-IOV (InfiniBand)

1. Install MLNX_OFED for Linux with SR-IOV support.
2. Verify SR-IOV enablement in the firmware:

```
mlxconfig -d /dev/mst/mt4115_pciconf0 q
```

Example output:

```
SRIOV_EN                    1
NUM_OF_VFS                  8
```

> To modify these settings, if needed:

```
mlxconfig -d /dev/mst/mt4115_pciconf0 set SRIOV_EN=1 NUM_OF_VFS=16
```

3. Reboot the server.
4. Create VFs. Depending on your kernel version, use one of the following sysfs files:
   - Standard (for newer kernels):

     ```
     echo <num_vfs> > /sys/class/infiniband/mlx5_0/device/sriov_numvfs
     ```

   - Legacy (for older kernels):

     ```
     echo <num_vfs> > /sys/class/infiniband/mlx5_0/device/mlx5_num_vfs
     ```

     The **sriov_numvfs** file is only present if **intel_iommu=on** was set in GRUB.

     Rules:
     - You can change the number of VFs only when none are assigned.
     - If VFs are assigned to VMs, the count cannot be changed.
     - Unloading the PF driver removes SR-IOV only if no VFs are assigned.
     - When the PF driver is reloaded, assigned VFs become operational again (the VF driver may need to be restarted).

5. Verify VF creation.

   ```
   lspci | grep Mellanox
   ```

   Example output:

   ```
   08:00.0 Infiniband controller: Mellanox Technologies MT27700 Family [ConnectX-4]
   08:00.1 Infiniband controller: Mellanox Technologies MT27700 Family [ConnectX-4]
   08:00.2 Infiniband controller: Mellanox Technologies MT27700 Family [ConnectX-4 Virtual Function]
   08:00.3 Infiniband controller: Mellanox Technologies MT27700 Family [ConnectX-4 Virtual Function]
   08:00.4 Infiniband controller: Mellanox Technologies MT27700 Family [ConnectX-4 Virtual Function]
   08:00.5 Infiniband controller: Mellanox Technologies MT27700 Family [ConnectX-4 Virtual Function]
   ```

6. Configure each VF. Sysfs entries are available under **/sys/class/infiniband/mlx5_<PF_INDEX>/device/sriov/** . Example output:

   ```
   sriov/
       0/
           node
           port
           policy
       1/
           node
           port
           policy
       2/
           node
           port
           policy
   ```

   - Node GUID:

     ```
     echo 00:11:22:33:44:55:1:0 > /sys/class/infiniband/mlx5_0/device/sriov/0/node
     ```

   - Port GUID:

```
echo 00:11:22:33:44:55:2:0 > /sys/class/infiniband/mlx5_0/device/sriov/0/port
```

- Policy (**/sys/class/infiniband/<PF>/device/sriov/<index>/policy**) – Defines VF port behavior. Options:

Value	Description
Down	Port state remains down
Up	Sets port to *Initialize*, allowing the SM to bring it up
Follow	Mirrors the physical port's state

> By default, all VF policies initialize as **Down** , except **VPort0** , which defaults to **Follow** .

7. Enable virtualization in OpenSM by adding the following to **/etc/opensm/opensm.conf** :

```
virt_enabled 2
```

> OpenSM and related InfiniBand tools (e.g., **iblinkinfo** , **ibqueryerr**) must run on the PF, not the VF. In multi-PF configurations, OpenSM should run on **host0** .

VF Initialization and Binding

Because the same **mlx5_core** driver handles both PFs and VFs, the PF driver attempts to initialize all VFs by default.
To assign a VF to a virtual machine, unbind it from the PF driver first:

1. Identify the VF PCIe address:

```
lspci -D
```

Example:

```
0000:09:00.2
```

2. Unbind from PF driver:

```
echo 0000:09:00.2 > /sys/bus/pci/drivers/mlx5_core/unbind
```

3. Bind again (if needed):

```
echo 0000:09:00.2 > /sys/bus/pci/drivers/mlx5_core/bind
```

PCIe BDF Mapping of PFs and VFs

PCIe addresses are sequential across PFs and VFs.

For example, if the card's PCIe slot is **05:00** and it has two ports:

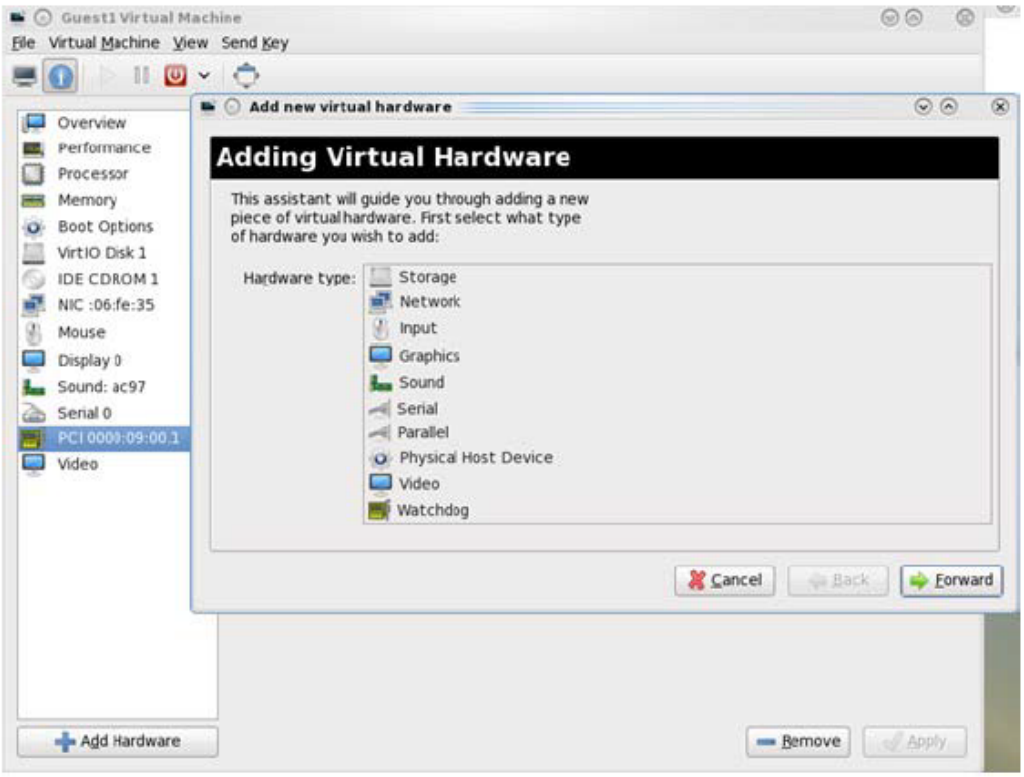
Function	PCIe BDF Range	Description
PF0	05:00.0	PF for port 0
PF1	05:00.1	PF for port 1
VFs for PF0	05:00.2–05:00.4	VFs 0–2 for PF0 (`mlx5_0`)
VFs for PF1	05:00.5–05:00.7	VFs 0–2 for PF1 (`mlx5_1`)

5.6.2.6.1.5 Additional SR-IOV Configurations

Assigning VF to Virtual Machine

This section describes how to attach an SR-IOV VF to a VM on a Red Hat KVM host using `virt-manager` (RHEL/KVM).

1. Run the virt-manager.
2. Double-click the VM and open its Properties.
3. Go to Details → Add Hardware → PCI Host Device.



4. Select the NVIDIA VF by its PCIe address (e.g., `00:03.1`).
5. Reboot the VM if it's running; otherwise, start it.
6. Inside the guest, verify the device is present:

```
lspci | grep Mellanox
```

Example:

```
01:00.0 Infiniband controller: Mellanox Technologies MT28800 Family [ConnectX-5 Ex]
```

7. (Optional) Configure the guest interface (e.g., via **/etc/sysconfig/network-scripts/ifcfg-ethX**).

> ⚠ VF MACs are randomly assigned by default; you don't need to set one unless you require a stable MAC.

Ethernet VF Configuration (Host)

You can configure VFs via **iproute2** (preferred) or **sysfs**.

- Using ip (preferred)

```
ip link set { dev <PF_DEVICE> | group <DEVGROUP> } [ up | down ] \
  vf <NUM> [ mac <LLADDR> ] [ vlan <VLANID> [ qos <VLAN-QOS> ] ] \
  [ spoofchk { on | off } ] \
  [ state { enable | disable | auto } ]
```

- Using sysfs (example layout, ConnectX-4)

```
/sys/class/net/<PF>/device/sriov/<VF>/
    config
    link_state
    mac
    mac_list
    max_tx_rate
    min_tx_rate
    spoofcheck
    stats
    trunk
    trust
```

VLAN Modes: VGT vs VST

- VGT (VLAN Guest Tagging) – Guest tags/untags its own traffic. *(Default)*
- VST (VLAN Switch Tagging) – Hypervisor enforces a VLAN/QoS for the VF; outgoing untagged/priority-tagged traffic is tagged by the hypervisor; incoming VLAN tags are stripped.

Configure VST:

```
ip link set dev <PF_DEVICE> vf <NUM> vlan <VLAN_ID> [qos <QOS>]
# Example:
ip link set dev eth2 vf 2 vlan 10 qos 3    # enable VST with VLAN 10, QoS 3
ip link set dev eth2 vf 2 vlan 0           # revert to VGT
```

Additional Ethernet VF Options

- Guest MAC (set a stable MAC before the guest driver loads):

```
ip link set dev <PF_DEVICE> vf <NUM> mac <LLADDR>
```

> ⚠ For legacy/ConnectX-4 guests (no random MAC), always configure via **ip link**.

- Spoof checking (kernel ≥ 3.1):

```
ip link set dev <PF_DEVICE> vf <NUM> spoofchk [on | off]
```

- Guest link state:

```
ip link set dev <PF_DEVICE> vf <UM> state [enable| disable| auto]
```

VF Statistics (sysfs)

The sysfs interface for displaying VF statistics is deprecated. To display VF statistics, use the following command:

```
/opt/mellanox/iproute2/sbin/ip -s link show dev <VF-interface-name>
```

On systems where the deprecated sysfs interface is still available, it displays the following counters:

```
# cat /sys/class/infiniband/mlx5_2/device/sriov/2/stats
tx_packets    : 5011
tx_bytes      : 4450870
tx_dropped    : 0
rx_packets    : 5003
rx_bytes      : 4450222
rx_broadcast : 0
rx_multicast : 0
rx_dropped    : 0
```

Mapping VFs to Ports

Use **ip link** (v2.6.34~3+):

```
ip link
```

Example (excerpt):

```
61: p1p1: ...
    vf 0 MAC 00:00:00:00:00:00, vlan 4095, spoof checking off, link-state auto
    vf 38 MAC ff:ff:ff:ff:ff:ff, vlan 65535, spoof checking off, link-state disable
```

A MAC of **ff:ff:ff:ff:ff:ff** indicates the VF is not assigned to this net device's port.

You can still configure such VFs from this PF; changes apply to the VF's actual port owner.

RoCE Support

RoCE is supported on VFs and can be used with VLANs. The hypervisor GID table has 16 entries; the remaining 112 entries are shared across VFs. With >56 VFs, some may have only a single GID entry, which is insufficient if a VF's Ethernet interface is assigned an IP. Plan VF counts accordingly.

SR-IOV Advanced Security

MAC Anti-Spoofing

Prevents a VF from sending frames with a MAC different from the one assigned by the admin. Disabled by default.

- Using **ip** (kernel ≥ 3.10):

```
ip link set ens785f1 vf 0 spoofchk on   # enable
ip link set ens785f1 vf 0 spoofchk off  # disable
```

- Using sysfs:

```
echo "ON"  > /sys/class/net/ens785f1/device/sriov/0/spoofcheck
echo "OFF" > /sys/class/net/ens785f1/device/sriov/0/spoofcheck
```

⚠ This setting is non-persistent across driver restarts.

Rate Limit per VF

See HowTo Configure Rate Limit per VF for ConnectX-4/ConnectX-5/ConnectX-6 Community post. Per-VF files (e.g., **/sys/class/net/<ifname>/device/sriov/<vf_num>/max_tx_rate**) still apply.

Rate Limit per Group of VFs

Group VFs and apply a group rate limit; effective VF limit is the min of the VF's own limit and the group's available bandwidth share.

Configuration outline:

1. When supported, the driver exposes **/sys/class/net/<ifname>/device/sriov/groups/**.
2. All VFs start in group 0.
3. Move a VF to a group:

```
echo 7 > /sys/class/net/<ifname>/device/sriov/5/group
```

4. Set group max rate:

```
echo 5000 > /sys/class/net/<ifname>/device/sriov/groups/7/max_tx_rate
```

5. Inspect VF/group:
   - VF stats include group ID:

```
cat /sys/class/net/<ifname>/device/sriov/<vf_num>/stats
```

   - Group config shows current rate limit and member count:

```
cat /sys/class/net/<ifname>/device/sriov/groups/<group_id>/config
```

Bandwidth Guarantee per Group of VFs

Guarantee a minimum transmit rate per group; ensure the sum of group minimums ≤ line rate.

Example (40 Gb/s link):

```
echo 20000 > /sys/class/net/<ifname>/device/sriov/groups/1/min_tx_rate
echo 5000 > /sys/class/net/<ifname>/device/sriov/groups/2/min_tx_rate
echo 15000 > /sys/class/net/<ifname>/device/sriov/groups/3/min_tx_rate
```

- Group 1: 20 Gb/s
- Group 2: 5 Gb/s
- Group 3: 15 Gb/s

- Groups with 0 have no guarantee.

> ⚠ You can still set per-VF min rates to split a group's guarantee among member VFs (sum should not exceed the group minimum).

Privileged VFs

Trusted VFs can receive a limited set of PF-like privileges (e.g., entering promiscuous mode).

- Using **ip** (kernel ≥ 4.5):

```
ip link set ens785f1 vf 0 trust on
ip link set ens785f1 vf 0 trust off
```

- Using sysfs:

```
echo "ON"  > /sys/class/net/ens785f1/device/sriov/0/trust
echo "OFF" > /sys/class/net/ens785f1/device/sriov/0/trust
```

Probed VFs

Probing VFs consumes resources. Disable probing if you don't need to monitor VMs:

- Kernel ≥ 4.12 (preferred) – use **sriov_drivers_autoprobe** (PCIe sysfs).
- Older kernels – use **mlx5_core** module param **probe_vf**:

```
echo 0 > /sys/module/mlx5_core/parameters/probe_vf
```

For more information on how to probe VFs, see HowTo Configure and Probe VFs on mlx5 Drivers Community post.

VF Promiscuous and All-multicast Modes

> ⚠ Only trusted VFs can enable these modes.

- Promiscuous Mode (receive unmatched and all multicast traffic):

```
ifconfig eth2 promisc     # enable
ifconfig eth2 -promisc    # disable
```

- All-Multicast Mode (receive all multicast on the port):

```
ifconfig eth2 allmulti    # enable
ifconfig eth2 -allmulti   # disable
```

5.6.2.6.1.6 Uninstalling the SR-IOV Driver

1. Detach all VFs from VMs or stop the VMs that use VFs.

> ⛔ Stopping the driver while VMs are using VFs may hang the host.

2. Run the uninstall script:

```
/usr/sbin/ofed_uninstall.sh
```

Follow the prompts. Example output (truncated):

```
This program will uninstall all OFED packages on your machine.
Do you want to continue? [y/N]: y
...
```

3. Reboot the server.

5.6.2.6.1.7 SR-IOV Live Migration

> ⚠ This feature is supported in Ethernet mode only.

1. Introduction

Live migration refers to the process of moving a guest virtual machine (VM) running on one physical host to another host without disrupting normal operations or causing other adverse effects for the end user.

Using the Migration process is useful for:

- Load balancing
- Hardware independence
- Energy saving
- Geographic migration
- Fault tolerance

Migration works by sending the state of the guest virtual machine's memory and any virtualized devices to a destination host physical machine. Migrations can be performed live or not, in the live case, the migration will not disrupt the user operations and it will be transparent to it as explained in the following sections.

1.1. Non-Live Migration

When using the non-live migration process, the Hypervisor suspends the guest virtual machine, then moves an image of the guest virtual machine's memory to the destination host physical machine. The guest virtual machine is then resumed on the destination host physical machine, and the memory the guest virtual machine used on the source host physical machine is freed. The time it takes to complete such a migration depends on the network bandwidth and latency. If the network is experiencing heavy use or low bandwidth, the migration will take longer then desired.

1.2. Live Migration

When using the Live Migration process, the guest virtual machine continues to run on the source host physical machine while its memory pages are transferred to the destination host physical machine. During migration, the Hypervisor monitors the source for any changes in the pages it has already transferred and begins to transfer these changes when all of the initial pages have been transferred.

It also estimates transfer speed during migration, so when the remaining amount of data to transfer will take a certain configurable period of time, it will suspend the original guest virtual

machine, transfer the remaining data, and resume the same guest virtual machine on the destination host physical machine.

1.2.1. MLX5 VF Live Migration

The purpose of this section is to demonstrate how to perform basic live migration of a QEMU VM with an MLX5 VF assigned to it. This section does not explains how to create VMs either using libvirt or directly via QEMU.

1.2.1.1. Requirements

The below are the requirements for working with MLX5 VF Live Migration.

Components	Description
Adapter Cards	• ConnectX-7 ETH • BlueField-3 ETH ⚠ The same PSID must be used on both the source and the target hosts (identical cards, same CAPs and features are needed), and have the same firmware version.
Firmware	• 28.41.1000 • 32.41.1000
Kernel	Linux v6.7 or newer
User Space Tools	iproute2 version 6.2 or newer
QEMU	QEMU 8.1 or newer
Libvirt	Libvirt 8.6 or newer

2. Setup

2.1.1. NVCONFIG

SR-IOV should be enabled and be configured to support the required number of VFs as of enabling live migration. This can be achieved by the below command:

```
mlxconfig -d *<PF_BDF>* s SRIOV_EN=1 NUM_OF_VFS=4 VF_MIGRATION_MODE=2
```

Where:

- **SRIOV_EN** – Enable SR-IOV.
- **NUM_OF_VFS** – The total number of VFs that can be supported, for each PF.
- **VF_MIGRATION_MODE** – Defines support for VF migration:
  - **0x0** – **DEVICE_DEFAULT**
  - **0x1** – **MIGRATION_DISABLED**
  - **0x2** – **MIGRATION_ENABLED**

2.1.2. Kernel Configuration

Needs to be compiled with driver **MLX5_VFIO_PCI** enabled. (i.e. **CONFIG_MLX5_VFIO_PCI**).

To load the driver, run:

```
modprobe mlx5_vfio_pci
```

2.1.3. QEMU

Needs to be compiled with **VFIO_PCI** enabled (this is enabled by default).

2.1. Host Preparation

As stated earlier, creating the VMs is beyond the scope of this guide and we assume that they are already created. However, the VM configuration should be a migratable configuration, similarly to how it is done without SR-IOV VFs.

> ⚠ The steps below should be done before running the VMs.

2.1.1. Over libvirt

1. Set the PF in the "switchdev" mode (on both the source and destination hosts).

   ```
   devlink dev eswitch set pci/<PF_BDF> mode switchdev
   ```

2. Create the VFs that will be assigned to the VMs (on both the source and destination hosts).

   ```
   echo "1" > /sys/bus/pci/devices/<PF_BDF>/sriov_numvfs
   ```

3. Set the VFs as migration capable (on both the source and destination hosts).
   a. See the name and PCI path (**VF_BDF**) of the VFs, run:

      ```
      lspci -D | grep nox
      ```

   b. Unbind the VFs from **mlx5_core** , run:

      ```
      echo '<VF_BDF>' > /sys/bus/pci/drivers/mlx5_core/unbind
      ```

   c. Determine the VF's physical function (**PF_BDF**) using the **VF_BDF** from step 3.a.:

      ```
      ls -la /sys/bus/pci/devices/<VF_BDF>/physfn
      ```

   d. Use devlink to set each VF as migration capable, using the **PF_BDF** from step c, run:

      ```
      devlink port function set pci/<PF_BDF>/1 migratable enable
      ```

   e. Check migration enabled using command

      ```
      devlink port show pci/<PF_BDF>/1
      ```

4. Assign the VFs to the VMs (in both source and destination VMs).
   a. Get the desired VM names:

```
virsh list --all
```

b. Edit the VMs XML file, run:

```
virsh edit <VM_NAME>
```

c. Assign the VFs to the VM by adding the following under the "devices" tag:

```
<hostdev mode='subsystem' type='pci' managed='no'>
  <driver name='vfio'/>
  <source>
    <!-- below is the VF BDF as seen on the Host.
    Note: the VF BDF can be different on the source
    and destination Hosts -->
    <address domain='0x0000' bus='0x08' slot='0x00' function='0x2'/>
  </source>
  <!-- below is the VF BDF as it will appear on the VM.
  Note that the VF BDF must be the same on both the source
  and destination VMs -->
  <address type='pci' domain='0x0000' bus='0x09' slot='0x00' function='0x0'/>
</hostdev>
```

5. Set the destination VM in incoming mode.

a. Edit the destination VM XML file, run:

```
virsh edit <VM_NAME>
```

b. Set the destination VM in migration incoming mode by adding the following under "domain" tag:

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
  [...]
  <qemu:commandline>
    <qemu:arg value='--incoming'/>
    <qemu:arg value='tcp:0:<DEST_PORT>'/>
  </qemu:commandline>
</domain>
```

6. Bind the VFs to **mlx5_vfio_pci** driver (on both the source and destination hosts).

1. a. Identify which **VF_BDF** to detach:

```
virsh nodedev-list
```

b. Detach the VFs from **libvirt** management, run:

```
virsh nodedev-detach pci_<VF_BDF>
```

c. Check which driver was used:

```
lspci -k | less
```

Expected output:

```
Kernel driver in use: mlx5_vfio_pci
```

d. If you see that the kernel driver in use is **vfio-pci**, do the following:

i. Rebind the VF to **mlx5_vfio_pci**:

```
echo '<VF_BDF>' > /sys/bus/pci/drivers/vfio-pci/unbind
```

ii. Set driver override, run:

```
echo 'mlx5_vfio_pci' > /sys/bus/pci/devices/<VF_BDF>/driver_override
```

iii. Bind the VFs to **mlx5_vfio_pci** driver, run:

```
echo '<VF_BDF>' > /sys/bus/pci/drivers/mlx5_vfio_pci/bind
```

2.1.2. Directly over QEMU

1. Set the PF in "switchdev" mode.

```
devlink dev eswitch set pci/<PF_BDF> mode switchdev
```

2. Create the VFs that will be assigned to the VMs.

```
echo "1" > /sys/bus/pci/devices/<PF_BDF>/sriov_numvfs
```

3. Set the VFs as migration capable.
   a. See the name of the VFs, run:

```
devlink port show
```

   b. Unbind the VFs from mlx5_core, run:

```
echo '<VF_BDF>' > /sys/bus/pci/drivers/mlx5_core/unbind
```

   c. Use devlink to set each VF as migration capable, run:

```
devlink port function set pci/<PF_BDF>/1 migratable enable
```

4. Bind the VFs to **mlx5_vfio_pci** driver:
   a. Set driver override, run:

```
echo 'mlx5_vfio_pci' > /sys/bus/pci/devices/<VF_BDF>/driver_override
```

   b. Bind the VFs to **mlx5_vfio_pci** driver, run:

```
echo '<VF_BDF>' > /sys/bus/pci/drivers/mlx5_vfio_pci/bind
```

2.2. Running the Migration

2.2.1. Over libvirt

1. Check if the VM is running:

```
virsh domstate <VM_NAME>
```

If the output shows **running**, destroy the VM:

```
virsh destroy <VM_NAME>
```

2. Start the VMs on the source and destination hosts:

```
virsh start <VM_NAME>
```

3. Enable switchover-ack QEMU migration capability. Run the following commands both in source and destination:

```
virsh qemu-monitor-command <VM_NAME> --hmp "migrate_set_capability return-path on"
```

```
virsh qemu-monitor-command <VM_NAME> --hmp "migrate_set_capability switchover-ack on"
```

4. [Optional] Configure the migration bandwidth and downtime limit in source side:

```
virsh qemu-monitor-command <VM_NAME> --hmp "migrate_set_parameter max-bandwidth <VALUE>"
virsh qemu-monitor-command <VM_NAME> --hmp "migrate_set_parameter downtime-limit <VALUE>"
```

5. Start migration by running the migration command in source side:

```
virsh qemu-monitor-command <VM_NAME> --hmp "migrate -d tcp:<DEST_IP>:<DEST_PORT>"
```

6. Check the migration status by running the info command in source side:

```
virsh qemu-monitor-command <VM_NAME> --hmp "info migrate"
```

2.2.2. Directly over QEMU

1. Start the VM in the source Host with the VF assigned to it:

```
qemu-system-x86_64 [...] -device vfio-pci,host=<VF_BDF>,id=mlx5_1
```

2. Start the VM in the destination Host with the VF assigned to it and with the "incoming" parameter:

```
qemu-system-x86_64 [...] -device vfio-pci,host=<VF_BDF>,id=mlx5_1
 --incoming tcp:0:<DEST_PORT>
```

3. Enable switchover-ack QEMU migration capability. Run the following commands in QEMU monitor, both in source and destination:

```
migrate_set_capability return-path on
```

```
migrate_set_capability switchover-ack on
```

4. [Optional] Configure the migration bandwidth and downtime limit in source side:

```
migrate_set_parameter max-bandwidth <VALUE>
migrate_set_parameter downtime-limit <VALUE>
```

5. Start migration by running the migration command in QEMU monitor in source side:

```
migrate -d tcp:<DEST_IP>:<DEST_PORT>
```

6. Check the migration status by running the info command in QEMU monitor in source side:

```
info migrate
```

2.3. Migration with Multi-port vHCA

Enables the usage of a dual port virtual HCA (vHCA) to share RDMA resources (e.g., MR, CQ, SRQ, PDs) across the two Ethernet (RoCE) NIC network ports and display the NIC as a dual port device.

Multi-port vHCA (MPV) VF is made of 2 "regular" VFs, one VF of each port. Creating a migratable MPV VF requires the same steps as regular VF (see the steps in section "Over libvirt"). The steps should be performed on each of the NIC ports. MPV VFs traffic cannot be configured with OVS. TC rules must be defined to configure the MPV VFs traffic.

2.4. Notes

⚠ In ConnectX-7 adapter cards, migration cannot run in parallel on more than 4 VFs. It is the administrator's responsibility to control that.

⚠ Live migration requires same firmware version on both the source and the target hosts.

5.6.2.6.2 Paravirtualization

The following steps demonstrate how to enable paravirtualization networking support on RHEL 7.x systems without NetworkManager, using a manually configured bridge.

1. Create a configuration file for the bridge interface (**bridge0**):

   ```
   vim /etc/sysconfig/network-scripts/ifcfg-bridge0
   ```

   Example contents:

   ```
   DEVICE=bridge0
   TYPE=Bridge
   IPADDR=12.195.15.1
   NETMASK=255.255.0.0
   BOOTPROTO=static
   ONBOOT=yes
   NM_CONTROLLED=no
   DELAY=0
   ```

2. Modify the configuration of the physical Ethernet interface (in this example, **eth5**) to associate it with the bridge:

   ```
   vim /etc/sysconfig/network-scripts/ifcfg-eth5
   ```

   Example contents:

   ```
   DEVICE=eth5
   BOOTPROTO=none
   STARTMODE=on
   HWADDR=00:02:c9:2e:66:52
   TYPE=Ethernet
   NM_CONTROLLED=no
   ONBOOT=yes
   BRIDGE=bridge0
   ```

3. Apply the configuration changes by restarting the network service:

```
service network restart
```

> ⚠ This will temporarily disrupt network connectivity.

4. Once the bridge is active, you can assign it to a virtual machine (VM). After booting the VM, verify that the virtual interface is using the bridged connection:

```
ifconfig -a
```

Example output:

```
eth6      Link encap:Ethernet  HWaddr 52:54:00:E7:77:99
          inet addr:13.195.15.5  Bcast:13.195.255.255  Mask:255.255.0.0
          inet6 addr: fe80::5054:ff:fee7:7799/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:481 errors:0 dropped:0 overruns:0 frame:0
          TX packets:450 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:22440 (21.9 KiB)  TX bytes:19232 (18.7 KiB)
```

5.6.2.6.3 VXLAN Hardware Stateless Offloads

VXLAN (Virtual Extensible LAN) technology addresses scalability and isolation challenges in modern data center networks by encapsulating Layer 2 frames within Layer 3 packets. However, this encapsulation introduces performance challenges that require extending traditional stateless offload capabilities to VXLAN traffic.

The ConnectX® family of adapter cards provides hardware acceleration for VXLAN-encapsulated packets, offering the same stateless offloads typically available for non-encapsulated traffic. VXLAN packets are encapsulated using an outer UDP header.

The following stateless offloads are supported for VXLAN traffic:

- Checksum Generation
  - Inner IP checksum
  - Inner TCP/UDP checksum
- Checksum Validation
  - Inner IP checksum
  - Inner TCP/UDP checksum
- TSO (TCP Segmentation Offload)
  - Supported for inner TCP payloads
- RSS (Receive Side Scaling)
  - Flow distribution based on inner packet attributes (e.g., IP addresses, TCP/UDP ports)
- Receive Queue Selection
  - Inner frames can be steered to specific Queue Pairs (QPs) for advanced traffic steering and isolation

5.6.2.6.3.1 Enabling VXLAN Hardware Stateless Offloads

VXLAN offload is enabled by default for ConnectX-4 family devices running the minimum required firmware version and a kernel version that includes VXLAN support.

To confirm if the current setup supports VXLAN, run:

```
ethtool -k $DEV | grep udp_tnl
```

Example:

```
ethtool -k ens1f0 | grep udp_tnl
tx-udp_tnl-segmentation: on
```

ConnectX-4 family devices support configuring multiple UDP ports for VXLAN offload. Ports can be added to the device by configuring a VXLAN device from the OS command line using the `ip` command.

> ⚠ If you configure multiple UDP ports for offload and exceed the total number of ports supported by hardware, then those additional ports will still function properly, but will not benefit from any of the stateless offloads.

Example:

```
ip link add vxlan0 type vxlan id 10 group 239.0.0.10 ttl 10 dev ens1f0 dstport 4789
ip addr add 192.168.4.7/24 dev vxlan0
ip link set up vxlan0
```

> ⚠ dstport' parameters are not supported in Ubuntu 14.4.

The VXLAN ports can be removed by deleting the VXLAN interfaces.

Example:

```
ip link delete vxlan0
```

5.6.2.6.3.2 Important Note

VXLAN tunneling adds an encapsulation overhead of 50 bytes per packet, comprising:

- 14 bytes – Ethernet header
- 20 bytes – Outer IP header
- 8 bytes – UDP header
- 8 bytes – VXLAN header

This overhead must be accounted for to prevent packet fragmentation or drops.

> ⚠ VXLAN adds 50 bytes of encapsulation overhead. To avoid fragmentation or packet loss, you must account for this overhead by adjusting the MTU:
>
> - Option 1: Decrease the MTU on the sending interface (e.g., VM virtio-net, host veth). For example: Use MTU 1450 instead of 1500.
> - Option 2: Increase the MTU on the physical uplink. For example: Use MTU 1550 instead of 1500.

5.6.2.6.4 Q-in-Q Encapsulation per VF in Linux

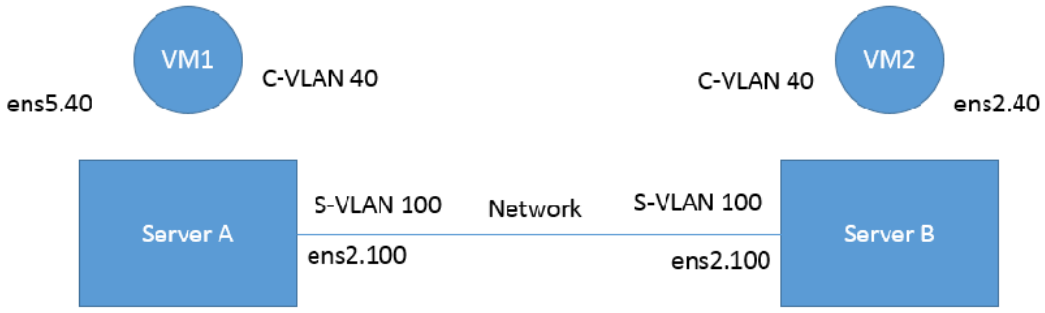
> ⚠ This feature is supported on ConnectX-5 and ConnectX-6 adapter cards only.

This section explains how to configure IEEE 802.1ad Q-in-Q VLAN tagging (S-VLAN) for a hypervisor on a per-VF basis. A VM connected to the VF via SR-IOV can transmit traffic with or without a C-VLAN tag.

When a VF is configured for VST Q-in-Q encapsulation, the adapter hardware automatically inserts an S-VLAN tag into outgoing packets from the VF to the physical port. For incoming packets, the hardware removes the S-VLAN tag before delivering the traffic to the VF.

5.6.2.6.4.1 Setup

The setup assumes there are two servers equipped with ConnectX-5/ConnectX-6 adapter cards.



5.6.2.6.4.2 Prerequisites

- Kernel must be of v3.10 or higher, or custom/inbox kernel must support vlan-stag
- Firmware version 16/20.21.0458 or higher must be installed for ConnectX-5/ConnectX-6 HCAs
- The server should be enabled in SR-IOV and the VF should be attached to a VM on the hypervisor.
- Network Considerations - the network switches may require increasing the MTU (to support 1522 MTU size) on the relevant switch ports.

5.6.2.6.4.3 Configuring Q-in-Q Encapsulation per Virtual Function for ConnectX-5/ ConnectX-6

1. Add the required S-VLAN (QinQ) tag (on the hypervisor) per port per VF. There are two ways to add the S-VLAN:
   a. By using sysfs:

```
echo '100:0:802.1ad' > /sys/class/net/ens1f0/device/sriov/0/vlan
```

   b. By using the ip link command (available only when using the latest Kernel version):

```
ip link set dev ens1f0 vf 0 vlan 100 proto 802.1ad
```

Check the configuration using the ip link show command:

```
# ip link show ens1f0
 ens1f0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT qlen 1000
    link/ether ec:0d:9a:44:37:84 brd ff:ff:ff:ff:ff:ff
    vf 0 MAC 00:00:00:00:00:00, vlan 100, vlan protocol 802.1ad, spoof checking off, link-state
auto, trust off
    vf 1 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
    vf 2 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
    vf 3 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
    vf 4 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
```

2. **Optional:** Add S-VLAN priority. Use the qos parameter in the ip link command (or sysfs):

```
ip link set dev ens1f0 vf 0 vlan 100 qos 3 proto 802.1ad
```

Check the configuration using the ip link show command:

```
# ip link show ens1f0
ens1f0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT qlen 1000
    link/ether ec:0d:9a:44:37:84 brd ff:ff:ff:ff:ff:ff
    vf 0 MAC 00:00:00:00:00:00, vlan 100, qos 3, vlan protocol 802.1ad, spoof checking off, link-state
auto, trust off
    vf 1 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
    vf 2 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
    vf 3 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
    vf 4 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
```

3. Create a VLAN interface on the VM and add an IP address.

```
ip link add link ens5 ens5.40 type vlan protocol 802.1q id 40
ip addr add 42.134.135.7/16 brd 42.134.255.255 dev ens5.40
ip link set dev ens5.40 up
```

4. To verify the setup, run ping between the two VMs and open Wireshark or tcpdump to capture the packet.

5.6.2.6.5 VLAN Double Tagging

This section describes the configuration of 802.1Q double-tagging support to the hypervisor per Virtual Function (VF). The Virtual Machine (VM) attached to the VF (via SR-IOV) can send traffic with or without C-VLAN. Once a VF is configured to VST encapsulation, the adapter's hardware will insert C-VLAN to any packet from the VF to the physical port. On the receive side, the adapter hardware will strip the C-VLAN from any packet coming from the wire to that VF.

5.6.2.6.5.1 Configuring 802.1Q Double-Tagging per Virtual Function

1. Add the required C-VLAN tag (on the hypervisor) per port per VF. There are two ways to add the C-VLAN:
    a. By using sysfs:

```
echo '100:0:802.1q' > /sys/class/net/ens1f0/device/sriov/0/vlan
```

    b. By using the ip link command (available only when using the latest Kernel version):

```
ip link set dev ens1f0 vf 0 vlan 100
```

Check the configuration using the ip link show command:

```
# ip link show ens1f0
```

```
ens1f0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT qlen 1000
    link/ether ec:0d:9a:44:37:84 brd ff:ff:ff:ff:ff:ff
    vf 0 MAC 00:00:00:00:00:00, vlan 100, spoof checking off, link-state auto, trust off
    vf 1 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
    vf 2 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
    vf 3 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
    vf 4 MAC 00:00:00:00:00:00, spoof checking off, link-state auto, trust off
```

2. Create a VLAN interface on the VM and add an IP address.

```
# ip link add link ens5 ens5.40 type vlan protocol 802.1q id 40
# ip addr add 42.134.135.7/16 brd 42.134.255.255 dev ens5.40
# ip link set dev ens5.40 up
```

3. To verify the setup, run ping between the two VMs and open Wireshark or tcpdump to capture the packet.

5.6.2.6.6 Scalable Functions

Scalable function is a lightweight function that has a parent PCI function on which it is deployed. Scalable functions are useful for containers where netdevice and RDMA devices of a scalable function can be assigned to a container. This way, the container can get complete offload capabilities of an eswitch, isolation and dedicated accelerated network device. For Step-by-Step Configuration instructions, follow the User Guide here.

5.6.2.7 Resiliency

The chapter contains the following sections:

- Reset Flow

5.6.2.7.1 Reset Flow

Reset Flow is activated by default. Once a "fatal device" error is recognized, both the HCA and the software are reset, the ULPs and user application are notified about it, and a recovery process is performed once the event is raised.

Currently, a reset flow can be triggered by a firmware assert with Recover Flow Request (RFR) only. Firmware RFR support should be enabled explicitly using mlxconfig commands.

- To query the current value, run:

```
mlxconfig -d /dev/mst/mt4115_pciconf0 query | grep SW_RECOVERY_ON_ERRORS
```

- To enable RFR bit support, run:

```
mlxconfig -d /dev/mst/mt4115_pciconf0 set SW_RECOVERY_ON_ERRORS=true
```

5.6.2.7.1.1 Kernel ULPs

Once a "fatal device" error is recognized, an `IB_EVENT_DEVICE_FATAL` event is created, ULPs are notified about the incident, and outstanding WQEs are simulated to be returned with "flush in error" message to enable each ULP to close its resources and not get stuck via calling its `remove_one` callback as part of "Reset Flow".

Once the unload part is terminated, each ULP is called with its **add_one** callback, its resources are re-initialized and it is re-activated.

5.6.2.7.1.2 User Space Applications (IB/RoCE)

Once a "fatal device" error is recognized an **IB_EVENT_DEVICE_FATAL** event is created, applications are notified about the incident and relevant recovery actions are taken.

Applications that ignore this event enter a zombie state, where each command sent to the kernel is returned with an error, and no completion on outstanding WQEs is expected.

The expected behavior from the applications is to register to receive such events and recover once the above event is raised. Same behavior is expected in case the NIC is unbounded from the PCI and later is rebounded. Applications running over RDMA CM should behave in the same manner once the **RDMA_CM_EVENT_DEVICE_REMOVAL** event is raised.

The following is an example of using the unbind/bind for NIC defined by **0000:04:00.0**:

```
echo 0000:04:00.0 > /sys/bus/pci/drivers/mlx5_core/unbind
echo 0000:04:00.0 > /sys/bus/pci/drivers/mlx5_core/bind
```

5.6.2.7.1.3 SR-IOV

If the PF recognizes the error, it notifies all the VFs about it by marking their communication channel with that information, consequently, all the VFs and the PF are reset.

If the VF encounters an error, only that VF is reset, whereas the PF and other VFs continue to work unaffected.

5.6.2.7.1.4 Forcing the VF to Reset

If an outside reset is forced by using the PCIe sysfs entry for a VF, a reset is executed on that VF once it runs any command over its communication channel.

For example, the following command can be used on a hypervisor to reset a VF defined by **0000:04:00.1**:

```
echo 1 >/sys/bus/pci/devices/0000:04:00.1/reset
```

5.6.2.7.1.5 Extended Error Handling (EEH)

Extended Error Handling (EEH) is a PowerPC mechanism that encapsulates AER, thus exposing AER events to the operating system as EEH events.

The behavior of ULPs and user space applications is identical to the behavior of AER.

5.6.2.7.1.6 CRDUMP

CRDUMP feature allows for taking an automatic snapshot of the device CR-Space in case the device's FW/HW fails to function properly.

Snapshots Triggers:

The snapshot is triggered after firmware detects a critical issue, requiring a recovery flow.

This snapshot can later be investigated and analyzed to track the root cause of the failure.

Currently, only the first snapshot is stored, and is exposed using a temporary virtual file. The virtual file is cleared upon driver reset.

When a critical event is detected, a message indicating CRDUMP collection will be printed to the Linux log. User should then back up the file pointed to in the printed message. The file location format is: **/proc/driver/mlx5_core/crdump/<pci address>**.

Snapshot should be copied by Linux standard tool for future investigation.

5.6.2.7.1.7 Firmware Tracer

This mechanism allows for the device's firmware/hardware to log important events into the event tracing system (/sys/kernel/debug/tracing) without requiring any NVIDIA tool.

> ⚠ To be able to use this feature, trace points must be enabled in the kernel.

This feature is enabled by default, and can be controlled using sysfs commands.

- To disable the feature:

```
echo 0 > /sys/kernel/debug/tracing/events/mlx5/fw_tracer/enable
```

- To enable the feature:

```
echo 1 > /sys/kernel/debug/tracing/events/mlx5/fw_tracer/enable
```

- To view FW traces using vim text editor:

```
vim /sys/kernel/debug/tracing/trace
```

5.6.2.8 Docker Containers

On Linux, Docker uses resource isolation of the Linux kernel, to allow independent "containers" to run within a single Linux kernel instance.
Docker containers are supported on MLNX_OFEDusing Docker runtime. Virtual RoCE and InfiniBand devices are supported using SR-IOV mode.

Currently, RDMA/RoCE devices are supported in the modes listed in the following table.

Orchestration and Clustering Tool	Version	Networking Mode	Link Layer	Virtualization Mode
Docker	Docker Engine 17.03 or higher	SR-IOV using SR-IOV plugin along with docker run wrapper tool	InfiniBand and Ethernet	SR-IOV
Kubernetes	Kubernetes 1.10.3 or higher	SR-IOV using device plugin, and using SR- IOV CNI plugin	InfiniBand and Ethernet	SR-IOV
		VXLAN using IPoIB bridge	InfiniBand	Shared HCA

5.6.2.8.1 Docker Using SR-IOV

This setup enables Docker containers to operate with SR-IOV (Single Root I/O Virtualization) for high-performance RDMA networking. It supports both InfiniBand and Ethernet link layers and is compatible with ConnectX-4, ConnectX-5, and ConnectX-6 adapter cards.

To ensure proper isolation of virtual RDMA devices within containers, the **docker_rdma_sriov** tool must be used.

Resources:

- SR-IOV Networking Plugin
- Container Tools Installer (includes **docker_rdma_sriov**)
- Docker RDMA SR-IOV Networking with ConnectX-4/ConnectX-5/ConnectX-6

5.6.2.8.2 Kubernetes Using SR-IOV

To enable RDMA in a Kubernetes environment with SR-IOV networking, two key components are required:

5.6.2.8.2.1 RDMA Device Plugin

This plugin is responsible for exposing RDMA-capable devices to Kubernetes Pods. It ensures that RDMA resources (e.g., InfiniBand or RoCE VFs) are made available and consumable through Kubernetes scheduling.

5.6.2.8.2.2 SR-IOV CNI Plugin

The SR-IOV Container Network Interface (CNI) plugin provisions Virtual Function (VF) network devices into the Pods. When used in SR-IOV mode, the plugin handles:

- SR-IOV enablement
- GUID and MAC address configuration
- Privilege mode setup
- Trust mode setup

The plugin also manages VF allocation during Pod scheduling based on requests made via Kubernetes resource definitions.

5.6.2.8.3 Kubernetes with Shared HCA

A single RDMA device (HCA) can be shared among multiple Pods running on the same Kubernetes worker node. This is typically achieved using user-defined networks based on VXLAN or VETH interfaces.

These virtual networking devices allow multiple Pods to access the same physical HCA while maintaining isolation and flexibility in network topology.

This approach differs from SR-IOV, as it does not require allocating dedicated Virtual Functions (VFs) per Pod.

5.6.2.9 HPC-X

For detailed information about HPC-X®, refer to the HPC-X User Manual.

5.6.2.10 Fast Driver Unload

This feature enables optimizing mlx5 driver teardown time in shutdown and kexec flows.

The fast driver unload is disabled by default. To enable it, the **prof_sel** module parameter of mlx5_core module should be set to 3.

6 DOCA Reference Applications

This page provides an overview of the DOCA reference applications implemented on top of NVIDIA® BlueField®.

All the DOCA reference applications described in this section are governed under the BSD-3 software license agreement.

6.1 Introduction

DOCA reference applications are an educational resource provided as a guide on how to program on the NVIDIA BlueField networking platform using DOCA API.

> ⚠ The reference applications are not production-ready and should be used as reference only.

For instructions regarding the development environment and installation, refer to the DOCA Developer Guide and the DOCA Installation Guide for Linux respectively.

> ℹ For questions, comments, and feedback, please contact us at doca-feedback@nvidia.com.

6.1.1 Installation

DOCA applications are installed under **/opt/mellanox/doca/applications** with each application having its own dedicated folder. Each directory contains the source code and compilation files for the matching application.

6.1.2 Prerequisites

The DOCA SDK references (samples and applications) require the use of meson, with a minimal version requirement of 0.61.2. This version is already included as part of the packages provided by most Linux distributions, hence it is recommended to simply install it from packages:

- Ubuntu/Debian:

```
$ sudo apt install meson
```

- RHEL:

```
$ sudo yum install meson
```

In the rare case in which the version provided by the distribution is lower than the minimal required version of 0.61.2, one can install meson directly through pip instead of through packages:

```
$ sudo pip3 install meson==0.61.2
```

Unlike the package installation, in the case of installation through pip, users should also remember to install their compiler of choice as it won't be installed automatically.

6.1.3 Compilation

As applications are shipped alongside their sources, developers may want to modify some of the code during their development process and then recompile the applications. The files required for the compilation are the following:

- **/opt/mellanox/doca/applications/meson.build** – main compilation file for a project that contains all the applications
- **/opt/mellanox/doca/applications/meson_options.txt** – configuration file for the compilation process
- **/opt/mellanox/doca/applications/<application_name>/meson.build** – application-specific compilation definitions

To recompile all the reference applications:

1. Move to the applications directory:

```
cd /opt/mellanox/doca/applications
```

2. Prepare the compilation definitions:

```
meson /tmp/build
```

3. Compile all the applications:

```
ninja -C /tmp/build
```

> The generated applications are located under the **/tmp/build/** directory, using the following path **/tmp/build/<application_name>/doca_<application_name>** .

> Compilation against DOCA's SDK relies on environment variables which are automatically defined per user session upon login. For more information, please refer to section "Meson Complains About Missing Dependencies" in the NVIDIA DOCA Troubleshooting Guide.

6.1.4 Developer Configurations

When recompiling the applications, meson compiles them by default in "debug" mode. Therefore, the binaries would not be optimized for performance as they would include the debug symbol. For comparison, the application binaries shipped as part of DOCA's installation are compiled in "release" mode. To compile the applications in something other than debug, please consult Meson's configuration guide.

The applications also offer developers the ability to use the DOCA log's **TRACE** level (**DOCA_LOG_TRC**) on top of the existing DOCA log levels. Enabling the **TRACE** log level during

compilation activates various developer log messages left out of the release compilation. Activating the **TRACE** log level may be done through **enable_trace_log** in the **meson_options.txt** file, or directly from the command line:

1. Prepare the compilation definitions to use the trace log level:

```
meson /tmp/build -Denable_trace_log=true
```

2. Compile the applications:

```
ninja -C /tmp/build
```

6.2 Application Use of DOCA Libs

The following table maps DOCA reference applications to the libraries they make use of.

Application Category	Application	Library Category											
		BareMetal/Virtualized Cloud					Secure Cloud Gateway		Cloud Storage	Monitoring	Streaming	HPC	
		Flow	DPA	DMA	Flex IO SDK	PCC	App Shield	SHA	Compress	Telemetry	GPUNetIO	Comch	UCX
Network	Ethernet L2 Forwarding												
	GPU Packet Processing										✔		
	Simple Forward VNF	✔											
	Switch	✔											
	IP Fragmentation	✔											
	Accelerated UPF	✔											
	Bifurcated Driver Model	✔											
	DPU GPU Remote Offload										✔	✔	
Security	App Shield Agent						✔			✔			

Application Category	Application	Library Category											
		BareMetal/Virtualized Cloud					Secure Cloud Gateway		Cloud Storage	Monitoring	Streaming	HPC	
		Flow	DPA	DMA	FlexIO SDK	PCC	AppShield	SHA	Compress	Telemetry	GPUNetIO	Comch	UCX
	East-west Overlay Encryption												
	IPsec Security Gateway	✔											
	PSP Gateway	✔											
	Secure Channel											✔	
	YARA Inspection						✔			✔			
Data Path Acceleration	DPA All-to-all		✔										
	PCC					✔							
Storage	DMA Copy			✔								✔	
	File Compression								✔			✔	
	File Integrity							✔				✔	
	NVMe Emulation	✔	✔	✔	✔							✔	
	Storage Zero Copy											✔	
HPC	UROM RDMO												✔
Telemetry	Time Sync		✔									✔	
	Telemetry OpenTelemetry									✔			
	Telemetry Traceback									✔			

6.3 DOCA App Shield Agent Application Guide

6.3.1 Introduction

The App Shield Agent uses the DOCA App Shield library to monitor processes running on the host system. This security feature allows for the detection of core process corruption from a trusted, independent DPU, providing an intrusion detection system (IDS) that otherwise cannot be achieved from within the host itself.

The DOCA App Shield library enables the DPU to directly read, analyze, and verify the memory of the host, whether it's a bare metal system or a virtual machine. By using this library, the application calculates cryptographic hashes of non-writable memory pages, including those that are unloaded, for a specific process and its associated libraries. It then periodically re-verifies these pages to identify any unauthorized changes.

After each check, the application outputs a pass or fail attestation result, continuing the process until the first failure is detected. These results are logged to the console and also sent to the DOCA Telemetry Service (DTS) via inter-process communication (IPC).

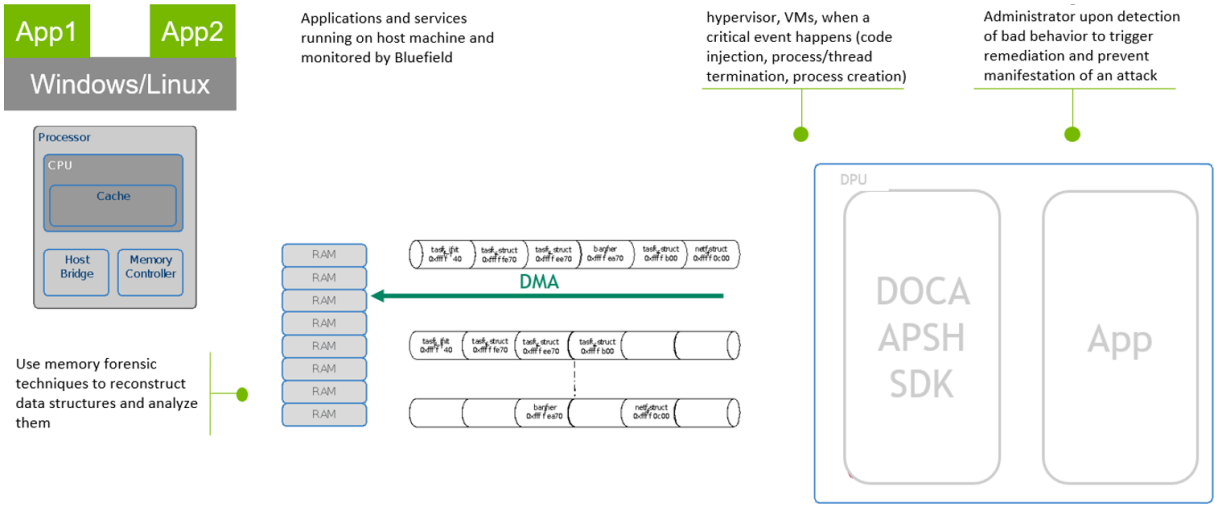
This guide illustrates how to implement secure process monitoring using the DOCA App Shield library, taking advantage of the BlueField DPU's capabilities, such as hardware-accelerated DMA and root-of-trust integrity enforcement.

6.3.2 System Design

The App Shield agent is designed to operate independently on the DPU's Arm core without impacting the host system's performance.

The host's role is minimal and primarily involves configuring the monitoring of a new process when necessary. This configuration includes generating the required ZIP and JSON files, which are then passed to the DPU. This setup is performed at the initial stage, often referred to as "time 0," when the host is still in a "safe" state.

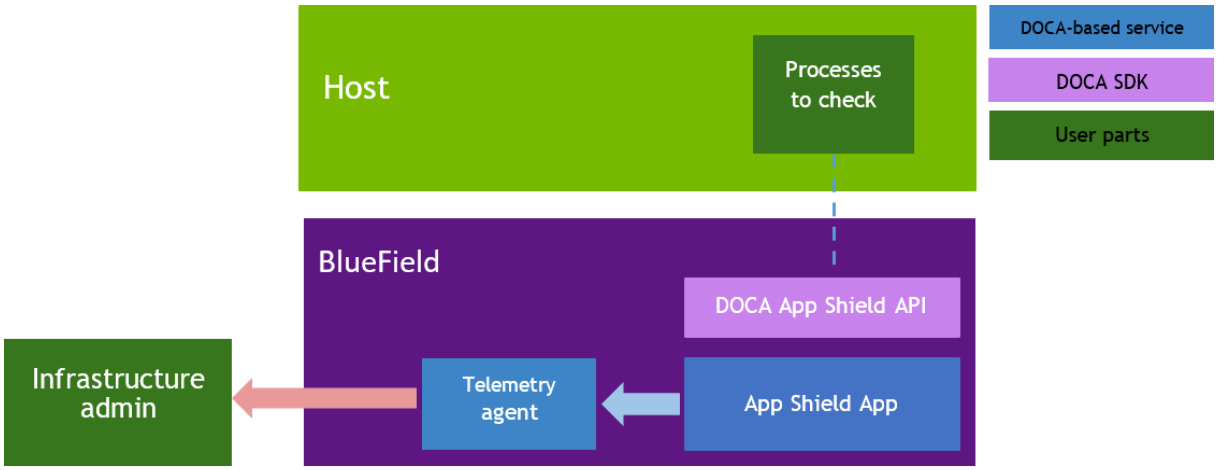
To generate these files, use the **`doca_apsh_config.py`** tool provided by DOCA App Shield. For detailed information, refer to DOCA App Shield.



6.3.3 Application Architecture

The user creates three mandatory files using the DOCA tool **doca_apsh_config.py** and copies them to the DPU. The application can report attestation results to the:

- File
- Terminal
- DTS



1. The files are generated by running **doca_apsh_config.py** on the host against the process at time zero.

   > ⚠ The actions 2-5 recur at regular time intervals.

2. The App Shield agent requests new attestation from DOCA App Shield library.

3. The DOCA App Shield library creates a new attestation:
    a. Scans and hashes process memory pages (that are currently in use).
    b. Compares the hash to the original hash.
    c. Creates attestation for each lib/exe involved in the process. Each of attestation includes the number of valid pages and the number of pages.
4. The App Shield agent searches each attestation for inconsistency between number of used pages and number of valid pages.
5. The App Shield agent reports results with a timestamp and scan count to:
    a. Local telemetry files – a folder and files representing the data a real DTS would have received. These files are used for the purposes of this example only as normally this data is not exported into user-readable files.
    b. DOCA log (without scan count).
    c. DTS IPC interface (even if no DTS is active).
6. The App Shield agent exits on first attestation failure.

6.3.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA App Shield
- DOCA Telemetry

Refer to their respective programming guide for more information.

6.3.5 Compiling the Application

> Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

> For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The sources of the application can be found under the application's directory: **/opt/mellanox/doca/applications/app_shield_agent/** .

6.3.5.1 Compiling All Applications

All DOCA applications are defined within a single meson project. Therefore, by default, the compilation process includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_app_shield_agent is created under **/tmp/build/app_shield_agent/**.

6.3.5.2 Compiling Only the Current Application

To build only the App Shield Agent application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_app_shield_agent=true
ninja -C /tmp/build
```

doca_app_shield_agent is created under **/tmp/build/app_shield_agent/**.

Alternatively, you can set the flags in the **meson_options.txt** file instead of providing them in the compilation command line:

1. Edit the following flags in **/opt/mellanox/doca/applications/meson_options.txt**:
   - Set **enable_all_applications** to **false**
   - Set **enable_app_shield_agent** to **true**
2. Run the following compilation commands:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_app_shield_agent is created under **/tmp/build/app_shield_agent/**.

6.3.6 Running the Application

6.3.6.1 Prerequisites

1. Configure the BlueField's firmware.
   a. On the BlueField system, configure the PF base address register and NVMe emulation. Run:

```
dpu> mlxconfig -d /dev/mst/mt41686_pciconf0 s PF_BAR2_SIZE=2 PF_BAR2_ENABLE=1
 NVME_EMULATION_ENABLE=1
```

   b. Perform a BlueField system reboot for the **mlxconfig** settings to take effect.
   c. You may verify these configurations using the following command:

```
dpu> mlxconfig -d /dev/mst/mt41686_pciconf0 q | grep -E "NVME|BAR"
```

2. Download target system (host/VM) symbols.
   - For Ubuntu:

```
host> sudo tee /etc/apt/sources.list.d/ddebs.list << EOF
deb http://ddebs.ubuntu.com/ $(lsb_release -cs) main restricted universe multiverse
deb http://ddebs.ubuntu.com/ $(lsb_release -cs)-updates main restricted universe multiverse
```

```
deb http://ddebs.ubuntu.com/ $(lsb_release -cs)-proposed main restricted universe multiverse
EOF
host> sudo apt install ubuntu-dbgsym-keyring
host> sudo apt-get update
host> sudo apt-get install linux-image-$(uname -r)-dbgsym
```

- For CentOS:

```
host> yum install --enablerepo=base-debuginfo kernel-devel-$(uname -r) kernel-debuginfo-$(uname -r)
kernel-debuginfo-common-$(uname -m)-$(uname -r)
```

- No action is needed for Windows

3. Perform IOMMU passthrough. This stage is only necessary if IOMMU is not enabled by default (for example, if the host is using an AMD CPU).

> ⚠ Skip this step if you are not sure whether it is needed. Return to it only if DMA fails with a message similar to the following in **dmesg**:
>
> ```
> host> dmesg
> [ 3839.822897] mlx5_core 0000:81:00.0: AMD-Vi: Event logged [IO_PAGE_FAULT domain=0x0047
> address=0x2a0aff8 flags=0x0000]
> ```

   a. Locate your OS's `grub` file (most likely **/boot/grub/grub.conf**, **/boot/grub2/grub.cfg**, or **/etc/default/grub**) and open it for editing. Run:

   ```
   host> vim /etc/default/grub
   ```

   b. Search for the line defining **GRUB_CMDLINE_LINUX_DEFAULT** and add the argument **iommu=pt**. For example:

   ```
   GRUB_CMDLINE_LINUX_DEFAULT="iommu=pt <intel/amd>_iommu=on"
   ```

   c. Run:

   > ⚠ Prior to performing a power cycle, make sure to do a graceful shutdown.

   - For Ubuntu:

   ```
   host> sudo update-grub
   host> ipmitool power cycle
   ```

   - For CentOS:

   ```
   host> grub2-mkconfig -o /boot/grub2/grub.cfg
   host> ipmitool power cycle
   ```

   - For Windows targets, turn off Hyper-V capability.

4. Prepare target:
   a. Install DOCA on the target system.
   b. Create the ZIP and JSON files. Run:

```
target-system> cd /opt/mellanox/doca/tools/
target-system> python3 doca_apsh_config.py --pid <pid-of-process-to-monitor> --os <windows/linux>
--path <path to dwarf2json executable  or pdbparse-to-json.py>
target-system> cp /opt/mellanox/doca/tools/*.* <shared-folder-with-baremetal>
dpu> scp <shared-folder-with-baremetal>/* <path-to-app-shield-binary>
```

If the target system does not have DOCA installed, the script can be copied from the BlueField.

The required **dwaf2json** and **pdbparse-to-json.py** are not provided with DOCA.

> ⚠ If the kernel and process **.exe** have not changed, there no need to redo this step.

6.3.6.2 Application Execution

1. The App Shield Agent application is provided in source form, so it needs to be compiled before it can be executed.
   a. Application usage instructions:

```
Usage: doca_app_shield_agent [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -p, --pid                         Process ID of process to be attested
  -e, --ehm <path>                  Exec hash map path
  -m, --memr <path>                 System memory regions map
  -f, --vuid                        VUID of the System device
  -d, --dma                         DMA device name
  -o, --osym <path>                 System OS symbol map path
  -s, --osty <windows|linux>        System OS type - windows/linux
  -t, --time <seconds>              Scan time interval in seconds
```

   b. CLI example for running the application on the BlueField:

```
./doca_app_shield_agent -p 13577 -e hash.zip -m mem_regions.json -o symbols.json -f
MT2125X03335MLNXS0D0F0VF1 -d mlx5_0 -t 3 -s linux
```

> ⚠ All used identifiers (**-f** , **-p** and **-d** flags) should match the identifier of the desired devices and processes.

6.3.6.3 Command Line Flags

6.3.6.3.1 General Flags

Short Flag	Long Flag	Description
-h	**--help**	Prints a help synopsis and exits
-v	**--version**	Prints program version information and exits

Short Flag	Long Flag	Description
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.3.6.3.2 Program Flags

Short Flag	Long Flag	Description
`p`	`pid`	PID of the process to be attested
`e`	`ehm`	Path to the pre-generated `hash.zip` file transferred from the host
`m`	`memr`	Path to the pre-generated `mem_regions.json` file transferred from the host

Short Flag	Long Flag	Description

f	pcif	System PCIe function vendor unique identifier (VUID) of the VF/PF exposed to the target system. Used for DMA operations.
		To obtain this argument, run:
		`target-system> lspci -vv | grep "\[VU\] Vendor specific:"`
		Example output:
		`[VU] Vendor specific: MT2125X03335MLNXS0D0F0`
		`[VU] Vendor specific: MT2125X03335MLNXS0D0F1`
		Two VUIDs are printed for each DPU connected to the target system. The first is of the DPU on `pf0` and the second is of the DPU on port `pf1`.
		⚠ Running this command on the DPU outputs VUIDs with an additional "EC" string in the middle. You must remove the "EC" to arrive at the correct VUID.
		The VUID of a VF allocated on PF0/1 is the VUID of the PF with an additional suffix, `VF<vf-number>`, where `vf-number` is the VF index +1.
		For example, for the output in the example above:
		• PF0 VUID = MT2125X03335MLNXS0D0F0
		• PF1 VUID = MT2125X03335MLNXS0D0F1
		• VUID of VF0 on PF0 = MT2125X03335MLNXS0D0F0VF1
		VUIDs are persistent even on reset.

d	dma	DMA device name to use
o	osym	Path to the pre-generated `symbols.json` file transferred from the host
s	osty	OS type (`windows` or `linux`) of the system where the process is running
t	time	Number of seconds to sleep between scans

6.3.6.4 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.3.7 Application Code Flow

1. Parse application argument.
    a. Initialize arg parser resources and register DOCA general parameters.

```
doca_argp_init();
```

    b. Register application parameters.

```
register_apsh_params();
```

    c. Parse the arguments.

```
doca_argp_start();
```

2. Initialize DOCA App Shield lib context.
    a. Create lib context.

```
doca_apsh_create();
```

    b. Set DMA device for lib.

```
doca_devinfo_list_create();
doca_dev_open();
doca_devinfo_list_destroy();
doca_apsh_dma_dev_set();
```

    c. Start the context

```
doca_apsh_start();
apsh_system_init();
```

3. Initialize DOCA App Shield lib system context handler.
    a. Get the representor of the remote PCIe function exposed to the system.

```
doca_devinfo_remote_list_create();
doca_dev_remote_open();
doca_devinfo_remote_list_destroy();
```

    b. Create and start the system context handler.

```
doca_apsh_system_create();
doca_apsh_sys_os_symbol_map_set();
doca_apsh_sys_mem_region_set();
doca_apsh_sys_dev_set();
doca_apsh_sys_os_type_set();
doca_apsh_system_start();
```

4. Find target process by **pid**.

```
doca_apsh_processes_get();
```

5. Telemetry initialization.

```
telemetry_start();
```

    a. Initialize a new telemetry schema.
    b. Register attestation type event.
    c. Set up output to file (in addition to default IPC).
    d. Start the telemetry schema.
    e. Initialize and start a new DTS source with the **gethostname()** name as source ID.

6. Get initial attestation of the process.

```
doca_apsh_attestation_get();
```

7. Loop until attestation validation fail.

```
doca_apsh_attst_refresh();
/* validation logic */
doca_telemetry_exporter_source_report();
DOCA_LOG_INFO();
sleep();
```

8. DOCA App Shield Agent destroy.

```
doca_apsh_attestation_free();
doca_apsh_processes_free();
doca_apsh_system_destroy();
doca_apsh_destroy();
doca_dev_close();
doca_dev_remote_close();
```

9. Telemetry destroy.

```
telemetry_destroy();
```

10. Arg parser destroy.

```
doca_argp_destroy();
```

6.3.8 References

- **/opt/mellanox/doca/applications/app_shield_agent/**

6.4 DOCA Bifurcated Driver Model Reference Application

This page provides an example of a Bifurcated Driver Model implementation using a multi-port eSwitch on top of the NVIDIA® BlueField® DPU.

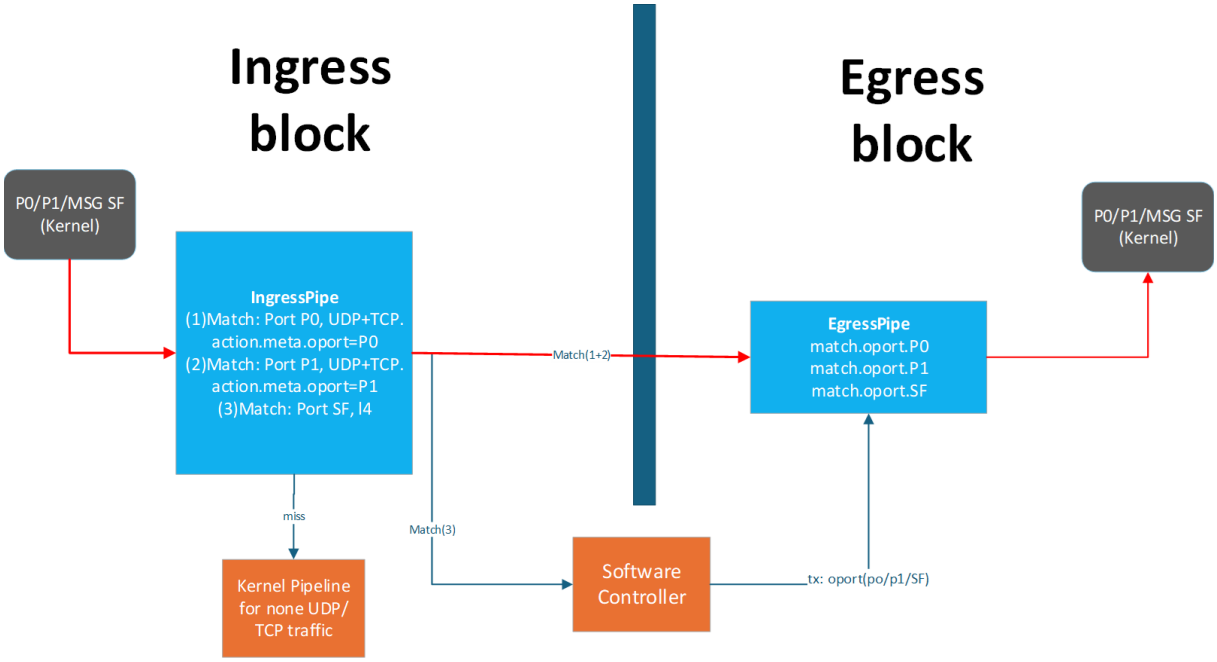
6.4.1 Introduction

The Bifurcated Driver Model leverages hardware capabilities to split incoming network traffic based on flow or packet type, allowing the standard Linux kernel driver and a DOCA-based application to operate simultaneously on the same network interface.

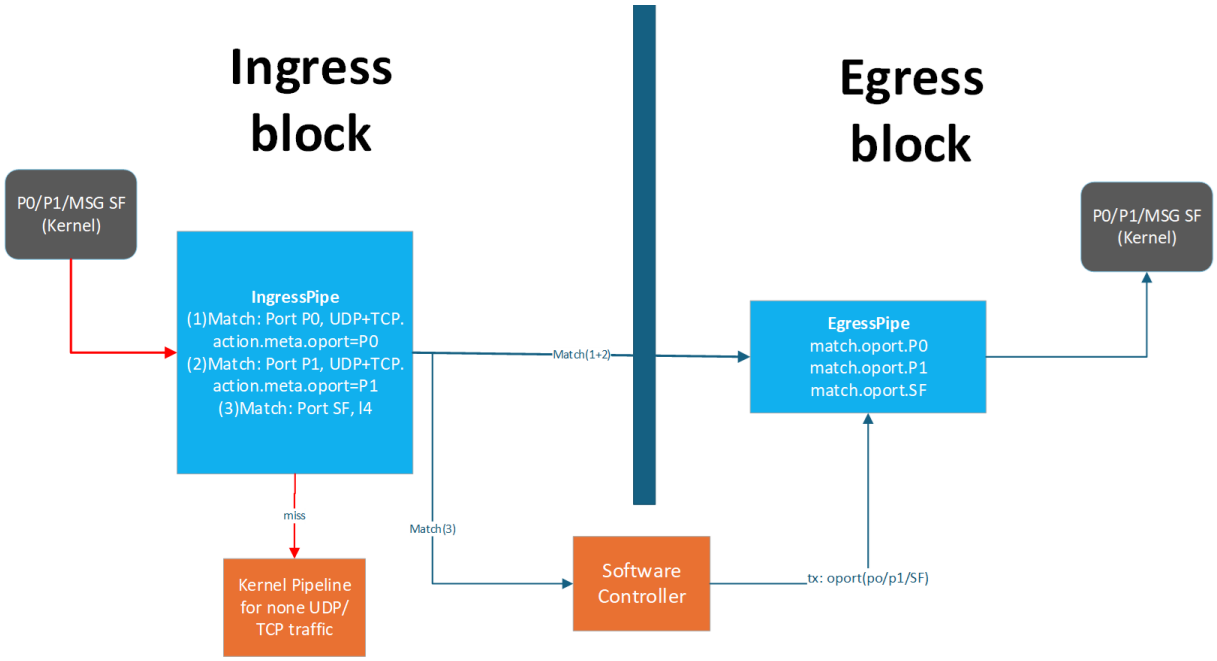
This architecture enables users to retain default kernel behavior for specific flows while implementing custom software logic for others. This application utilizes the DOCA Flow library to demonstrate efficient traffic forwarding between Physical Ports (wire) and Linux Ports (kernel).

6.4.2 System Design

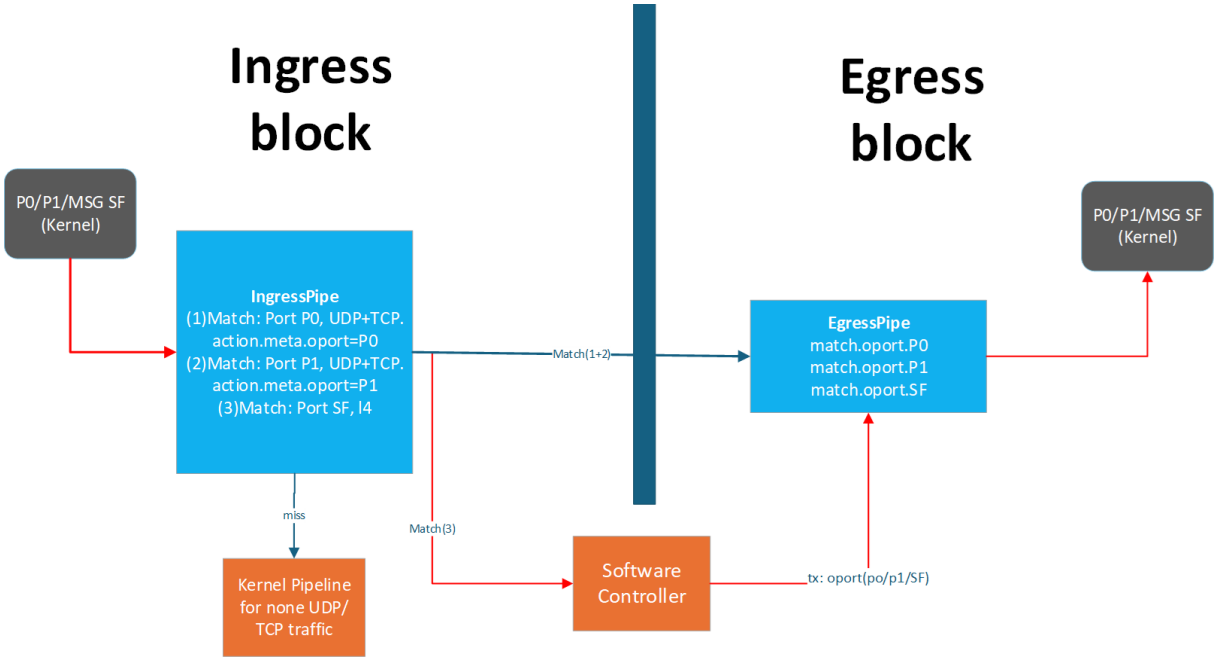
Port 0 ↔ Port 1 (Data Plane): High-volume traffic matching defined DOCA Flow rules are forwarded directly between Physical Port 0 and Physical Port 1. This bypasses the kernel network stack, remaining in the data plane for fast processing.



Port[0-1] ↔ Kernel Pipeline (Control Plane): External traffic coming from Physical Port 0 or Physical Port 1 that is *not* TCP/UDP (e.g., ARP, ICMPv6, or specific control protocols) is missed and steered to the Linux Driver.



Kernel Port ↔ Kernel Port (Messaging): Internal Messaging traffic from the Linux Port is matched and steered to the Linux Representor Port, utilizing RSS for efficient kernel-space application processing.



6.4.3 Application Architecture

The Bifurcated Driver Model Reference Application consists of core logic that manages the DOCA Flow pipes and entries to implement traffic bifurcation.

1. Initialization: The application initializes DPDK and DOCA Flow resources, including ports and queues.
2. DOCA Flow Switch Configuration: setting DOCA Flow in a switch mode.
3. Port Setup: Creating DOCA Flow switch ports corresponding to the physical and Linux representor ports.
4. Default miss behavior: Unmatched packets are forwarded to the kernel, ensuring control plane protocols receive proper kernel processing.
5. Pipe Creation: Creates two pipes to implement the bifurcation logic:
    a. Egress Pipe (EGRESS domain): Matches on packet metadata (pkt_meta) and forwards packets to the corresponding port. Contains three entries:
        - $Metadata = PORT_P0 \rightarrow Forward\ to\ Physical\ Port\ 0$
        - $Metadata = PORT_LINUX \rightarrow Forward\ to\ Linux\ Port$
        - $Metadata = PORT_P1 \rightarrow Forward\ to\ Physical\ Port\ 1$
    b. Ingress Classifier Pipe (DEFAULT domain): Routes traffic based on source port_id and outer L4 protocol type, sets packet metadata, and forwards to either the egress pipe or RSS queues. Contains nine entries:
        - $P0 + TCP/UDP \rightarrow Set\ metadata\ to\ P1, forward\ to\ egress\ pipe$
        - $P1 + TCP/UDP \rightarrow Set\ metadata\ to\ P0, forward\ to\ egress\ pipe$
        - $PORT_LINUX \rightarrow Forward\ to\ RSS\ queue$

6. Monitoring Configuration: Each pipe entry is configured with a non-shared counter resource to track packet statistics. The ingress root classifier pipe also enables a miss counter to monitor packets forwarded to the kernel.
7. Entry Processing: All flow entries are processed and offloaded to hardware.
8. Packet Processing: The main application loop demonstrates Linux port packet handling by receiving packets from queue 0, setting appropriate metadata, and transmitting them back through the switch port.
9. Statistics Collection: Upon completion, the application queries and displays per-entry packet counters for both ingress and egress pipes, along with the ingress miss counter, providing visibility into traffic patterns.

6.4.4 DOCA Libraries

This application leverages the following DOCA library:

- DOCA Flow – The core library used to program the hardware eSwitch to create the flow rules for traffic steering and bifurcation.

Refer to the respective programming guide for more information.

6.4.5 Compiling the Application

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for the application is located in the following directory:

```
/opt/mellanox/doca/applications/bifurcated_driver_model/
```

6.4.5.1 Compiling All Applications

To build all DOCA applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_bifurcated_driver_model is created under **/tmp/build/applications/bifurcated_driver_model/** .

6.4.5.2 Recompiling Only the Current Application

To directly build only the Bifurcated Driver Model application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_bifurcated_driver_model=true
ninja -C /tmp/build
```

doca_bifurcated_driver_model is created under **/tmp/build/applications/bifurcated_driver_model/** .

6.4.6 Running the Application

The application is based on DOCA Flow and DPDK. Therefore, the user is required to allocate huge pages.

```
$ echo '1024' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
$ sudo mkdir /mnt/huge
$ sudo mount -t hugetlbfs -o pagesize=2M nodev /mnt/huge
```

6.4.6.1 Application Execution

- Application usage instructions:

  The application is provided in source form, requiring compilation before execution.

  ```
  Usage: doca_bifurcated_driver_model [DPDK Flags] -- [DOCA Flags] [Program Flags]
  DOCA Flags:
    -h, --help                      Print a help synopsis
    -v, --version                   Print program version information
    -l, --log-level                 Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
  30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
    --sdk-log-level                 Set the SDK (numeric) log level for the program <10=DISABLE,
  20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
    --log-filter                    Filter logs from specific modules, separated by comma
    -j, --json <path>               Parse command line flags from an input json file
  Program Flags:
    -a, --device                    device
    -r, --rep                       device representor
    -exp, --expert-mode             set expert mode
  ```

- This usage printout can be printed to the command line using the -h (or --help) options:

  ```
  ./doca_bifurcated_driver_model -- -h
  ```

  For additional information, refer to section "Command Line Flags".

- CLI example for running the application on BlueField:

  ```
  cd /opt/mellanox/doca/applications/
  ./bifurcated_driver_model/doca_bifurcated_driver_model
   -- -l 60 -a pci/08:00.0 -r pci/08:00.0,pf0sf0 -r pci/08:00.1
  ```

  SFs must be enabled according to the BlueField Scalable Functions.

- Before creating SFs on a specific physical port, it is important to verify the encap mode on the respective PF FDB. The default mode is basic. To check the encap mode, run:

  ```
  cat /sys/class/net/p0/compat/devlink/encap
  ```

  In this case, disable encap on the PF FDB before creating the SFs by running:

```
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode legacy
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode legacy
echo none > /sys/class/net/p0/compat/devlink/encap
echo none > /sys/class/net/p1/compat/devlink/encap
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode switchdev
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode switchdev
```

If the encap mode is set to **basic** then the application fails upon initialization.

- The flag **-a pci/08:00.0** with representors **-r pci/08:00.0,pf0sf0 -r pci/08:00.1** is mandatory for proper usage of the application.
  - Modifying this flag will result in an unexpected behavior as only 2 ports are supported.
  - The SF number is arbitrary and configurable.
- The SF numbers must match the desired SF devices.
  - The device identifiers must match the desired network devices.
  - For additional information regarding the command line syntax of the device and device representor identifiers, refer to the matching section in the DOCA Arg Parser.
  - For more information, refer to section "Running DOCA Application on Host" in DOCA Virtual Functions.

6.4.6.2 Command Line Flags

6.4.6.2.1 General Flags

Short Flag	Long Flag	Description
-h	**--help**	Prints a help synopsis and exits
-v	**--version**	Prints program version information and exits
-l	**--log-level**	Sets the numeric log level for the application: • **10** – DISABLE • **20** – CRITICAL • **30** – ERROR • **40** – WARNING • **50** – INFO • **60** – DEBUG • **70** – TRACE (requires compilation with TRACE support)
N/A	**--sdk-log-level**	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	**--log-filter**	Filters logs from specific modules (comma-separated list)
-j	**--json**	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.4.6.2.2 Program Flags

Flag Type	Short Flag	Long Flag	Description
Program flags	`a`	`device`	Device identifier
	`r`	`rep`	Device representor identifiers
	`exp`	`expert-mode`	Set expert mode

6.4.6.3 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.4.7 Application Code Flow

The application initializes resources, sets up the flow pipes, and enters a processing loop.

1. Initialize logging:
    a. **`doca_log_backend_create_standard();`**
    b. **`doca_log_backend_create_with_file_sdk();`**
    c. **`doca_log_backend_set_sdk_level();`**
2. Argument Parsing Initialization:
    a. **`doca_argp_init()`** : Initialize arg parser.
    b. **`register_doca_flow_switch_params()`** : Register switch applications parameters.
3. Parse all command line arguments.
    a. **`doca_argp_start();`**
4. DOCA Flow devices initialization:
    a. **`init_doca_flow_devs();`**
5. DPDK & Port Initialization:
    a. **`dpdk_queues_and_ports_init();`**
6. DOCA Flow & Application Setup:
    a. **`bifurcated_driver_model_init():`**
        i. Initializes DOCA Flow:
        init_doca_flow();
        ii. Creates DOCA ports for all physical and Linux representor ports:
        init_doca_flow_switch_ports();
        iii. Set default miss behavior:
        doca_flow_port_update_default_miss();
        iv. Creates all necessary pipes: Ingress, and Egress pipes, defining the flow structure:
            1. **`create_egress_pipe();`**
            2. **`add_egress_pipe_entries();`**
            3. **`create_ingress_classifier_pipe();`**
            4. **`add_ingress_classifier_entries();`**

a. Data Plane (P0 ↔ P1): 4 entries for TCP/UDP forwarding
b. Control Plane (Linux → RSS): 5 entries for various protocols

7. Process flow entries:
    a. `doca_flow_entries_process();`
8. Main processing loop
    a. `handle_rx_tx_pkts()`
9. Statistics collection:
    a. `print_flow_statistics();`
        i. `doca_flow_resource_query_entry()` : Query and display ingress entry counters.
        ii. `doca_flow_resource_query_pipe_miss()` : Query and display ingress miss counter.
10. Cleanup and destroy:
    a. `stop_doca_flow_ports()`
    b. `doca_flow_destroy()`
    c. `dpdk_queues_and_ports_fini()`
    d. `dpdk_fini()`
    e. `destroy_doca_flow_devs()`
    f. `doca_argp_destroy()`

6.5 DOCA DMA Copy Application Guide

This guide provides an example of a DMA Copy implementation on top of NVIDIA® BlueField® DPU.

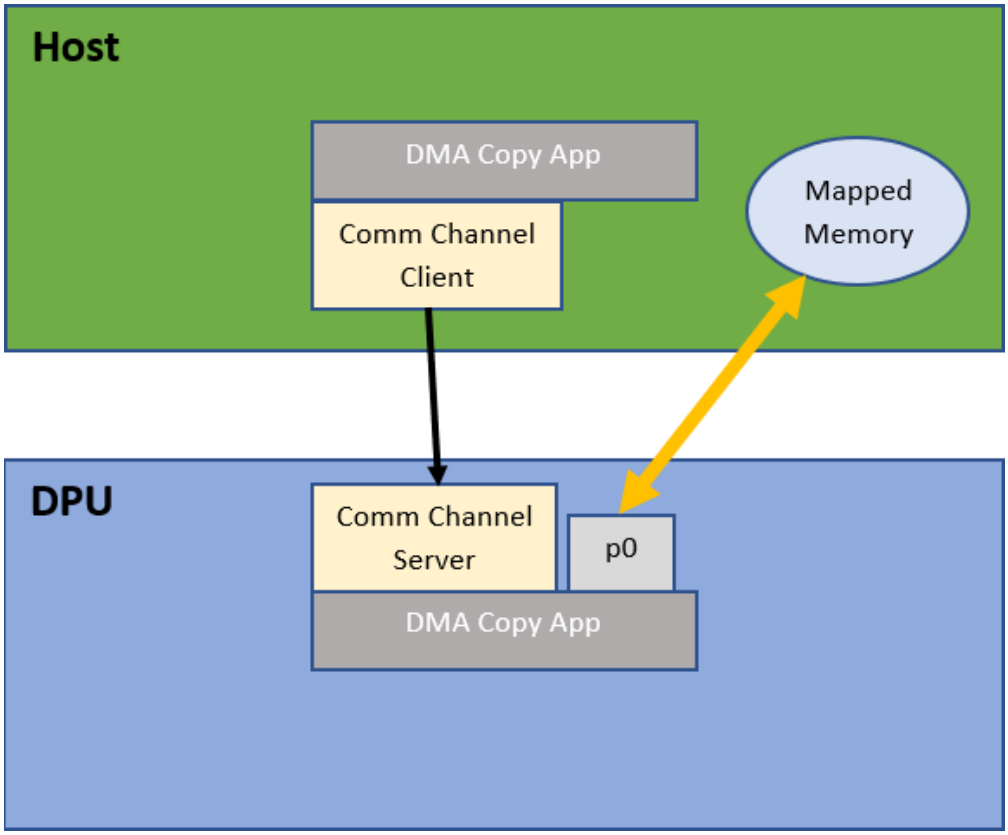
6.5.1 Introduction

DOCA DMA (direct memory access) Copy application transfers files (data path), up to the maximum supported size by the hardware, between the DPU and the x86 host using the DOCA DMA Library which provides an API to copy data between DOCA buffers using hardware acceleration, supporting both local and remote memory.

DOCA DMA allows complex memory copy operations to be easily executed in an optimized, hardware-accelerated manner.

6.5.2 System Design

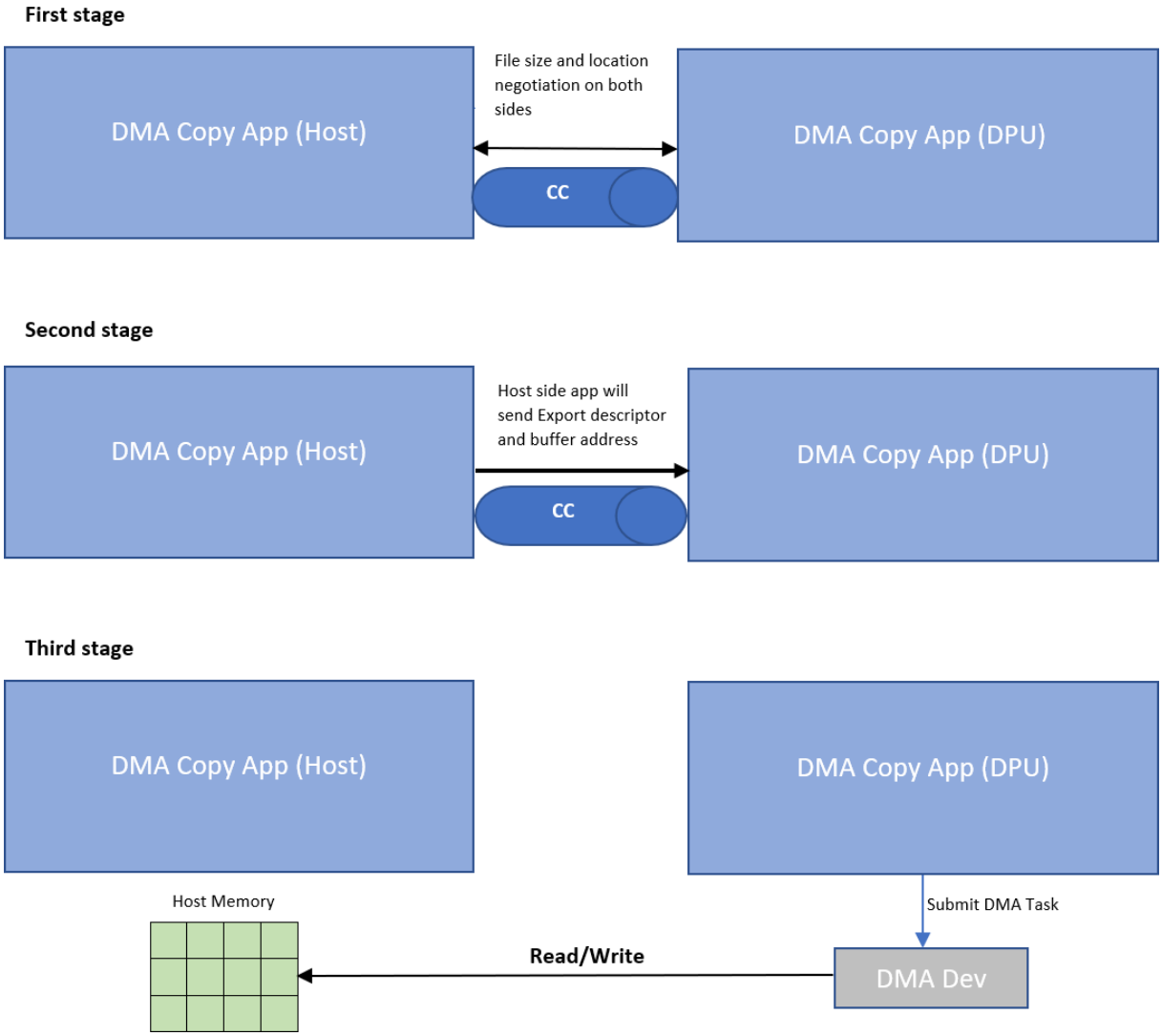
DOCA DMA Copy is designed to run on the instances of the BlueField DPU and x86 host. The DPU application must be the first to spawn as it opens the DOCA Comch server between the two sides on which all the necessary DOCA DMA library configuration files (control path) are transferred.



This reference application is designed for simplicity and clarity. Therefore, security-related requirements, such as isolation between the Host/DPU or among DPU processes, are out of scope and have not been addressed.

6.5.3 Application Architecture

DOCA DMA Copy runs on top of DOCA DMA to read/write directly from the host's memory without any user/kernel space context switches, allowing for a fast memory copy.



Flow:

1. The two sides initiate a short negotiation in which the file size and location are determined.
2. The host side creates the export descriptor with **doca_mmap_export_pci()** and sends it with the local buffer address and length on the Comch to the DPU side application.
3. The DPU side application uses the received export descriptor to create a remote memory map locally with **doca_mmap_create_from_export()** and the host buffer information to create a remote DOCA buffer.
4. From this point on, the DPU side application has all the necessary memory information and the DMA copy can take place.

6.5.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA DMA

- DOCA Comch

Refer to their respective programming guide for more information.

6.5.5 Compiling the Application

> Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

The installation of DOCA's reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

> For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The sources of the application can be found under the application's directory: **/opt/mellanox/doca/applications/dma_copy/** .

6.5.5.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

> **doca_dma_copy** is created under **/tmp/build/dma_copy/** .

6.5.5.2 Compiling Only the Current Application

To directly build only the DMA Copy application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_dma_copy=true
ninja -C /tmp/build
```

> **doca_dma_copy** is created under **/tmp/build/dma_copy/** .

Alternatively, one can set the desired flags in the **meson_options.txt** file instead of providing them in the compilation command line:

1. Edit the following flags in **/opt/mellanox/doca/applications/meson_options.txt** :
   - Set **enable_all_applications** to **false**

- Set **enable_dma_copy** to **true**

2. Run the following compilation commands:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

> **doca_dma_copy** is created under **/tmp/build/dma_copy/**.

6.5.6 Running the Application

6.5.6.1 Application Execution

The DMA Copy application is provided in source form. Therefore, a compilation is required before the application can be executed.

- Application usage instructions:

```
Usage: doca_dma_copy [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -f, --file                        Full path to file to be copied/created after a successful DMA copy
  -p, --pci-addr                    DOCA Comm Channel device PCI address
  -r, --rep-pci                     DOCA Comm Channel device representor PCI address (needed only on DPU)
```

> This usage printout can be printed to the command line using the **-h** (or **--help**) options:
>
> ```
> ./doca_dma_copy -h
> ```

> For additional information, refer to section "Command Line Flags".

- CLI example for running the application on the BlueField:

```
./doca_dma_copy -p 03:00.0 -r 3b:00.0 -f received.txt
```

> Both the DOCA Comch device PCIe address (**03:00.0**) and the DOCA Comch device representor PCIe address (**3b:00.0**) should match the addresses of the desired PCIe devices.

- CLI example for running the application on the host:

```
./doca_dma_copy -p 3b:00.0 -f send.txt
```

The DOCA Comch device PCIe address, **3b:00.0**, should match the address of the desired PCIe device.

6.5.6.2 Command Line Flags

6.5.6.2.1 General Flags

Short Flag	Long Flag	Description
-h	**--help**	Prints a help synopsis and exits
-v	**--version**	Prints program version information and exits
-l	**--log-level**	Sets the numeric log level for the application: • **10** – DISABLE • **20** – CRITICAL • **30** – ERROR • **40** – WARNING • **50** – INFO • **60** – DEBUG • **70** – TRACE (requires compilation with TRACE support)
N/A	**--sdk-log-level**	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	**--log-filter**	Filters logs from specific modules (comma-separated list)
-j	**--json**	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.5.6.2.2 Program Flags

Short Flag	Long Flag	Description
f	**file**	Full path to file to be copied/created after a successful copy. This is a mandatory flag.
p	**pci-addr**	DOCA Comch device PCIe address. This is a mandatory flag.

Short Flag	Long Flag	Description
r	rep-pci	DOCA Comch device representor PCIe address. ⚠ This is a mandatory flag only on the DPU.

6.5.6.3 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.5.7 Application Code Flow

1. Parse application argument.
    a. Initialize arg parser resources and register DOCA general parameters.

```
doca_argp_init();
```

    b. Register DMA Copy application parameters.

```
register_dma_copy_params();
```

    c. Parse the arguments.

```
doca_argp_start();
```

2. Initialize Comch endpoint.

```
init_cc();
```

    a. Create Comch endpoint.
    b. Parse user PCIe address for Comch device.
    c. Open Comch DOCA device.
    d. Parse user PCIe address for Comch device representor (on DPU side).
    e. Open Comch DOCA device representor (on DPU side).
    f. Set Comch endpoint properties.
3. Open the DOCA hardware device from which the copy would be made.

```
open_dma_device();
```

    a. Parse the PCIe address provided by the user.
    b. Create a list of all available DOCA devices.
    c. Find the appropriate DOCA device according to specific properties.
    d. Open the device.
4. Create all required DOCA core objects.

```
create_core_objects();
```

5. Initiate DOCA core objects.

```
init_core_objects();
```

6. Start host/DPU DMA Copy.
    a. Host side application:

```
host_start_dma_copy();
```

        i. Start negotiation with the DPU side application for the location and size of the file.
        ii. Allocate memory for the DMA buffer.
        iii. Export the memory map and send the output (export descriptor) to the DPU side application.
        iv. Send the host local buffer memory address and length on the Comch to the DPU side application.
        v. Wait for the DPU to notify that DMA Copy ended.
        vi. Close all memory objects.
        vii. Clean resources.

    b. DPU side application:

```
dpu_start_dma_copy();
```

        i. Start negotiation with the host side application for file location and size.
        ii. Allocate memory for the DMA buffer.
        iii. Receive the export descriptor on the Comch.
        iv. Create the DOCA memory map for the remote buffer on the host.
        v. Receive the host buffer information on the Comch.
        vi. Create two DOCA buffers, one for the remote (host) buffer and one for the local buffer.
        vii. Submit the DMA copy task.
        viii. Send a host message to notify that DMA copy ended.
        ix. Clean resources.
7. Destroy Comch.

```
destroy_cc();
```

8. Destroy DOCA core objects.

```
destroy_core_objects();
```

9. Arg parser destroy.

```
doca_argp_destroy();
```

6.5.8 References

- **/opt/mellanox/doca/applications/dma_copy/**

6.6 DOCA DPA All-to-all Application Guide

This guide explains all-to-all collective operation example when accelerated using the DPA in NVIDIA® BlueField®-3 DPU.

6.6.1 Introduction

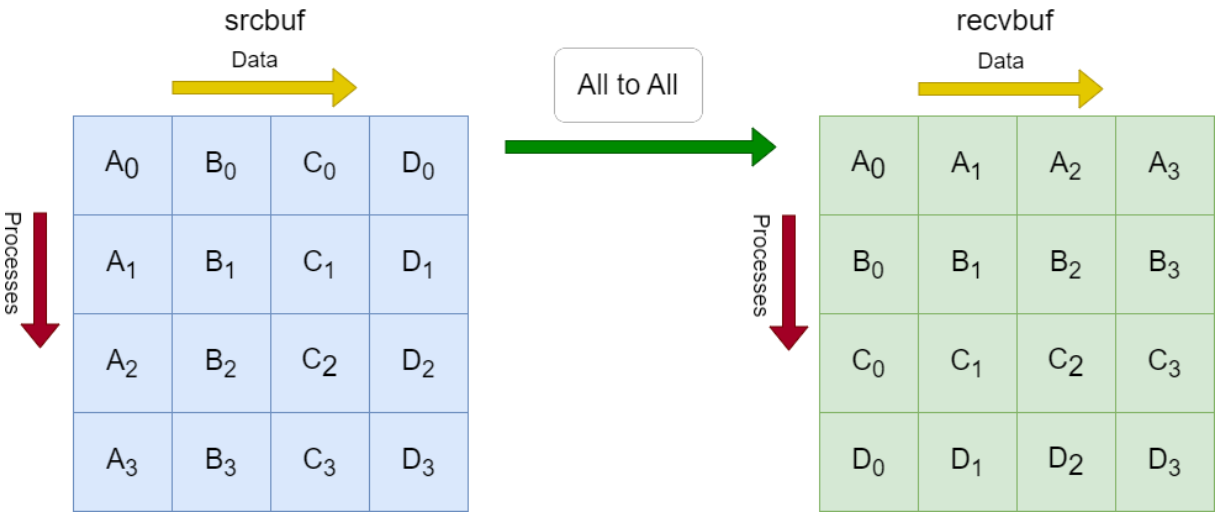
This reference application shows how the message passing interface (MPI) all-to-all collective can be accelerated on the Data Path Accelerator (DPA). In an MPI collective, all processes in the same job call the collective routine.

Given a communicator of *n* ranks, the application performs a collective operation in which all processes send and receive the same amount of data from all processes (hence all-to-all).

This document describes how to run the all-to-all example using the DOCA DPA API.

6.6.2 System Design

All-to-all is an MPI method. MPI is a standardized and portable message passing standard designed to function on parallel computing architectures. An MPI program is one where several processes run in parallel.

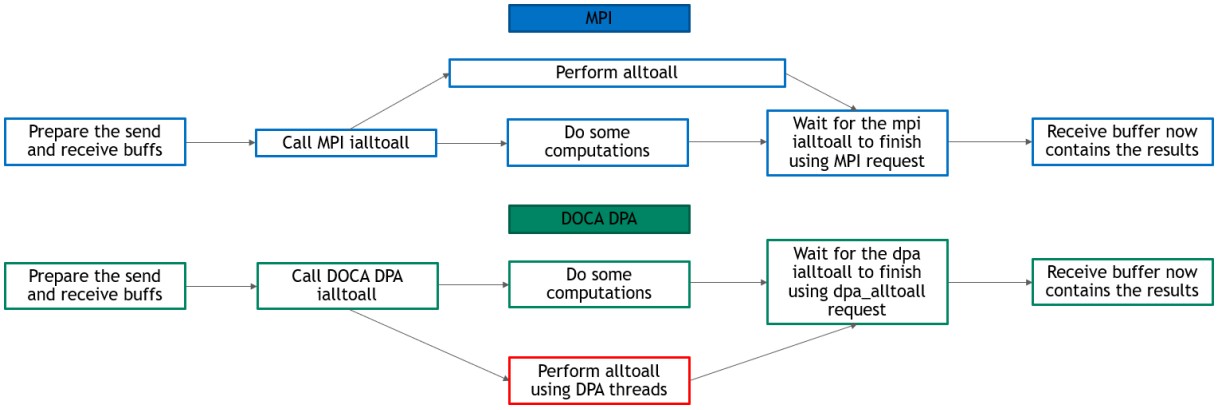


Each process in the diagram divides its local sendbuf into *n* blocks (4 in this example), each containing sendcount elements (4 in this example). Process *i* sends the *k*-th block of its local sendbuf to process *k* which places the data in the *i*-th block of its local recvbuf.

Implementing all-to-all method using DOCA DPA offloads the copying of the elements from the srcbuf to the recvbufs to the DPA, and leaves the CPU free to perform other computations.

6.6.3 Application Architecture

The following diagram describes the differences between host-based all-to-all and DPA all-to-all.



- In DPA all-to-all, DPA threads perform all-to-all and the CPU is free to do other computations
- In host-based all-to-all, CPU must still perform all-to-all at some point and is not completely free for other computations

6.6.4 DOCA Libraries

This application leverages the following DOCA library:

- DOCA DPA

Refer to its programming guide for more information.

6.6.5 Dependencies

- NVIDIA BlueField-3 platform is required
- The application can be run on target BlueField or on host.
- Open MPI version 4.1.5rc2 or greater (included in DOCA's installation).

6.6.6 Compiling the Application

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for the application is located in the following directory:

```
/opt/mellanox/doca/applications/dpa_all_to_all/
```

6.6.6.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

MPI is used for the compilation of this application. Make sure that MPI is installed on your setup (**openmpi** is provided as part of the installation of DOCA, as part of the **doca-all** and **doca-ofed** meta-packages).

Compiling the application requires updating the **LD_LIBRARY_PATH** and **PATH** environment variable to include MPI. For example, if **openmpi** is installed under **/usr/mpi/gcc/openmpi-4.1.7rc1**, then updating the environment variables should be like the following

```
export PATH=/usr/mpi/gcc/openmpi-4.1.7rc1/bin:${PATH}
export LD_LIBRARY_PATH=/usr/mpi/gcc/openmpi-4.1.7rc1/lib:${LD_LIBRARY_PATH}
```

To build all applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_dpa_all_to_all is created under **/tmp/build/dpa_all_to_all/**.

6.6.6.2 Compiling DPA All-to-all Application Only

To directly build only all-to-all application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_dpa_all_to_all=true
ninja -C /tmp/build
```

doca_dpa_all_to_all is created under **/tmp/build/dpa_all_to_all/**.

Alternatively, one can set the desired flags in **meson_options.txt** file instead of providing them in the compilation command line:

1. Edit the following flags in **/opt/mellanox/doca/applications/meson_options.txt**:
   - Set **enable_all_applications** to **false**
   - Set **enable_dpa_all_to_all** to **true**
2. Run the following compilation commands:

   ```
   cd /opt/mellanox/doca/applications/
   meson /tmp/build
   ninja -C /tmp/build
   ```

doca_dpa_all_to_all is created under **/tmp/build/dpa_all_to_all/**.

6.6.7 Running the Application

6.6.7.1 Prerequisites

MPI is used to run this application. Make sure that MPI is installed on your setup (**openmpi** is provided as part of the installation of DOCA on the host, as part of the **doca-all** and **doca-ofed** meta-packages).

Running the application requires updating the **LD_LIBRARY_PATH** and **PATH** environment variable to include MPI. For example, if **openmpi** is installed under **/usr/mpi/gcc/openmpi-4.1.7rc1**, then updating the environment variables should be like the following:

```
export PATH=/usr/mpi/gcc/openmpi-4.1.7rc1/bin:${PATH}
export LD_LIBRARY_PATH=/usr/mpi/gcc/openmpi-4.1.7rc1/lib:${LD_LIBRARY_PATH}
```

6.6.7.2 Application Execution

DPA all-to-all application is provided in source form. Therefore, a compilation is required before application can be executed.

1. Application usage instructions:

```
Usage: doca_dpa_all_to_all [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -m, --msgsize <Message size>      The message size - the size of the sendbuf and recvbuf (in bytes). Must
be in multiplies of integer size. Default is size of one integer times the number of processes.
  -pf_devs, --pf-devices <PF device name> PF devices names that supports DPA, separated by comma without
spaces (max of two devices). If not provided then a random device will be chosen.
```

This usage printout can be printed to the command line using the **-h** (or **--help**) option:

```
./doca_dpa_all_to_all -h
```

The **rdma_devs** flag is only available when running the application from the target BlueField.

For additional information, please refer to section "Command Line Flags".

2. CLI example for running the application on host:

This is an MPI program, so use **mpirun** to run the application (with the **-np** flag to specify the number of processes to run).

- The following runs the DPA all-to-all application with 8 processes using the default message size (the number of processes, which is 8, times the size of 1 integer) with a random InfiniBand device:

```
mpirun -np 8 ./doca_dpa_all_to_all
```

- The following runs DPA all-to-all application with 8 processes, with 128 bytes as message size, and with **mlx5_0** and **mlx5_1** as the InfiniBand devices:

```
mpirun -np 8 ./doca_dpa_all_to_all -m 128 -pf_devs "mlx5_0,mlx5_1"
```

The application supports running with a maximum of 16 processes. If you try to run with more processes, an error is printed and the application exits.

6.6.7.3 Command Line Flags

6.6.7.3.1 General Flags

Short Flag	Long Flag	Description
-h	**--help**	Prints a help synopsis and exits
-v	**--version**	Prints program version information and exits
-l	**--log-level**	Sets the numeric log level for the application: • **10** – DISABLE • **20** – CRITICAL • **30** – ERROR • **40** – WARNING • **50** – INFO • **60** – DEBUG • **70** – TRACE (requires compilation with TRACE support)
N/A	**--sdk-log-level**	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	**--log-filter**	Filters logs from specific modules (comma-separated list)
-j	**--json**	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.6.7.3.2 Program Flags

Flag Type	Short Flag	Long Flag	Description
Program flags	`m`	`msgsize`	The message size. The size of the sendbuf and recvbuf (in bytes). Must be in multiples of an integer. The default is size of 1 integer times the number of processes.
	`pf_devs`	`pf-devices`	InfiniBand devices names that support DPA, separated by comma without spaces (max of two devices). If `NOT_SET` then a random InfiniBand device is chosen.

6.6.7.4 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.6.8 Application Code Flow

1. Initialize MPI.

```
MPI_Init(&argc, &argv);
```

2. Parse application arguments.
   a. Initialize arg parser resources and register DOCA general parameters.

```
doca_argp_init();
```

   b. Register the application's parameters.

```
register_all_to_all_params();
```

   c. Parse the arguments.

```
doca_argp_start();
```

      i. The `msgsize` parameter is the size of the sendbuf and recvbuf (in bytes). It must be in multiples of an integer and at least the number of processes times an integer size.
      ii. The `pf_devices_param` parameter is the names of the InfiniBand devices to use (must support DPA). It can include up to two devices names.
   d. Only let the first process (of rank 0) parse the parameters to then broadcast them to the rest of the processes.
3. Check and prepare the needed resources for the `all_to_all` call:

a. Check the number of processes (maximum is 16).
b. Check the **msgsize**. It must be in multiples of integer size and at least the number of processes times integer size.
c. Allocate the sendbuf and recvbuf according to **msgsize**.

4. Prepare the resources required to perform all-to-all method using DOCA DPA:

> The application uses MPI without an additional security layer. The data exported (for sync event, RDMA, and MMAP), as described in this step, should be passed over a secure channel in a production deployment.

a. Initialize DOCA DPA context(s):
   i. Open DOCA DPA device(s) (DOCA device that supports DPA).

```
open_dpa_devices();
```

   ii. Initialize DOCA DPA context using the opened device.

```
extern struct doca_dpa_app *dpa_all2all_app;

doca_dpa_create(pf_doca_device, &pf_doca_dpa);

doca_dpa_set_app(pf_doca_dpa, dpa_all2all_app);

doca_dpa_start(pf_doca_dpa);
```

      1. When running from target BlueField: Initialize Extended DOCA DPA context using the opened RDMA device.

```
#ifdef DOCA_ARCH_DPU
    doca_dpa_device_extend(pf_doca_dpa, rdma_doca_device, &rdma_doca_dpa);
#else
    *rdma_doca_device = *pf_doca_device;
#endif
```

b. Initialize the required DOCA Sync Events for the all-to-all:
   i. One completion event for the kernel launch where the subscriber is CPU and the publisher is DPA.
   ii. Kernel events, published by remote peer and subscribed to by DPA, as the number of processes.

```
create_dpa_a2a_events() {
    // initialize completion event
    doca_sync_event_create(&comp_event);

    doca_sync_event_add_publisher_location_dpa(comp_event);

    doca_sync_event_add_subscriber_location_cpu(comp_event);

    doca_sync_event_start(comp_event);

    // initialize kernels events
    for (i = 0; i < resources->num_ranks; i++) {
        doca_sync_event_create(&(kernel_events[i]));

        doca_sync_event_add_publisher_location_remote_net(kernel_events[i]);

        doca_sync_event_add_subscriber_location_dpa(kernel_events[i]);

        doca_sync_event_start(kernel_events[i]);
    }
}
```

c. Prepare DOCA RDMAs and set them to work on DPA:
   i. Create DOCA RDMAs as the number of processes/ranks.

```
for (i = 0; i < resources->num_ranks; i++) {
    doca_rdma_create(&rdma);

    rdma_as_doca_ctx = doca_rdma_as_ctx(rdma);

    doca_rdma_set_permissions(rdma);

    doca_rdma_set_grh_enabled(rdma);

    doca_ctx_set_datapath_on_dpa(rdma_as_doca_ctx, rdma_doca_dpa);

    doca_ctx_start(rdma_as_doca_ctx);
}
```

ii. Connect local DOCA RDMAs to the remote DOCA RDMAs.

```
connect_dpa_a2a_rdmas();
```

iii. Get DPA handles for local DOCA RDMAs (so they can be used by DPA kernel) and copy them to DPA heap memory.

```
for (int i = 0; i < resources->num_ranks; i++) {
    doca_rdma_get_dpa_handle(rdmas[i], &(rdma_handles[i]));
}

doca_dpa_mem_alloc(&dev_ptr_rdma_handles);

doca_dpa_h2d_memcpy(dev_ptr_rdma_handles, rdma_handles);
```

d. Prepare the memory required to perform all-to-all method using DOCA Mmap. This includes creating DPA memory handles for sendbuf and recvbuf, getting other processes recvbufs handles, and copying these memory handles and their remote keys and events handlers to DPA heap memory.

```
prepare_dpa_a2a_memory();
```

5. Launch **alltoall_kernel** using DOCA DPA kernel launch with all required parameters:
   a. Every MPI rank launches a kernel of up to **MAX_NUM_THREADS**. This example defines **MAX_NUM_THREADS** as 16.
   b. Launch **alltoall_kernel** using **kernel_launch**.

```
doca_dpa_kernel_launch_update_set();
```

   c. Each process should perform **num_ranks** RDMA write operations, with local and remote buffers calculated based on the rank of the process that is performing the RDMA write operation and the rank of the remote process that is being written to. The application iterates over the rank of the remote process.i
   Each process runs **num_threads** threads on this kernel, therefore the number of RDMA write operations (which is the number of processes) is divided by the number of threads.
   Each thread should wait on its local events to make sure that the remote processes have finished RDMA write operations.

```
for (i = thread_rank; i < num_ranks; i += num_threads) {
    doca_dpa_dev_rdma_post_write();
    doca_dpa_dev_rdma_signal_set();
}

for (i = thread_rank; i < num_ranks; i += num_threads) {
    doca_dpa_dev_sync_event_wait_gt();
}
```

> RDMA operations should be executed over a secure channel in a production deployment, given the sensitivity arising from the nature of the protocol.

d. Wait until **alltoall_kernel** has finished.

```
doca_sync_event_wait_gt();
```

> Add an MPI barrier after waiting for the event to make sure that all of the processes have finished executing **alltoall_kernel**.
>
> ```
> MPI_Barrier();
> ```

After **alltoall_kernel** is finished, the recvbuf of all processes contains the expected output of all-to-all method.

6. Destroy **a2a_resources**:

a. Free all DOCA DPA memories.

```
doca_dpa_mem_free();
```

b. Destroy all DOCA Mmaps

```
doca_mmap_destroy();
```

c. Destroy all DOCA RDMAs.

```
doca_ctx_stop();
doca_rdma_destroy();
```

d. Destroy all DOCA Sync Events.

```
doca_sync_event_destroy();
```

e. Destroy DOCA DPA context.

```
doca_dpa_destroy();
```

f. Close DOCA device.

```
doca_dev_close();
```

6.6.9 References

- **/opt/mellanox/doca/applications/dpa_all_to_all/**

6.7 DOCA East-west Overlay Encryption Application

This guide describes IPsec-based strongSwan solution on top of NVIDIA® BlueField® DPU.

Important note for NVIDIA® BlueField®-2 DPUs

If your target application utilizes 100Gb/s or higher bandwidth, where a substantial part of the bandwidth is allocated for IPsec traffic, please refer to the *NVIDIA BlueField-2 DPUs Product Release Notes* to learn about a potential bandwidth limitation. To access the relevant product release notes, please contact your NVIDIA sales representative.

6.7.1 Introduction

IPsec is used to set up encrypted connections between different devices. It helps keep data sent over public networks secure. IPsec is often used to set up VPNs, and it works by encrypting IP packets as well as authenticating the packets' originator.

IPsec contains the following main modules:

- Key exchange – a key is a string of random bytes that can be used for encryption and decryption of messages. IPsec sets up keys with a key exchange between the connected devices, so that each device can decrypt the other device's messages.
- Authentication – IPsec provides authentication for each packet which ensures that they come from a trusted source.
- Encryption – IPsec encrypts the payloads within each packet and possibly, based on the transport mode, the packet's IP header.
- Decryption – at the other end of the communication, packets are decrypted by the IPsec supported node.

IPsec supports two types of headers:

- Authentication header (AH) – AH protocol ensures that packets are from a trusted source. AH does not provide any encryption.
- Encapsulating security protocol (ESP) – ESP encrypts the payload for each packet as well as the IP header depending on the transport mode. ESP adds its own header and a trailer to each data packet.

IPsec support two types of transport mode:

- IPsec tunnel mode – used between two network nodes, each acting as tunnel initiator/terminator on a public network. In this mode, the original IP header and payload are both encrypted. Since the IP header is encrypted, an IP tunnel is added for network forwarding. At each end of the tunnel, the routers decrypt the IP headers to route the packets to their destinations.
- Transport mode – the payload of each packet is encrypted, but the original IP header is not. Intermediary network nodes are therefore able to view the destination of each packet and route the packet, unless a separate tunneling protocol is used.

strongSwan is an open-source IPsec-based VPN solution. For more information, refer to strongSwan documentation.

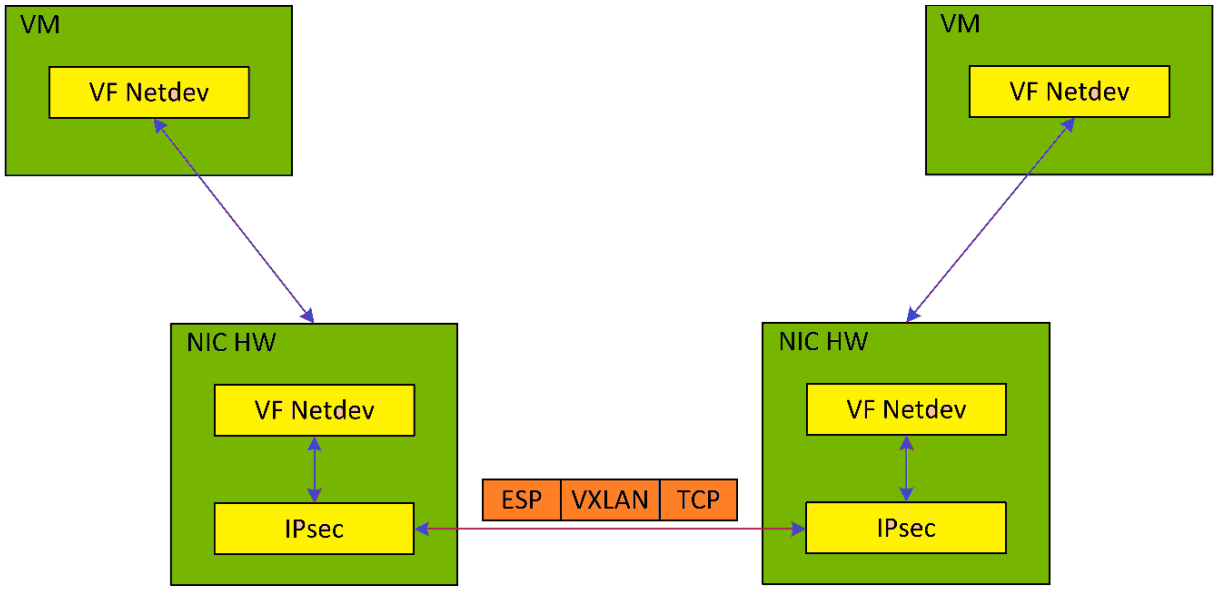
6.7.2 System Design

IPsec packet offload offloads both IPsec crypto (encrypt/decrypt) and IPsec encapsulation to the hardware.

The deployment model allows the IPsec offload to be transparent to the host with the benefits of securing legacy workloads (no dependency on host SW stack) and to zero CPU utilization on host.

IPsec packet offload configuration works with and is transparent to OVS offload. This means all packets from OVS offload are encrypted by IPsec rules.

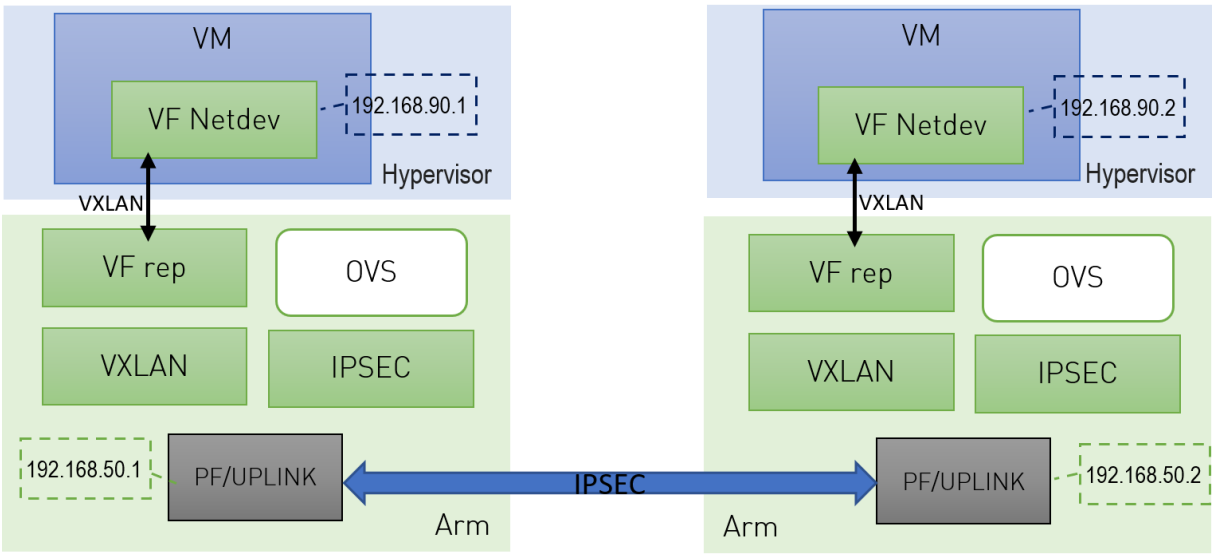
The following figure illustrates the interaction between IPsec packet offload and OVS VXLAN offload.



IPsec packet offload is only supported on Ubuntu Bluefield kernel 5.15

OVS offload and IPsec IPv6 do not work together.

6.7.3 Application Architecture



1. Configure strongSwan IPsec offload using **swanctl.conf** configuration file.
2. Traffic is sent from the host through BlueField.
3. Using OVS, the packets are encapsulated on ingress using tunnel protocols (VXLAN for example) to match IPsec configuration by strongSwan.
4. Set by strongSwan configuration file, traffic will be encrypted using the hardware offload.
5. Egress flow is decryption first, decapsulation of the tunnel header and forward to the relevant physical function.

6.7.4 DOCA Libraries

N/A

6.7.5 Dependencies

- NVIDIA® BlueField®-2 platform or higher is required
- BlueField Ubuntu BFB with kernel 5.15
- strongSwan version 5.9.10 or greater

6.7.5.1 Compiling and Installing strongSwan

The following section provides instruction for compiling and installing NVIDIA's strongSwan fork (branch BF-5.9.10), but users can use the upstream strongSwan instructions (as long as the version is 5.9.10 or greater).

To compile and install strongswan from the source, perform these steps on BlueField (Arm cores):

1. Install dependencies:

```
sudo apt install dpkg-dev devscripts debhelper dh-autoreconf autoconf flex libtool gperf lzip yacc bison
flex m4 libgmp-dev libsystemd-dev -y
```

2. Clone and compile strongSwan:

```
$ git clone https://github.com/Mellanox/strongswan.git
$ cd strongswan
$ git checkout BF-5.9.10
$ dpkg-buildpackage -uc -us -b
```

This should create two **.deb** files in the parent folder, **strongswan_5.9.10-2.bf_arm64.deb** and **strongswan-swanctl_5.9.10-2.bf_arm64.deb**.

> strongSwan BF-5.9.10 is based on the official strongSwan 5.9.10 branch with added packaging and support for DOCA IPsec plugin. Refer to the DOCA IPsec Security Gateway Application Guide for more information regarding the strongSwan DOCA plugin.

3. Uninstall all old strongSwan and charon packages to prevent conflicts:

```
$ sudo dpkg -P --force-depends <packages>
```

Example:

```
$ sudo dpkg -P --force-depends libstrongswan libstrongswan-standard-plugins strongswan strongswan-charon
strongswan-libcharon strongswan-starter libcharon-extauth-plugins
```

> Users can find all strongSwan and charon packages by running **dpkg -l | grep strongswan** and **dpkg -l | grep charon**.

4. Install the two strongSwan .deb files (they should've been created in the parent folder).

```
$ sudo dpkg -i ../strongswan_5.9.10-2.bf_arm64.deb ../strongswan-swanctl_5.9.10-2.bf_arm64.deb
```

6.7.6 Configuration Flow

The following section provides information on manually configuring IPsec packet offload in general and on using OVS IPsec with strongSwan specifically.

> There is a script, **east_west_overlay_encryption.sh** which performs the steps in this section automatically.

If you are working directly with the **ip xfrm** tool, use **/opt/mellanox/iproute2/sbin/ip** to benefit from IPsec packet offload support.

There are two parts in the configuration flow

1. Enabling IPsec packet offload mode.

2. Configuring the IPsec OVS bridge using one of three modes of authentication.

> ⚠ An alternative for step two is configuring **swanctl.conf** files (configuration files for strongSwan) manually and using strongSwan directly instead of using IPsec OVS (which automatically generates **swanctl.conf** files) as explained in section "Configuring OVS IPsec Using strongSwan Manually".

6.7.6.1 Enabling IPsec Packet Offload

This section explicitly enables IPsec packet offload on the Arm cores before setting up offload-aware IPsec tunnels.

> ⚠ If an OVS VXLAN tunnel configuration already exists, stop **openvswitch** service prior to performing the steps below and restart the service afterwards.

Explicitly enable IPsec full offload on the Arm cores.

1. Set **IPSEC_FULL_OFFLOAD="yes"** in **/etc/mellanox/mlnx-bf.conf**.

   > ⚠ If **IPSEC_FULL_OFFLOAD** does not appear in **/etc/mellanox/mlnx-bf.conf** then you are probably using an old version of the BlueField image. Check the way of enabling IPsec full offload in a previous DOCA versions in the NVIDIA DOCA Documentation Archives.

2. Restart IB driver (rebooting also works). Run:

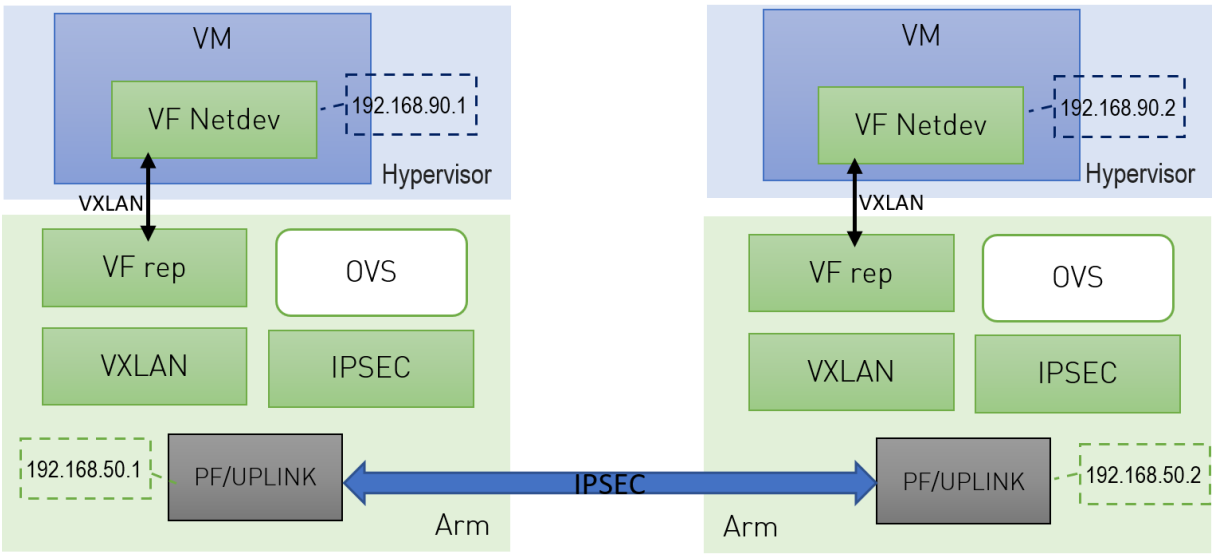
   ```
   /etc/init.d/openibd restart
   ```

> ⚠ To revert IPsec full offload mode, redo the procedure from step 1, only difference is to set **IPSEC_FULL_OFFLOAD="no"** in **/etc/mellanox/mlnx-bf.conf**.

6.7.6.2 Configuring OVS IPsec

> ⚠ Before proceeding with this section, make sure to follow the procedure in section "Enabling IPsec Packet Offload" for both DPUs.

This section configures OVS IPsec VXLAN tunnel which automatically generates the **swanctl.conf** files and runs strongSwan (the IPsec daemon). The following figure illustrates an example with two BlueField DPUs, Left and Right, operating with a secured VXLAN channel.



Two BlueField DPUs are required to build an OVS IPsec tunnel between the two hosts, Right and Left.

The OVS IPsec tunnel configures an unaware IPsec connection between the two hosts' InfiniBand devices. For the sake of this example, the host's InfiniBand network device is **HOST_PF**, and the DPU's host representor is **PF_REP** and the DPU's physical function **PF**.

This example sets up the following variables on both Arms:

```
# host_ip1=1.1.1.1
# host_ip2=1.1.1.2
# HOST_PF=ens7np0
# ip1=192.168.50.1
# ip2=192.168.50.2
# PF=p0
# PF_REP=pf0hpf
```

The name of the **HOST_PF** could be different in your machine. You may verify this by running:

```
host# ibdev2netdev
mlx5_0 port 1 ==> ens7np0 (Down)
mlx5_1 port 1 ==> ens8np1 (Down)
```

This example uses the first InfiniBand's (**mlx5_0**) network device which is **ens7np0**.

1. Configure IP addresses for the **HOST_PF**s of both hosts (x86):
   a. On **host_1**:

   ```
   # ifconfig $HOST_PF $host_ip1/24 up
   ```

   b. On **host_2**:

   ```
   # ifconfig $HOST_PF $host_ip2/24 up
   ```

> ⚠ Step 1 is the only command that is performed on the host, the rest of the commands are performed on the Arm (DPU) side.

2. Configure IP addresses for the PFs of both Arms:
   a. On **Arm_1**:

```
# ifconfig $PF $ip1/24 up
```

   b. On **Arm_2**:

```
# ifconfig $PF $ip2/24 up
```

3. Start Open vSwitch. If your operating system is Ubuntu, run the following on both **Arm_1** and **Arm_2**:

```
# service openvswitch-switch start
```

   If your operating system is CentOS, run the following on both **Arm_1** and **Arm_2**:

```
# service openvswitch restart
```

4. Start OVS IPsec service. Run on both **Arm_1** and **Arm_2**:

```
# systemctl start openvswitch-ipsec.service
```

5. Set up OVS bridges in both DPUs. Run on both **Arm_1** and **Arm_2**:

```
# ovs-vsctl add-br vxlan-br
# ovs-vsctl add-port ovs-br $PF_REP
# ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
```

   > ⚠ Configuring **other_config:hw-offload=true** sets IPsec Packet offload. Setting it to **false** sets software IPsec.

   > ⚠ The MTU of the MTU of the tunnel interface (PF) should be at least 50 bytes larger than the MTU of the endpoints of the tunnels above (PF_REP) to account for the size of the VXLAN tunnel header. For example, if the MTU of PF_REP is 1500 then the MTU of PF should be at least 1550.
   >
   > To configure the MTU of the PF:
   >
   > ```
   > # ifconfig $PF mtu $PF_MTU up
   > ```

6. Set up IPsec tunnel on the OVS bridge. Three authentication methods are possible, choose your preferred authentication method and follow the steps relevant to it. Note that the last two authentication methods requires you to create certificates (self-signed certificates or certificate authority certificates).

⚠ After the IPsec tunnel is set up using one of the three methods of authentication, strongSwan configuration is done automatically and the **swanctl.conf** files will be generated and strongSwan will run automatically.

6.7.6.2.1 Authentication Methods

The following subsections detail the possible authentication methods for setting up the IPsec tunnel on the OVS bridge.

6.7.6.2.1.1 Pre-shared Key

This method configures OVS IPsec using a pre-shared key. You must select a pre-shared key, for example:

```
psk=swordfish
```

1. Set up the VXLAN tunnel:
    a. On **Arm_1**, run:

```
# ovs-vsctl add-port vxlan-br tun -- \
            set interface tun type=vxlan \
                        options:local_ip=$ip1 \
                        options:remote_ip=$ip2 \
                        options:key=100 \
                        options:dst_port=4789 \
                        options:psk=$psk
```

    b. On **Arm_2**, run:

```
# ovs-vsctl add-port vxlan-br tun -- \
            set interface tun type=vxlan \
                        options:local_ip=$ip2 \
                        options:remote_ip=$ip1 \
                        options:key=100 \
                        options:dst_port=4789\
                        options:psk=$psk
```

6.7.6.2.1.2 Self-signed Certificate

This method configures OVS IPsec using self-signed certificates. You must generate self-signed certificates and keys. This example demonstrates how to generate self-signed certificates using **ovs-pki** but you may generate them in any other way while skipping step 1.

1. Generate self-signed certificates using **ovs-pki**:
    a. On **Arm_1**, run:

```
# ovs-pki req -u host_1
# ovs-pki self-sign host_1
```

    After running this code you should have **host_1-cert.pem** and **host_1-privkey.pem**.
    b. On **Arm_2**, run:

```
# ovs-pki req -u host_2
# ovs-pki self-sign host_2
```

    After running this code you should have **host_2-cert.pem** and **host_2-privkey.pem**.

2. Configure the certificates and private keys:
    a. Copy the certificate of **Arm_1** to **Arm_2**, and the certificate of **Arm_2** to **Arm_1**.
    b. On each machine, move both **host_1-privkey.pem** and **host_2-cert.pem** to **/etc/swanctl/x509/** if on Ubuntu, or **/etc/strongswan/swanctl/x509/** if on CentOS.
    c. On each machine, move the local private key (**host_1-privkey.pem** on **Arm_1** and **host_2-privkey.pem** on **Arm_2**) to **/etc/swanctl/private** if on Ubuntu, or **/etc/strongswan/swanctl/private** if on CentOS.
3. Set up OVS **other_config** on both sides.
    a. On **A rm_1**:

```
# ovs-vsctl set Open_vSwitch . other_config:certificate=/etc/swanctl/x509/host_1-cert.pem \
  other_config:private_key=/etc/swanctl/private/host_1-privkey.pem
```

    b. On **Arm_2**:

```
# ovs-vsctl set Open_vSwitch . other_config:certificate=/etc/swanctl/x509/host_2-cert.pem \
  other_config:private_key=/etc/swanctl/private/host_2-privkey.pem
```

4. Set up the VXLAN tunnel:
    a. On **Arm_1**:

```
# ovs-vsctl add-port vxlan-br vxlanp0 -- set interface vxlanp0 type=vxlan options:local_ip=$ip1 \
  options:remote_ip=$ip2 options:key=100 options:dst_port=4789 \
  options:remote_cert=/etc/swanctl/x509/host_2-cert.pem
# service openvswitch-switch restart
```

    b. On **Arm_2**:

```
# ovs-vsctl add-port vxlan-br vxlanp0 -- set interface vxlanp0 type=vxlan options:local_ip=$ip2 \
  options:remote_ip=$ip1 options:key=100 options:dst_port=4789 \
  options:remote_cert=/etc/swanctl/x509/host_1-cert.pem
# service openvswitch-switch restart
```

In steps 3 and 4, if you are in CentOS you must change the path of the certificates to **/etc/strongswan/swanctl/x509/** and the path of the private keys to **/etc/strongswan/swanctl/private**.

6.7.6.2.1.3 CA-signed Certificate

This method configures OVS IPsec using certificate authority (CA)-signed certificates. You must generate CA-signed certificates and keys. The example demonstrates how to generate CA-signed certificates using **ovs-pki** but you may generate them in any other way while skipping step 1.

1. Generate CA-signed certificates using **ovs-pki**. For this method, all the certificates and the requests must be in the same directory during the certificate generating and signing. This example refers to this directory as **certsworkspace**.
    a. On **Arm_1**, run:

```
# ovs-pki init --force
# cp /var/lib/openvswitch/pki/controllerca/cacert.pem <path_to>/certsworkspace
# cd <path_to>/certsworkspace
# ovs-pki req -u host_1
```

```
# ovs-pki sign host1 switch
```

After running this code, you should have **host_1-cert.pem**, **host_1-privkey.pem**, and cacert.pm in the **certsworkspace** folder.

b. On **Arm_2**, run:

```
# ovs-pki init --force
# cp /var/lib/openvswitch/pki/controllerca/cacert.pem <path_to>/certsworkspace
# cd <path_to>/certsworkspace
# ovs-pki req -u host_2
# ovs-pki sign host_2 switch
```

After running this code, you should have **host_2-cert.pem**, **host_2-privkey.pem**, and **cacert.pm** in the **certsworkspace** folder.

2. Configure the certificates and private keys:
   a. Copy the certificate of **Arm_1** to **Arm_2** and the certificate of **Arm_2** to **Arm_1**.
   b. On each machine, move both **host_1-privkey.pem** and **host_2-cert.pem** to **/etc/swanctl/x509/** if on Ubuntu, or **/etc/strongswan/swanctl/x509/** if on CentOS.
   c. On each machine, move the local private key (**host_1-privkey.pem** if on **Arm_1** and **host_2-privkey.pem** if on **Arm_2**) to **/etc/swanctl/private** if on Ubuntu, or **/etc/strongswan/swanctl/private** if on CentOS.
   d. On each machine, copy **cacert.pem** to the **x509ca** directory under **/etc/swanctl/x509ca/** if on Ubuntu, or **/etc/strongswan/swanctl/x509ca/** if on CentOS.
3. Set up OVS **other_config** on both sides.
   a. On **Arm_1**:

```
# ovs-vsctl set Open_vSwitch . \
        other_config:certificate=/etc/strongswan/swanctl/x509/host_1.pem \
        other_config:private_key=/etc/strongswan/swanctl/private/host_1-privkey.pem \
        other_config:ca_cert=/etc/strongswan/swanctl/x509ca/cacert.pem
```

   b. On **Arm_2**:

```
# ovs-vsctl set Open_vSwitch . \
        other_config:certificate=/etc/strongswan/swanctl/x509/host_2.pem \
        other_config:private_key=/etc/strongswan/swanctl/private/host_2-privkey.pem \
        other_config:ca_cert=/etc/strongswan/swanctl/x509ca/cacert.pem
```

4. Set up the tunnel:
   a. On **Arm_1**:

```
# ovs-vsctl add-port vxlan-br vxlanp0 -- set interface vxlanp0 type=vxlan options:local_ip=$ip1 \
options:remote_ip=$ip2 options:key=100 options:dst_port=4789 \ options:remote_name=host_2
# service openvswitch-switch restart
```

   b. On **Arm_2**:

```
# ovs-vsctl add-port vxlan-br vxlanp0 -- set interface vxlanp0 type=vxlan options:local_ip=$ip2 \
options:remote_ip=$ip1 options:key=100 options:dst_port=4789 \ options:remote_name=host_1
# service openvswitch-switch restart
```

⚠ In steps 3 and 4, if you are in CenOS you must change the path of the certificates to **/etc/strongswan/swanctl/x509/**, the path of the CA certificate to **/etc/strongswan/swanctl/x509ca/**, and the path of the private keys to **/etc/strongswan/swanctl/private/**.

6.7.6.3 Ensuring IPsec is Configured

Using **/opt/mellanox/iproute2/sbin/ip xfrm state show**, you should be able to see 4 IPsec states for the IPsec connection you configured with the keyword **in mode packet** meaning which means that you are in IPsec packet HW offload mode.

For example, after configuring IPsec using pre-shared key method, you would get something similar to the following on **Arm_1**:

```
# /opt/mellanox/iproute2/sbin/ip xfrm state show

src 192.168.50.1 dst 192.168.50.2
    proto esp spi 0xcc8bf8ad reqid 1 mode transport
    replay-window 0 flag esn
    aead rfc4106(gcm(aes)) 0x9f45cc4577e70c4e077bcc0c1473a782143e7ad199f58566519639d03b593b8996383f11 128
    anti-replay esn context:
     seq-hi 0x0, seq 0x0, oseq-hi 0x0, oseq 0x0
     replay_window 1, bitmap-length 1
     00000000
    crypto offload parameters: dev p0 dir out mode packet
    sel src 192.168.50.1/32 dst 192.168.50.2/32 proto udp sport 4789
src 192.168.50.2 dst 192.168.50.1
    proto esp spi 0xce8bf4b6 reqid 1 mode transport
    replay-window 0 flag esn
    aead rfc4106(gcm(aes)) 0xf2d0e335d9a64ef6e385a630a32b0e43bb52f581290cd34bbb8f7592d54f11657ed0258e 128
    anti-replay esn context:
     seq-hi 0x0, seq 0x0, oseq-hi 0x0, oseq 0x0
     replay_window 32, bitmap-length 1
     00000000
    crypto offload parameters: dev p0 dir in mode packet
    sel src 192.168.50.2/32 dst 192.168.50.1/32 proto udp dport 4789
src 192.168.50.1 dst 192.168.50.2
    proto esp spi 0xcb600a84 reqid 2 mode transport
    replay-window 0 flag esn
    aead rfc4106(gcm(aes)) 0x7fb26035299bcc9b973abea5d581acfbcf87cbf0bd053b745c4d95c62311f934010973f6 128
    anti-replay esn context:
     seq-hi 0x0, seq 0x0, oseq-hi 0x0, oseq 0x0
     replay_window 1, bitmap-length 1
     00000000
    crypto offload parameters: dev p0 dir out mode packet
    sel src 192.168.50.1/32 dst 192.168.50.2/32 proto udp dport 4789
src 192.168.50.2 dst 192.168.50.1
    proto esp spi 0xc137d5a0 reqid 2 mode transport
    replay-window 0 flag esn
    aead rfc4106(gcm(aes)) 0x28e3d12ad4e24aa9d9de9459de8ef8bb4379e8e12faac0054c5b629b6aa50fdeda8e4574 128
    anti-replay esn context:
     seq-hi 0x0, seq 0x0, oseq-hi 0x0, oseq 0x0
     replay_window 32, bitmap-length 1
     00000000
    crypto offload parameters: dev p0 dir in mode packet
    sel src 192.168.50.2/32 dst 192.168.50.1/32 proto udp sport 4789
```

After insuring that the IPsec connection is configured, you can send encrypted traffic between **host_1** and **host_2** using the **HOST_PF** s IP addresses.

6.7.6.4 Configuring OVS IPsec Using strongSwan Manually

This section configures an OVS VXLAN tunnel which then uses **swanctl.conf** files and runs strongSwan (the IPsec daemon) manually.

> ⚠ Before proceeding with this section, make sure to follow the procedure in section "Enabling IPsec Packet Offload" for both DPUs.

1. Build a VXLAN tunnel over OVS and connect the PF representor to the same OVS bridge.
   a. On **Arm_1**:

```
# ovs-vsctl add-br vxlan-br
# ovs-vsctl add-port vxlan-br PF_REP
# ovs-vsctl add-port vxlan-br vxlan11 -- set interface vxlan11 type=vxlan options:local_ip=$ip1 \
        options:remote_ip=$ip2 options:key=100 options:dst_port=4789 \
# ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
```

b. On **Arm_2**:

```
# ovs-vsctl add-br vxlan-br
# ovs-vsctl add-port vxlan-br PF_REP
# ovs-vsctl add-port vxlan-br vxlan11 -- set interface vxlan11 type=vxlan options:local_ip=$ip2 \
        options:remote_ip=$ip1 options:key=100 options:dst_port=4789 \
# ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
```

2. If your operating system is Ubuntu, run on both **Arm_1** and **Arm_2**:

```
service openvswitch-switch start
```

If your operating system is CentOS, run:

```
service openvswitch restart
```

3. Enable TC offloading for the PF. Run on both **Arm_1** and **Arm_2**:

```
# ethtool -K $PF hw-tc-offload on
```

4. Disable host PF as the port owner from Arm. Run on both **Arm_1** and **Arm_2**:

```
# mlxprivhost -d /dev/mst/mt${pciconf} --disable_port_owner r
```

⚠ To get **${pciconf}**, run the following on the DPU:

```
# ls --color=never /dev/mst/ | grep --color=never '^m.*f0$' | cut -c 3-
```

For example:

```
# mlxprivhost -d /dev/mst/mt41686_pciconf0 --disable_port_owner r
```

5. Configure the **swanctl.conf** files for each machine. See section swanctl.conf Files.

⚠ Each machine should have exactly one **.swanctl.conf** file in **/etc/swanctl/conf.d/**.

6. Load the **swanctl.conf** files and initialize strongSwan. Run:
   a. On the **Arm_2**, run:

```
systemctl restart strongswan.service
swanctl --load-all
```

   b. On the **Arm_1**, run:

```
systemctl restart strongswan.service
swanctl --load-all
swanctl -i --child bf
```

Now the IPsec connection should be established.

6.7.6.5 swanctl.conf Files

strongSwan configures IPSec packet HW offload using a new value added to its configuration file **swanctl.conf**. The file should be placed under **sysconfdir** which by default can be found at **/etc/swanctl/swanctl.conf**.

The terms Left (BFL) and Right (BFR), in reference to the illustration under "Application Architecture", are used to identify the two nodes (or machines) that communicate.

> ⚠ Either side (BFL or BFR) can fulfill either role (initiator or receiver).

In this example, 192.168.50.1 is used for the left PF uplink and 192.168.50.2 for the right PF uplink.

```
connections {
   BFL-BFR {
      local_addrs  = 192.168.50.1
      remote_addrs = 192.168.50.2

      local {
        auth = psk
        id = host1
      }
      remote {
        auth = psk
        id = host2
      }
        children {
         bf-out {
            local_ts = 192.168.50.1/24 [udp]
            remote_ts = 192.168.50.2/24 [udp/4789]
            esp_proposals = aes128gcm128-x25519-esn
            mode = transport
            policies_fwd_out = yes
            hw_offload = packet
         }
         bf-in {
            local_ts = 192.168.50.1/24 [udp/4789]
            remote_ts = 192.168.50.2/24 [udp]
            esp_proposals = aes128gcm128-x25519-esn
            mode = transport
            policies_fwd_out = yes
            hw_offload = packet
         }
      }
      version = 2
      mobike = no
      reauth_time = 0
      proposals = aes128-sha256-x25519
   }
}

secrets {
   ike-BF {
      id-host1 = host1
      id-host2 = host2
      secret = 0sv+NkxY9LLZvwj4qCC2o/gGrWDF2d21jL
   }
}
```

The BFB installation will place two example **swanctl.conf** files for BFL and BFR (**BFL.swanctl.conf** and **BFR.swanctl.conf** respectively) in the strongSwan **conf.d** directory. Each node should have only one **swanctl.conf** file in its strongSwan **conf.d** directory.

Note that:

- "**hw_offload = packet**" is responsible for configuring IPsec packet offload
- Packet offload support has been added to the existing **hw_offload** field and preserves backward compatibility.
  For your reference:

Value	Description
no	Do not configure HW offload.
crypto	Configure crypto HW offload if supported by the kernel and hardware, fail if not supported.
yes	Same as crypto (considered legacy).
packet	Configure packet HW offload if supported by the kernel and hardware, fail if not supported.
auto	Configure packet HW offload if supported by the kernel and hardware, do not fail (perform fallback to crypto or no as necessary).

- Whenever the value of **hw_offload** is changed, strongSwan configuration must be reloaded.
- Switching to crypto HW offload requires setting up **devlink/ipsec_mode** to **none** beforehand.
- Switching to packet HW offload requires setting up
- **[udp/4789]** is crucial for instructing strongSwan to IPSec only VXLAN communication.
- Packet HW offload can only be done on what is streamed over VXLAN.

Mind the following limitations:

Fields	Limitation
reauth_time	Ignored if set
rekey_time	Do not use. Ignored if set.
rekey_bytes	Do not use. Not supported and will fail if it is set.
rekey_packets	Use for rekeying

6.7.7 Running the Application

6.7.7.1 Installation

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

6.7.7.2 Application Execution

Notes:

- IPsec daemons are started by **systemd strongswan.service**
- Use **systemctl [start | stop | restart]** to control IPsec daemons through **strongswan.service**. For example, to restart, run:

```
systemctl restart strongswan.service
```

This command effectively does the same thing as **ipsec restart**.

> ⚠ Do not use the **ipsec** script (located under **/usr/sbin/ipsec**) to restart/stop/start the IPsec connection.

This subsection explains how to configure and set an IPsec connection using the script. To configure the IPsec connection, you need two DPUs, referred to as the initiator and receiver machines. There are no differences between the two machines except that the initiator is the one that initiates the connection between the two (and should run the script after the receiver).

The script is located under **/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh**.

6.7.7.2.1 Script Parameters

Parameter	Description	Valid Values	Use when	Notes
side	The side of the connection (receiver or initiator).	• **r|receiver** • **i|intitiator**	Always	This parameter must be always passed on the command line and cannot be passed in the JSON parameter file.
j|json	The JSON parameters file full path.	JSON file path, written according to the template demonstrated in the following file: **/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption_params.json**.	To pass the parameters as a JSON file	When running the script with JSON file, you cannot pass on the command line other parameters than the **side** and the JSON file.
initiator_ip_addr	The IP address of the initiator machine's port interface for the IPsec connection.	A valid IP address, ranging from 1.1.1.1 to 255.255.255.255.	Always	In the JSON file, it is set by default to 192.168.50.1.
receiver_ip_addr	The IP address of the receiver machine's port interface for the IPsec connection.	A valid IP address, ranging from 1.1.1.1 to 255.255.255.255.	Always	In the JSON file, it is set by default to 192.168.50.2.
port_num	The number of the port interface (p0/p1) for the IPsec connection.	0 or 1.	Always	In the JSON file, it is set by default to 0.

Parameter	Description	Valid Values	Use when	Notes
`auth_method`	the authentication method of IPsec. can be `psk` (pre-shared key), `ssc` (self-signed certificate) or `ca` (CA-signed certificate). Set by default to `psk`.	Can be `psk` (pre-shared key), `ssc` (self-signed certificate) or `ca` (CA-signed certificate).	Always	In the JSON file, it is set by default to `psk.`
`preshared_key`	The pre-shared key.	A sequence of characters (string).	The `auth_method` parameter is `psk`	In the JSON file it is set by default to `swordfish`. Both the initiator and receiver must configure the same `preshared_key`.
`initiator_cert_path`	The initiator's certificate.	Any valid self-signed or CA signed certificate. Must provide full path of certificate.	The `auth_method` parameter is `ssc` or `ca`	Both the initiator and receiver must configure the same `initiator_cert_path`.
`receiver_cert_path`	The receiver's certificate.	Any valid self-signed or CA signed certificate. Must provide full path of certificate.	The `auth_method` parameter is `ssc` or `ca`	Both the initiator and receiver must configure the same `receiver _cert_path`.
`initiator_key_path`	the initiator's private-key.	Any valid private key that is generated with the certificate. Must provide full path of private key.	The `side` parameter is set to `initiator` and the `auth_method` is set to `ssc` or `ca`	N/A
`receiver_key_path`	the receiver's private-key.	Any valid private key that is generated with the certificate. Must provide full path of private key.	The `side` parameter is set to `receiver` and the `auth_method` is set to `ssc` or `ca`	N/A

Parameter	Description	Valid Values	Use when	Notes
`initiator_cacert_path`	The initiator's CA certificate.	Any valid CA certificate. Must provide full path of certificate.	The `side` and `auth_method` parameters are set to `initiator` and `ca` respectively	N/A
`receiver_cacert_path`	The receiver's CA certificate.	Any valid CA certificate. Must provide full path of certificate.	The `side` and `auth_method` parameters are set to `receiver` and `ca` respectively	N/A
`initiator_cn`	The common name (CN) of the initiator's certificate.	Must be the same as the CN described in the initiator's certificate.	The `side` and `auth_method` parameters are set to `receiver` and `ca` respectively	N/A
`receiver_cn`	The CN of the receiver's certificate.	Must be the same as the CN described in the receiver's certificate.	The `side` and `auth_method` parameters are set to `initiator` and `ca` respectively	N/A

There are two ways of passing the parameters, either using the JSON parameters file or by passing the parameters on the command line.

6.7.7.2.2 Using JSON Parameters File

In this method, you must configure the parameters file and the then run the script:

1. Configure the JSON parameters file located under `/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption_params.json` or create a JSON file

according to the template of **east_west_overlay_encryption_params.json** for the script according to the explanation under section "Script Parameters".

2. Run the script on the receiver's DPU with the JSON file:

```
/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh --side=r --
json=/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption_params.json
```

3. Run the script on the initiator's DPU:

```
/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh --side=i --
json=/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption_params.json
```

You may now send encrypted data over the PF interface (192.168.50.[1|2]) configured for VXLAN.

6.7.7.2.3 Passing Parameters on Command Line

In this method, you do not need to configure the parameters file and can run the script with the appropriate parameters.

6.7.7.2.3.1 Passing Parameters for Pre-shared Key Authentication Method

1. Run the script on the receiver's DPU:

```
/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh --side=r --
initiator_ip_addr=INITIATOR_IP_ADDRESS --receiver_ip_addr=RECEIVER_IP_ADDRESS --port_num=PORT_NUM \
--auth_method=psk --preshared_key=PRESHARED_KEY
```

2. Run the script on the initiator's DPU:

```
/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh --side=i --
initiator_ip_addr=INITIATOR_IP_ADDRESS --receiver_ip_addr=RECEIVER_IP_ADDRESS --port_num=PORT_NUM \
--auth_method=psk --preshared_key=PRESHARED_KEY
```

6.7.7.2.3.2 Passing Parameters for Self-signed Certificates Authentication Method

1. Run the script on the receiver's DPU:

```
/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh --side=r --
initiator_ip_addr=INITIATOR_IP_ADDRESS --receiver_ip_addr=RECEIVER_IP_ADDRESS --port_num=PORT_NUM \
--auth_method=ssc --initiator_cert_path=INITIATOR_CERT_PATH --receiver_cert_path=RECEIVER_CERT_PATH --
receiver_key_path=RECEIVER_KEY_PATH
```

2. Run the script on the initiator's DPU:

```
/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh --side=i --
initiator_ip_addr=INITIATOR_IP_ADDRESS --receiver_ip_addr=RECEIVER_IP_ADDRESS --port_num=PORT_NUM \
--auth_method=ssc --initiator_cert_path=INITIATOR_CERT_PATH --receiver_cert_path=RECEIVER_CERT_PATH --
initiator_key_path=INITIATOR_KEY_PATH
```

6.7.7.2.3.3 Passing Parameters for CA Certificates Authentication Method

1. Run the script on the receiver's DPU:

```
/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh --side=r --
initiator_ip_addr=INITIATOR_IP_ADDRESS --receiver_ip_addr=RECEIVER_IP_ADDRESS --port_num=PORT_NUM \
--auth_method=ca --initiator_cert_path=INITIATOR_CERT_PATH --receiver_cert_path=RECEIVER_CERT_PATH --
receiver_key_path=RECEIVER_KEY_PATH --receiver_cacert_path=RECEIVER_CACERT_PATH --initiator_cn=INITIATOR_CN
```

2. Run the script on the initiator's DPU:

```
/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh --side=i --
initiator_ip_addr=INITIATOR_IP_ADDRESS --receiver_ip_addr=RECEIVER_IP_ADDRESS --port_num=PORT_NUM \
--auth_method=ssc --initiator_cert_path=INITIATOR_CERT_PATH --receiver_cert_path=RECEIVER_CERT_PATH --
initiator_key_path=INITIATOR_KEY_PATH --initiator_cacert_path=INITIATOR_CACERT_PATH --
receiver_cn=RECEIVER_CN
```

For help and usage, run the script with **--help** / **-h** flag:

```
/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh -h
```

6.7.7.3 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.7.7.4 Reverting IPsec Configuration

To destroy IPsec configuration, run the following on both machines:

```
/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh -d
```

Make sure to run this command only after at least two minutes have passed from running the application on either machines. Otherwise, this may lead to errors.

If you run this command without initializing the connection first (in Running strongSwan Example), you may receive errors. These errors have no functional impact and may be safely ignored.

6.7.8 References

- **/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption.sh**
- **/opt/mellanox/doca/applications/east_west_overlay_encryption/east_west_overlay_encryption_params.json**

6.8 DOCA Eth L2 Forwarding Application Guide

This document provides an Ethernet L2 Forwarding implementation on top of the NVIDIA® BlueField® DPU.

6.8.1 Introduction

The Ethernet L2 Forwarding application is a DOCA Ethernet based application that forwards traffic from a single RX port to a single TX port and vice versa, leveraging DOCA's task/event batching feature for enhanced performance.

The application can run both on the host and the BlueField, and has two main modes:

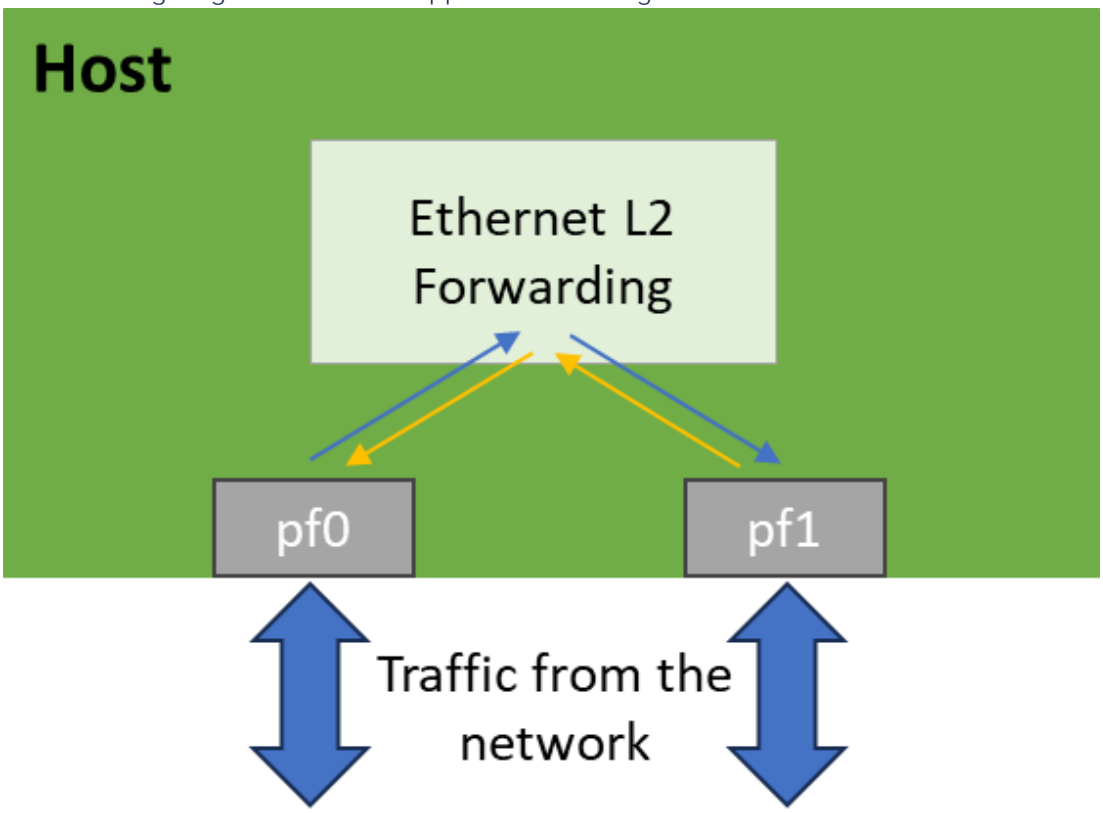
- Two-sided forwarding – device 1 → device 2 *and* device 2 → device 1
- One-sided forwarding – device 1 → device 2 *or* device 2 → device 1

The one-sided mode offers better performance, enlarging the packets forwarding rate.

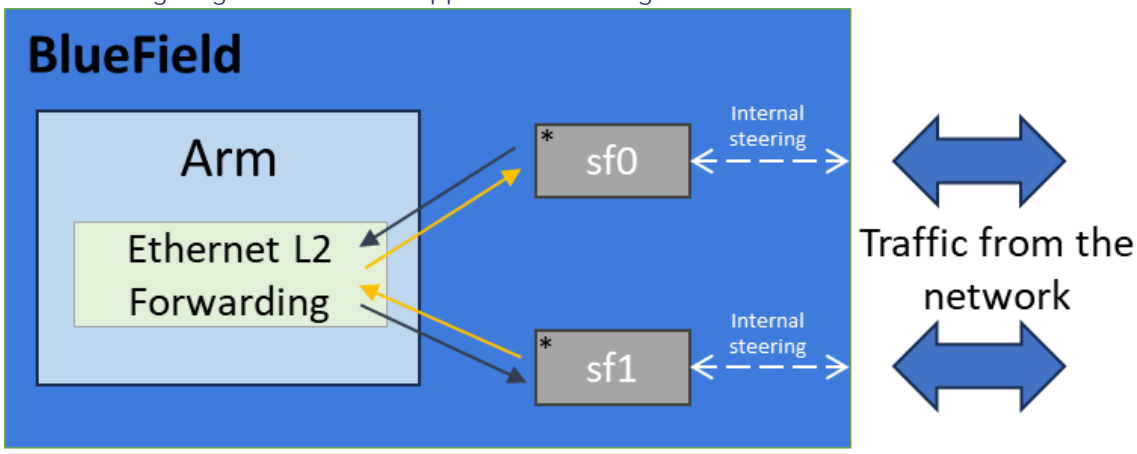
6.8.2 System Design

The Ethernet L2 Forwarding application runs on the host or the BlueField.

- The following diagram shows the application running on the host.



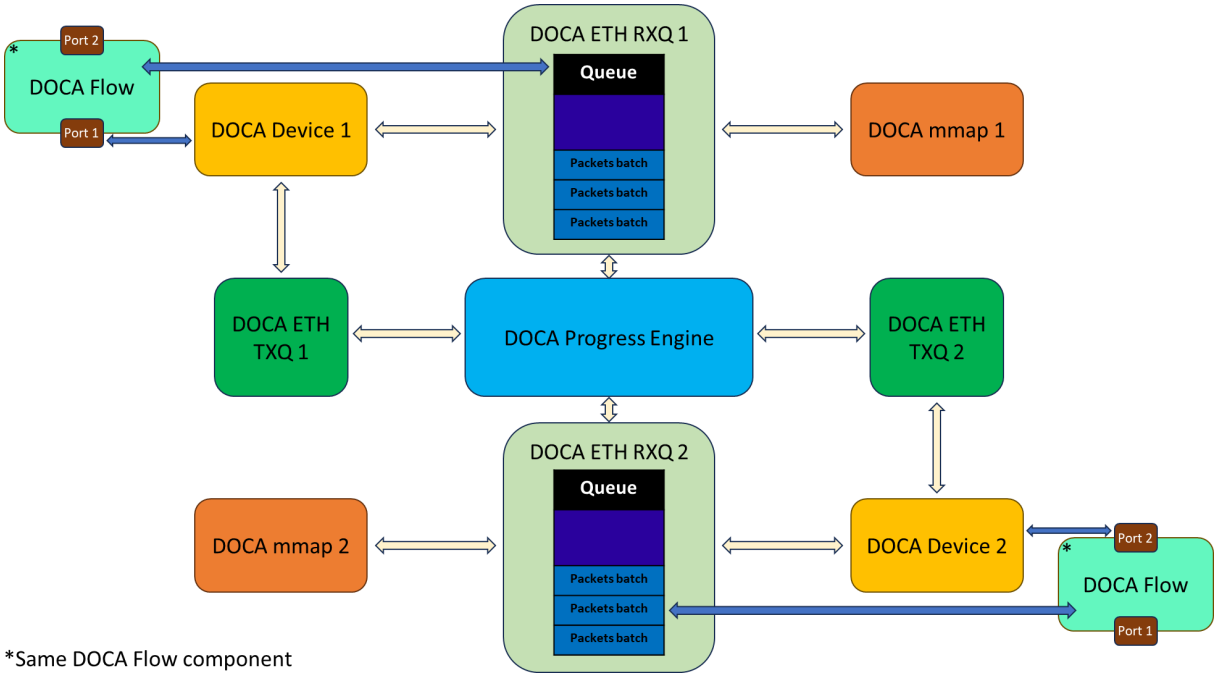
- The following diagram shows the application running on the BlueField:



*DOCA Ethernet library only handles SFs

6.8.3 Application Architecture

The Ethernet L2 Forwarding application runs on top of the DOCA Ethernet API to form an (two/one-sided) L2 forwarding between two ports.



*Same DOCA Flow component

1. Two DOCA devices are opened.
2. Two DOCA mmaps are created.
3. Two DOCA Flow ports are configured and started, each with a different opened DOCA device.

4. Two DOCA Ethernet TXQ and RXQ contexts are initialized, each TXQ-RXQ pair with a different opened DOCA device such that traffic is steered from the device to the corresponding RXQ, and from the corresponding TXQ to the device.
5. Forwarding - Packets received by device x are steered to RXQ x, then allocated to TXQ y and sent by device y (and vice versa).

6.8.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Ethernet
- DOCA Flow

For additional information about the used DOCA libraries, please refer to the respective programming guides.

6.8.5 Compiling the Application

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for the application is located in the following directory:

```
/opt/mellanox/doca/applications/eth_l2_fwd/
```

6.8.5.1 Compiling All Applications

The applications are all defined under a single meson project, meaning that the default compilation will compile all the DOCA applications.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_eth_l2_fwd will be created under **/tmp/build/eth_l2_fwd/** .

6.8.5.2 Compiling Only the Current Application

1. To directly build only the Ethernet L2 Forwarding application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_eth_l2_fwd=true
ninja -C /tmp/build
```

doca_eth_l2_fwd will be created under **/tmp/build/eth_l2_fwd/**.

2. Alternatively, one can set the desired flags in the **meson_options.txt** file instead of providing them in the compilation command line:
    a. Edit the following flags in **/opt/mellanox/doca/applications/meson_options.txt**:
        - Set **enable_all_applications** to **false**
        - Set **enable_eth_l2_fwd** to **true**
    b. The same compilation commands should be used, as were shown in the previous section:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_eth_l2_fwd will be created under **/tmp/build/eth_l2_fwd/**.

6.8.6 Running the Application

6.8.6.1 Application Execution

The Ethernet L2 Forwarding application is provided in source form, hence a compilation is required before the application can be executed.

- Application usage instructions:

```
Usage: doca_eth_l2_fwd [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -d, --devs-names <name1,name2>    Set two IB devices names separated by a comma, without spaces.
  -r, --rate <rate>                 Set packets receive rate (in [MB/s]), default is 12500.
  -ps, --pkt-size <size>            Set max packet size (in [B]), default is 1600.
  -t, --time <time>                 Set packet max process time (in [µs]), default is 1.
  -nt, --num-tasks <num>            Set number of tasks per batch, default is 128.
  -nb, --num-batches <num>          Set number of task batches, default is 32.
  -o, --one-sided-forwarding <num>  Set one-sided forwarding: 0 - two-sided forwarding, 1 - device 1 ->
device 2, 2 - device 2 -> device 1. default is 0.
  -f, --max-forwardings <num>       Set max forwardings after which the application run will end, default
 is 0, meaning no limit.
```

For additional information, please refer to the Command Line Flags section below.

The above usage printout can be printed to the command line using the **-h** (or **--help**) options:

```
./doca_eth_l2_fwd -h
```

- CLI example for running the application either on the BlueField or on the host:

```
./doca_eth_l2_fwd -d mlx5_0,mlx5_1
```

> **Both** IB devices identifiers (**mlx5_0, mlx5_1**) should match the identifiers of the desired IB devices.

6.8.6.2 Command Line Flags

6.8.6.2.1 General Flags

Short Flag	Long Flag	Description
-h	**--help**	Prints a help synopsis and exits
-v	**--version**	Prints program version information and exits
-l	**--log-level**	Sets the numeric log level for the application: • **10** – DISABLE • **20** – CRITICAL • **30** – ERROR • **40** – WARNING • **50** – INFO • **60** – DEBUG • **70** – TRACE (requires compilation with TRACE support)
N/A	**--sdk-log-level**	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	**--log-filter**	Filters logs from specific modules (comma-separated list)
-j	**--json**	Parses command-line flags from a specified input JSON file

> Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.8.6.2.2 Program Flags

Short Flag	Long Flag	Description
d	**devs-names**	Two IB devices names, separated by a comma, without spaces. ⚠ This is a mandatory flag.

Short Flag	Long Flag	Description
r	rate	The rate (in [MB/s]) in which the RX port is expected to receive traffic.
ps	pkt-size	The maximum size (in [B]) of a received packet.
t	time	The maximum time taking to process a single packet.
nt	num-tasks	The number of tasks to set per a single task batch.
nb	num-batches	The number of task batches to set for the TX side.
o	one-sided-forwarding	Flag to set one of 3 options: • 0 – Two-sided forwarding. • 1 – One-sided forwarding from device 1 to device 2. • 2 – One-sided forwarding from device 2 to device 1.
f	max-forwardings	The maximum number of forwarding

6.8.6.3 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.8.7 Application Code Flow

1. Parse application argument.
    a. Initialize Arg parser resources and register DOCA general parameters.

```
doca_argp_init();
```

    b. Register Ethernet L2 Forwarding application parameters.

```
register_eth_l2_forwarding_params();
```

    c. Parse the arguments.

```
doca_argp_start();
```

        i. Parse DOCA flags.
        ii. Parse application parameters.
2. Execute Ethernet L2 Forwarding application main logic.

```
eth_l2_fwd_execute();
```

    a. Open the two chosen DOCA devices.
    b. Initialize necessary DOCA Core objects.
    c. Initialize ETH RXQ/TXQ contexts for the devices.
    d. Forward packets.
3. Clean up application resources.

```
eth_l2_fwd_cleanup();
```

a. Stop all contexts and drain tasks.
b. Free all application resources.

4. Arg parser destroy.

```
doca_argp_destroy()
```

6.8.8 References

- **/opt/mellanox/doca/applications/eth_l2_fwd/**

6.9 DOCA File Compression Application Guide

This document provides a file compression implementation on top of the NVIDIA® BlueField® DPU.

6.9.1 Introduction

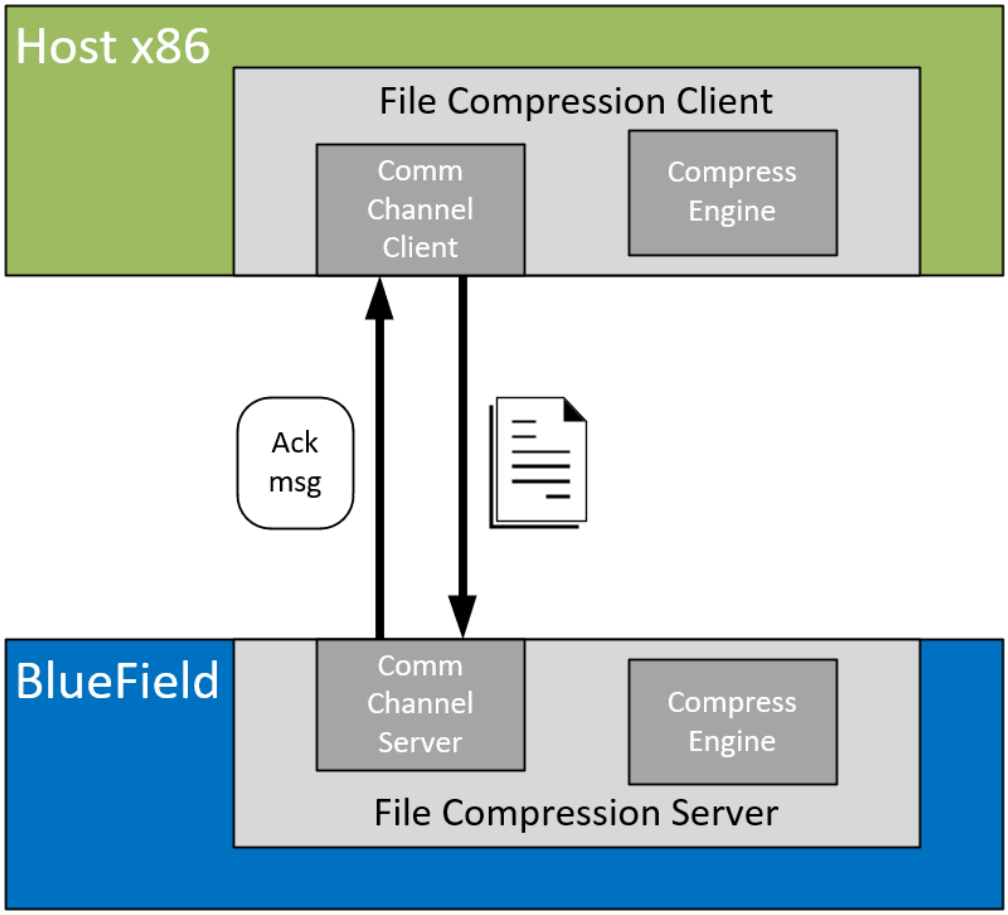
The file compression application exhibits how to use the DOCA Compress API to compress and decompress data using hardware acceleration as well as sending and receiving it using the DOCA Comch API.

The application's logic includes both a client and a server:

- Client side – the application opens a file, compresses it, and sends the checksum of the source file with the compressed data to the server
- Server side – the application saves the received file in a buffer, decompresses it, and compares the received checksum with the calculated one

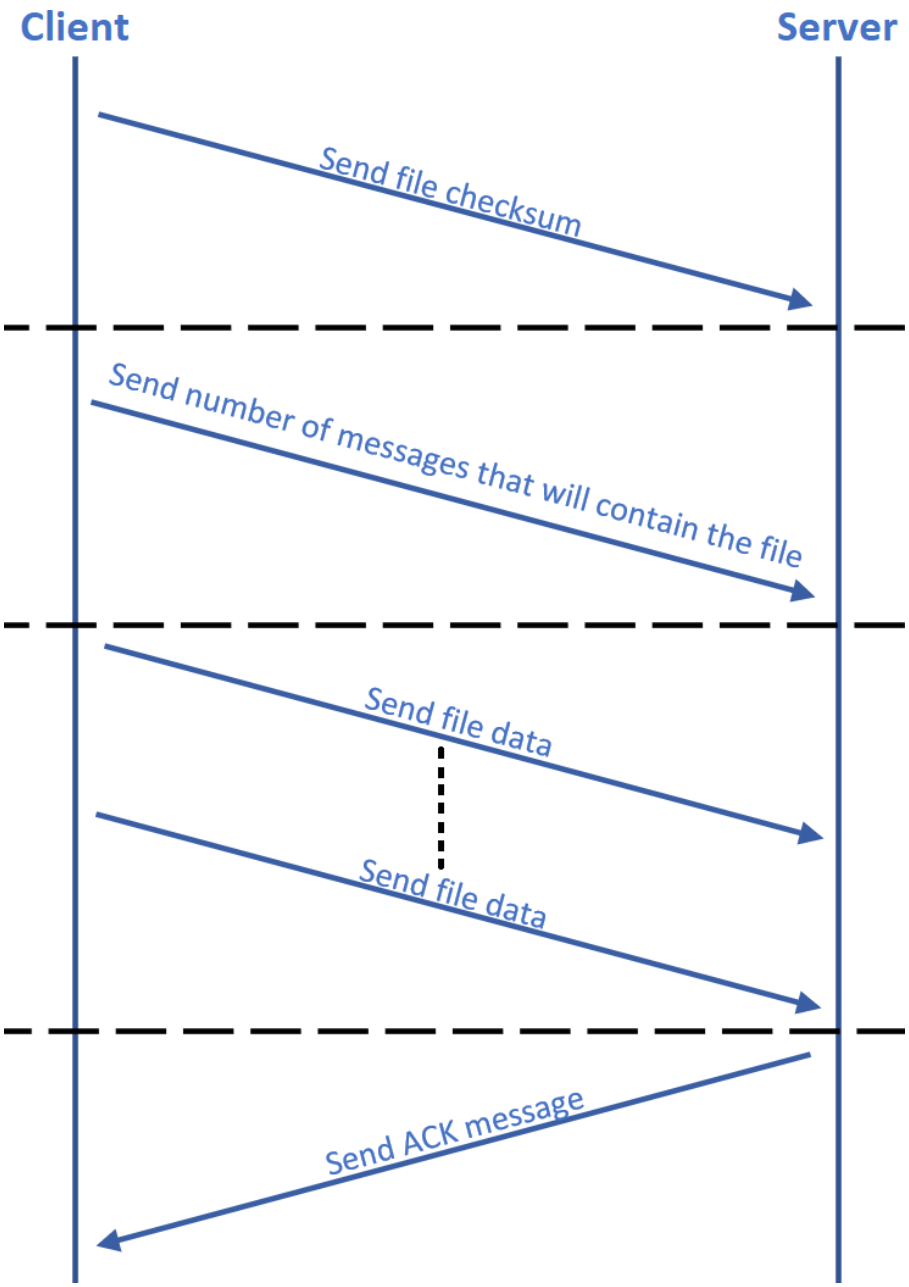
6.9.2 System Design

The file compression application client runs on the host and the server runs on the DPU.



6.9.3 Application Architecture

The file compression application runs on top of the DOCA Comm Channel API to send and receive the file from the host and to the DPU.



1. Connection is established on both sides by DOCA Comm Channel API.
2. Client compresses the data:
    - When compress engine is available – submits compress job with DOCA Compress API and sends the result to the server
    - When compress engine is unavailable – compresses the data in software
3. Client sends the number of messages needed to send the compressed content of the file.
4. Client sends data segments in size of up to 4080 bytes.
5. Server saves the received data in a buffer and submits a decompress job.
6. Server sends an ACK message to the client when all parts of the file are received successfully.

7. Server compares the received checksum to the calculated checksum.
8. Server writes the decompressed data to an output file.

6.9.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Compress
- DOCA Comch

Refer to their respective programming guide for more information.

6.9.5 Compiling the Application

> Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

> For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for the application is located in the following directory:

```
/opt/mellanox/doca/applications/file_compression/
```

6.9.5.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

> **doca_file_compression** is created under **/tmp/build/file_compression/**.

6.9.5.2 Compiling File Compression Application Only

To directly build only the file compression application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_file_compression=true
ninja -C /tmp/build
```

doca_file_compression is created under **/tmp/build/file_compression/**.

Alternatively, the user may set the desired flags in the **meson_options.txt** file instead of providing them in the compilation command line:

1. Edit the following flags in **/opt/mellanox/doca/applications/meson_options.txt**:
   - Set **enable_all_applications** to **false**
   - Set **enable_file_compression** to **true**
2. Run the following compilation commands:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_file_compression is created under **/tmp/build/file_compression/**.

6.9.6 Running the Application

6.9.6.1 Application Execution

The file compression application is provided in source form. Therefore, a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_file_compression [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -p, --pci-addr                    DOCA Comm Channel device PCI address
  -r, --rep-pci                     DOCA Comm Channel device representor PCI address
  -f, --file                        File to send by the client / File to write by the server
  -t, --timeout                     Application timeout for receiving file content messages, default is 5
 sec
```

This usage printout can be printed to the command line using the **-h** (or **--help**) options:

```
./doca_file_compression -h
```

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on BlueField:

```
./doca_file_compression -p 03:00.0 -r 3b:00.0 -f received.txt
```

Both the DOCA Comm Channel device PCIe address (**03:00.0**) and the DOCA Comm Channel device representor PCIe address (**3b:00.0**) should match the addresses of the desired PCIe devices.

3. CLI example for running the application on the host:

```
./doca_file_compression -p 3b:00.0 -f send.txt
```

The DOCA Comm Channel device PCIe address (**3b:00.0**) should match the address of the desired PCIe device.

6.9.6.2 Command Line Flags

6.9.6.2.1 General Flags

Short Flag	Long Flag	Description
-h	**--help**	Prints a help synopsis and exits
-v	**--version**	Prints program version information and exits
-l	**--log-level**	Sets the numeric log level for the application: • **10** – DISABLE • **20** – CRITICAL • **30** – ERROR • **40** – WARNING • **50** – INFO • **60** – DEBUG • **70** – TRACE (requires compilation with TRACE support)
N/A	**--sdk-log-level**	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	**--log-filter**	Filters logs from specific modules (comma-separated list)
-j	**--json**	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.9.6.2.2 Program Flags

Short Flag	Long Flag	Description
f	file	• For client – path to the file to be sent • For server – path to write the file into ⚠ This is a mandatory flag.
p	pci-addr	Comm Channel DOCA device PCIe address ⚠ This is a mandatory flag.
r	rep-pci	Comm Channel DOCA device representor PCIe address ⚠ This flag is mandatory only on the DPU.

6.9.6.3 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.9.7 Application Code Flow

1. Parse application argument.
    a. Initialize arg parser resources and register DOCA general parameters.

```
doca_argp_init();
```

    b. Register file compression application parameters.

```
register_file_compression_params();
```

    c. Parse the arguments.

```
doca_argp_start();
```

        i. Parse app parameters.
2. Set endpoint attributes.

```
set_endpoint_properties();
```

    a. Set maximum message size of 4080 bytes.
    b. Set maximum number of messages allowed.
3. Create comm channel endpoint.

```
doca_comm_channel_ep_create();
```

    a. Create endpoint for client/server.

4. Run client/server main logic.

```
file_compression_client/server();
```

5. Clean up the file compression application.

```
file_compression_cleanup();
```

    a. Free all application resources.

6. Arg parser destroy.

```
doca_argp_destroy()
```

6.9.8 References

- **/opt/mellanox/doca/applications/file_compression/**

6.10 DOCA File Integrity Application Guide

This guide provides a file integrity implementation on top of NVIDIA® BlueField® DPU.

6.10.1 Introduction

The file integrity application exhibits how to use the DOCA Comch and DOCA SHA libraries to send and receive a file securely.

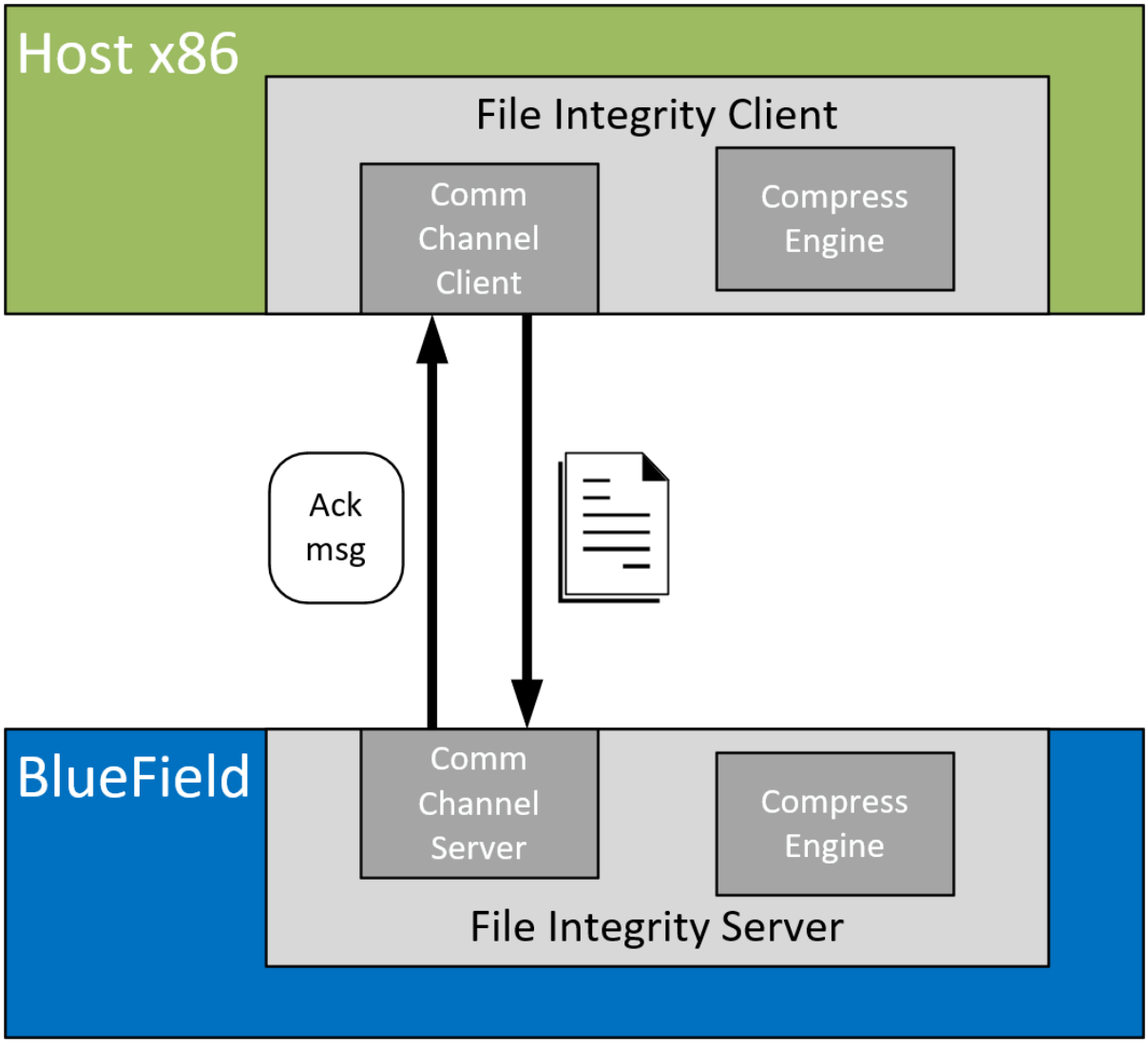
The application's logic includes both a client and a server:

- Client side – the application opens a file, calculates the SHA (secure hash algorithm) digest on it, and sends the digest of the source file alongside the file itself to the server
- Server side – the application calculates the SHA on the received file and compares the received digest to the calculated one to check if the file has been compromised

> ⚠ SHA hardware acceleration is only available on the BlueField-2 DPU. This application is not supported on BlueField-3.

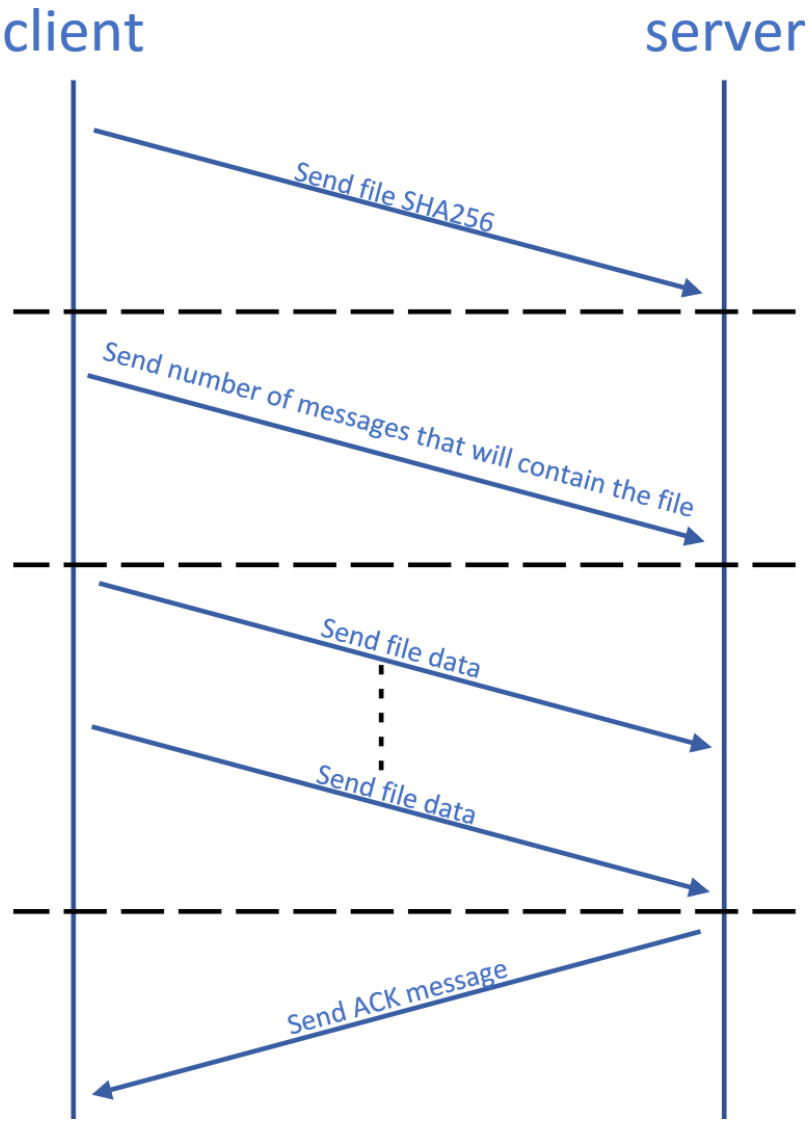
6.10.2 System Design

The file integrity application runs in client mode (host) and server mode (DPU).



6.10.3 Application Architecture

The file integrity application runs on top of the DOCA Comm Channel API to send and receive files from the host and DPU.



1. Connection is established on both sides by the Comm Channel API.
2. Client submits SHA job with the DOCA SHA library and sends the result to the server.
3. Client sends the number of messages required to send the content of the file.
4. Client sends data segments in size of up to 4032 bytes.
5. Server submits a partial SHA job on each received segment.
6. Server sends an ACK message to the client when all parts of the file are received successfully.
7. Server compares the received SHA to the calculated SHA.

6.10.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA SHA
- DOCA Comch

Refer to their respective programming guide for more information.

6.10.5 Compiling the Application

> Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

> For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for the application is located in the following directory:

```
/opt/mellanox/doca/applications/file_integrity/
```

6.10.5.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

> **doca_file_integrity** is created under **/tmp/build/file_integrity/** .

6.10.5.2 Compiling Only the Current Application

To directly build only the file integrity application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_file_integrity=true
ninja -C /tmp/build
```

> **doca_file_integrity** is created under **/tmp/build/file_integrity/** .

Alternatively, one can set the desired flags in the **meson_options.txt** file instead of providing them in the compilation command line:

1. Edit the following flags in **/opt/mellanox/doca/applications/meson_options.txt** :
   - Set **enable_all_applications** to **false**
   - Set **enable_file_integrity** to **true**
2. Run the following compilation commands:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_file_integrity is created under **/tmp/build/file_integrity/**.

6.10.6 Running the Application

6.10.6.1 Application Execution

The file integrity application is provided in source form. Therefore, a compilation is required before the application can be executed.

- Application usage instructions:

```
Usage: doca_file_integrity [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -p, --pci-addr                    DOCA Comm Channel device PCI address
  -r, --rep-pci                     DOCA Comm Channel device representor PCI address
  -f, --file                        File to send by the client / File to write by the server
  -t, --timeout                     Application timeout for receiving file content messages, default is 5
 sec
```

This usage printout can be printed to the command line using the **-h** (or **--help**) options:

```
./doca_file_integrity -h
```

For additional information, refer to section "Command Line Flags".

- CLI example for running the application on BlueField:

```
./doca_file_integrity -p 03:00.0 -r 3b:00.0 -f received.txt
```

Both the DOCA Comm Channel device PCIe address (**03:00.0**) and the DOCA Comm Channel device representor PCIe address (**3b:00.0**) should match the addresses of the desired PCIe devices.

- CLI example for running the application on the host:

```
./doca_file_integrity -p 3b:00.0 -f send.txt
```

> ⚠ The DOCA Comm Channel device PCIe address (`3b:00.0`) should match the address of the desired PCIe device.

6.10.6.2 Command Line Flags

6.10.6.2.1 General Flags

Short Flag	Long Flag	Description
`-h`	`--help`	Prints a help synopsis and exits
`-v`	`--version`	Prints program version information and exits
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

> ℹ Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.10.6.2.2 Host Program Flags

Short Flag	Long Flag	Description
`f`	`file`	• For client – path to the file to be sent • For server – path to write the file into ⚠ This is a mandatory flag.
`p`	`pci-addr`	Comm Channel DOCA device PCIe address ⚠ This is a mandatory flag.

Short Flag	Long Flag	Description
r	rep-pci	Comm Channel DOCA device representor PCIe address ⚠ This flag is mandatory only on the DPU.

6.10.6.3 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.10.7 Application Code Flow

1. Parse application argument.
    a. Initialize the arg parser resources and register DOCA general parameters.

```
doca_arg_init();
```

    b. Register file integrity application parameters.

```
register_file_integrity_params();
```

    c. Parse application parameters.

```
doca_argp_start();
```

2. Set endpoint attributes.

```
set_endpoint_properties();
```

    a. Set maximum message size of 4032 bytes.
    b. Set number of maximum messages allowed per connection.
3. Create Comm Channel endpoint.

```
doca_comm_channel_ep_create();
```

    a. Create endpoint for client/server.
4. Create SHA context.

```
doca_sha_create();
```

    a. Create SHA context for submitting SHA jobs for client/server.
5. Run client/server main logic.

```
file_integrity_client/server();
```

6. Clean up the File Integrity app.

```
file_integrity_cleanup();
```

a. Free all application resources.

6.10.8 References

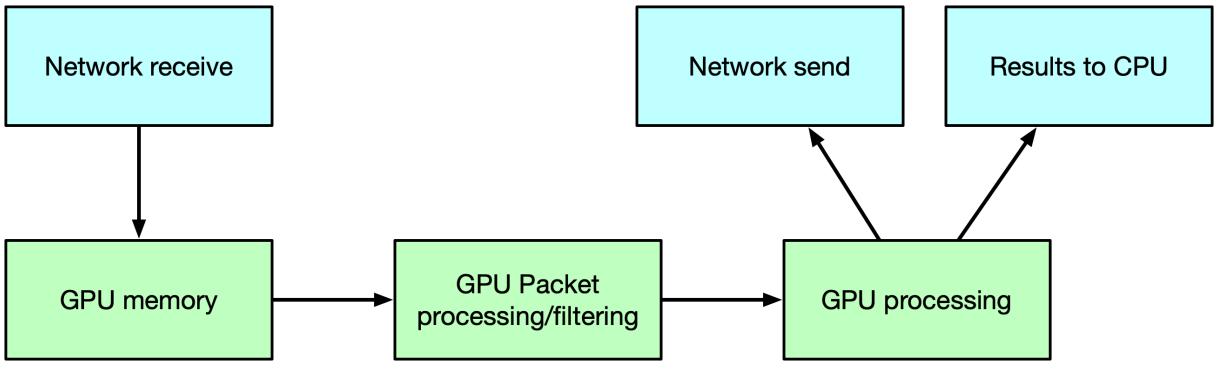
- `/opt/mellanox/doca/applications/file_integrity/`

6.11 DOCA GPU Packet Processing Application Guide

This guide provides a description of the GPU packet processing application to demonstrate the use of DOCA GPUNetIO, DOCA Ethernet, and DOCA Flow libraries to implement a GPU traffic analyzer.

6.11.1 Introduction

Real-time GPU processing of network packets is a useful technique to several different application domains, including signal processing, network security, information gathering, and input reconstruction. The goal of these applications is to realize an inline packet processing pipeline to receive packets in GPU memory (without staging copies through CPU memory), process them in parallel with one or more CUDA kernels, and then run inference, evaluate, or send the result of the calculation over the network.

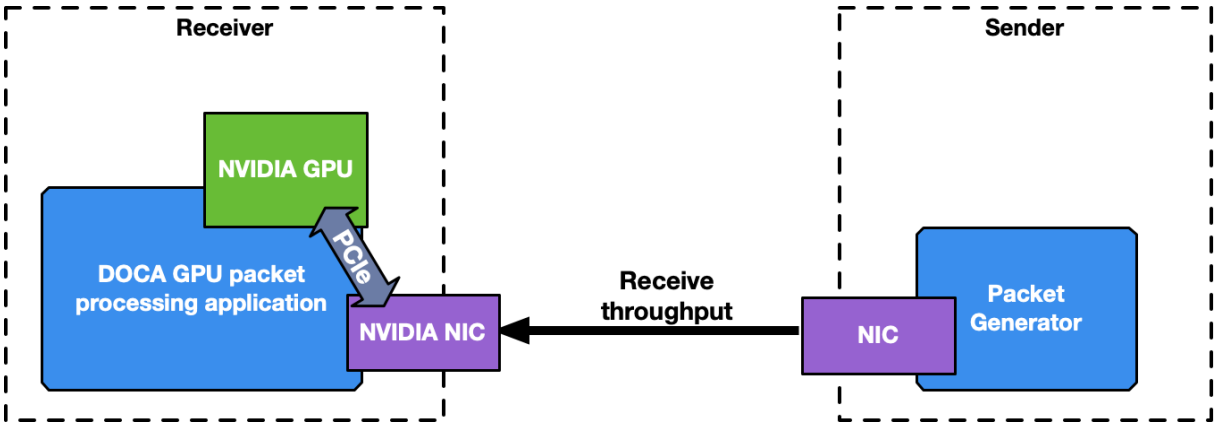


The type of data processing heavily depends on the use case. The goal of this application is to provide a basic layout to reuse in the most common use cases of being able to receive, differentiate and manage the following types of network traffic in multiple queues: UDP, TCP and ICMP.

This application is an enhancement of the use cases presented in this NVIDIA blog post about DOCA GPUNetIO.

6.11.2 System Design

This is a receive-and-process DOCA application, so a packet generator sending packets is required to test the application.



To launch the application, the PCIe address of the GPU and NIC are required.

6.11.3 Application Architecture

The application manages different types of traffic differently, dedicating up to 4 receive queues to each one using DOCA Flow with RSS mode to assign each packet to the right queue. The more queues the application uses, the higher is the degree of parallelism in how receive data is processed and how long it takes.

It is highly recommended to use more than one receive queue for 100Gb/s or higher network traffic throughput.

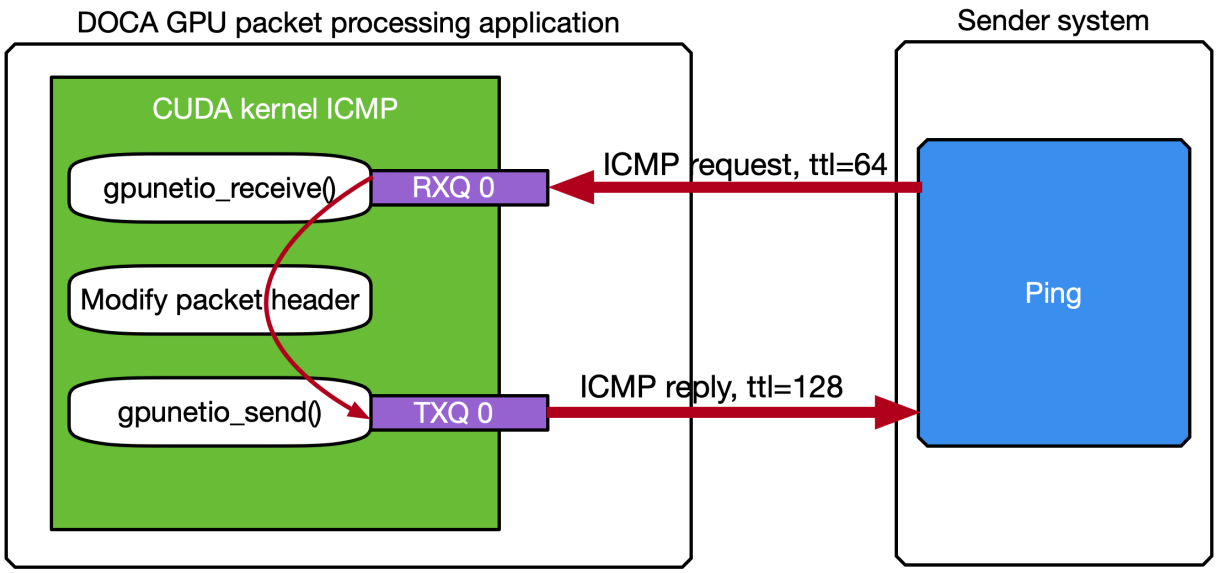
6.11.3.1 ICMP Network Traffic

If the network interface used for the application has an IP address, it is possible to ping that interface. ICMP packets are received by a dedicated CUDA kernel (file **gpu_kernels/receive_icmp.cu**) which:

1. Receives packets using the DOCA GPUNetIO CUDA warp-level function **doca_gpu_dev_eth_rxq_receive_warp**.
2. Checks if the packet is an ICMP echo request.
3. Forwards the same packet, modifying some header info (e.g., swapping MAC and IP addresses, changing ICMP packet type).
4. Pushes the modified packet into the send queue using the DOCA GPUNetIO thread-level function **doca_gpu_dev_eth_txq_send_enqueue_strong**.
5. Sends the packet using the DOCA GPUNetIO thread-level functions **doca_gpu_dev_eth_txq_commit_strong** and **doca_gpu_dev_eth_txq_push**.

This is not a compute intensive use case, so a single CUDA warp with only one receive queue and one send queue is enough to keep up with a decent latency.

By default, the OS CPU ping TTL is set to 64. Therefore, to be sure the GPU is actually replying to ICMP ping requests, TTL is set to 128 in this application.



The following are motivations for this use case:

- Providing an easy tool to check connectivity between packet the generator machine and the DOCA application machine
- Having a sense of network latency between the two machines using a well-known tool like ping
- Showing an easy way to receive and forward modified packets
- Providing a warp-level implementation of a CUDA kernel receiving and forwarding traffic

Assuming the IP address of the network interface to ping is `192.168.1.1`, this is the expected output:

```
$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=0.324 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=64 time=0.332 ms
64 bytes from 192.168.1.1: icmp_seq=3 ttl=64 time=0.299 ms
64 bytes from 192.168.1.1: icmp_seq=4 ttl=64 time=0.309 ms
64 bytes from 192.168.1.1: icmp_seq=5 ttl=64 time=0.323 ms
64 bytes from 192.168.1.1: icmp_seq=6 ttl=64 time=0.300 ms
64 bytes from 192.168.1.1: icmp_seq=7 ttl=64 time=0.274 ms
64 bytes from 192.168.1.1: icmp_seq=8 ttl=64 time=0.314 ms
64 bytes from 192.168.1.1: icmp_seq=9 ttl=64 time=0.327 ms
64 bytes from 192.168.1.1: icmp_seq=10 ttl=64 time=0.384 ms
# At this point, the DOCA application has been started on the 192.168.1.1 interface
# TTL becomes 128 as it's the GPU replying to ICMP requests now instead of the OS
64 bytes from 192.168.1.1: icmp_seq=11 ttl=128 time=0.346 ms
64 bytes from 192.168.1.1: icmp_seq=12 ttl=128 time=0.274 ms
64 bytes from 192.168.1.1: icmp_seq=13 ttl=128 time=0.294 ms
64 bytes from 192.168.1.1: icmp_seq=14 ttl=128 time=0.240 ms
64 bytes from 192.168.1.1: icmp_seq=15 ttl=128 time=0.273 ms
64 bytes from 192.168.1.1: icmp_seq=16 ttl=128 time=0.238 ms
64 bytes from 192.168.1.1: icmp_seq=17 ttl=128 time=0.252 ms
64 bytes from 192.168.1.1: icmp_seq=18 ttl=128 time=0.232 ms
64 bytes from 192.168.1.1: icmp_seq=19 ttl=128 time=0.278 ms
......
```

A DOCA Progress Engine is attached to the DOCA Ethernet Txq context used to forward ICMP packets. Those packets are sent from the GPU with the `DOCA_GPU_SEND_FLAG_NOTIFY` flag, which result in creating a notification after every packet is sent by the NIC.

All the notifications are then analyzed by the CPU through the **doca_pe_progress** function. The final effect is the output of the application which returns the distance, in seconds, between two pings. The following is an example with a ping every 0.5 seconds:

```
$ ping -i 0.5 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=128 time=0.202 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=128 time=0.179 ms
64 bytes from 192.168.1.1: icmp_seq=3 ttl=128 time=0.199 ms
64 bytes from 192.168.1.1: icmp_seq=4 ttl=128 time=0.180 ms
64 bytes from 192.168.1.1: icmp_seq=5 ttl=128 time=0.200 ms
64 bytes from 192.168.1.1: icmp_seq=6 ttl=128 time=0.189 ms
......
```

On the DOCA side, the application should print a log for all the ICMP packets received and retransmitted:

```
Seconds 5
[UDP] QUEUE: 0 DNS: 0 OTHER: 0 TOTAL: 0
[TCP] QUEUE: 0 HTTP: 0 HTTP HEAD: 0 HTTP GET: 0 HTTP POST: 0 TCP [SYN: 0 FIN: 0 ACK: 0] OTHER: 0 TOTAL: 0
[13:54:19:202061][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 3 sent at 1702302859201997120 time from last ICMP is 0.512025 sec
[13:54:19:713960][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 4 sent at 1702302859713896620 time from last ICMP is 0.511899 sec
[13:54:20:225891][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 5 sent at 1702302860225868072 time from last ICMP is 0.511971 sec
[13:54:20:737823][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 6 sent at 1702302860737781760 time from last ICMP is 0.511914 sec
[13:54:21:249763][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 7 sent at 1702302861249723044 time from last ICMP is 0.511941 sec
[13:54:21:761614][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 8 sent at 1702302861761588848 time from last ICMP is 0.511866 sec
[13:54:22:273689][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 9 sent at 1702302862273643536 time from last ICMP is 0.512055 sec
[13:54:22:785543][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 10 sent at 1702302862785527576 time from last ICMP is 0.511884 sec
[13:54:23:297545][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 11 sent at 1702302863297501448 time from last ICMP is 0.511974 sec
[13:54:23:809406][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 12 sent at 1702302863809350664 time from last ICMP is 0.511849 sec

Seconds 10
[UDP] QUEUE: 0 DNS: 0 OTHER: 0 TOTAL: 0
[TCP] QUEUE: 0 HTTP: 0 HTTP HEAD: 0 HTTP GET: 0 HTTP POST: 0 TCP [SYN: 0 FIN: 0 ACK: 0] OTHER: 0 TOTAL: 0
[13:54:24:321405][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 13 sent at 1702302864321391148 time from last ICMP is 0.512040 sec
[13:54:24:833338][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 14 sent at 1702302864833270356 time from last ICMP is 0.511879 sec
[13:54:25:345302][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 15 sent at 1702302865345282728 time from last ICMP is 0.512012 sec
[13:54:25:857199][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 16 sent at 1702302865857133664 time from last ICMP is 0.511851 sec
[13:54:26:369131][2688665][DOCA][INF][gpu_packet_processing.c:77][debug_send_packet_icmp_cb] ICMP debug event:
Queue 0 packet 17 sent at 1702302866369128728 time from last ICMP is 0.511995 sec......
```

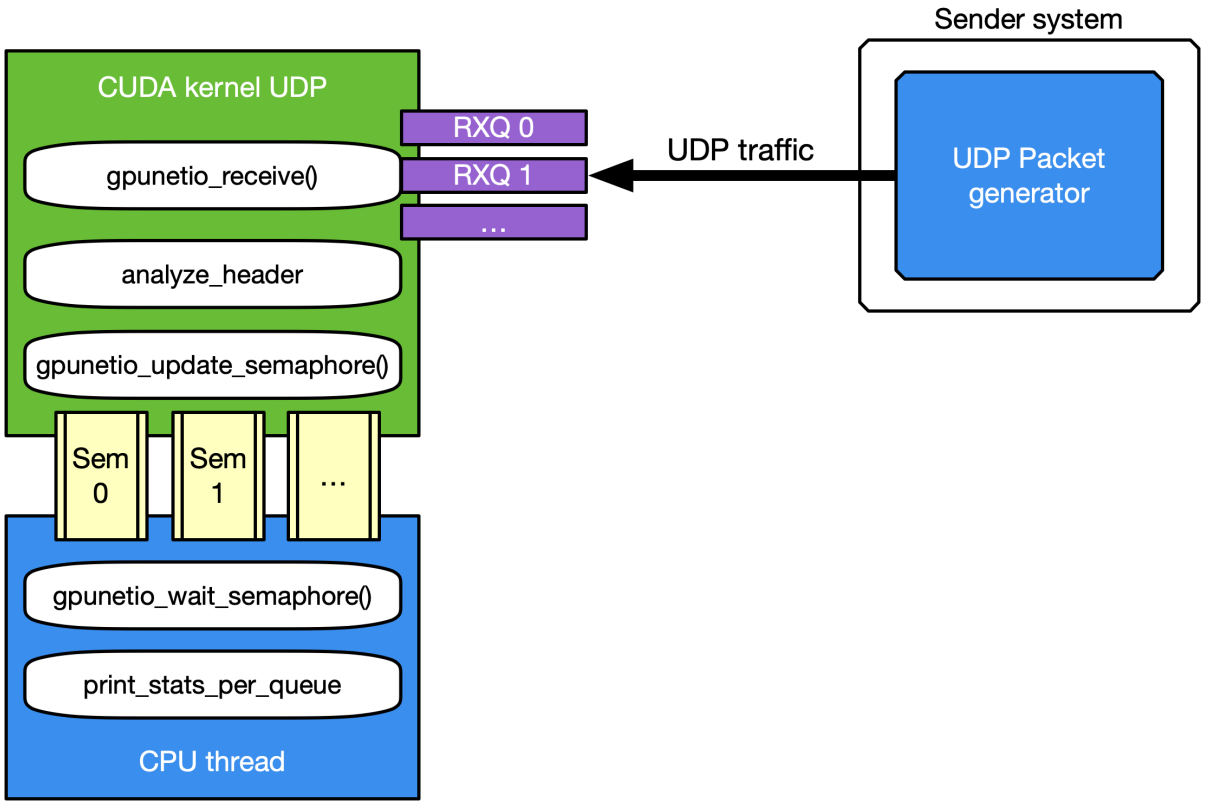
6.11.3.2 UDP Network Traffic

This is the most generic use case of receive-and-analyze packet headers. Designed to keep up with 100Gb/s of incoming network traffic, the CUDA kernel responsible for the UDP traffic dedicates one CUDA block of 512 CUDA threads (file **gpu_kernels/receive_udp.cu**) to a different Ethernet UDP receive queue.

The data path loop is:

1. Receive packets using the DOCA GPUNetIO CUDA block-level function **doca_gpu_dev_eth_rxq_receive_block**.
2. Each CUDA thread works on a subset of received packets.
3. DOCA buffer containing the packet is retrieved.

4. Packet payload is analyzed to differentiate between DNS packets from other UDP generic packets.
5. Packet payload is wiped-out to ensure that old stale packets are not analyzed again.
6. Each CUDA block reports to the CPU thread statistics about types of received packets through a DOCA GPUNetIO semaphore.
7. CPU thread polls on semaphores to retrieve and print the statistics to the console.



The motivation for this use case is mostly to provide an application template to:

- Receive and analyze packet headers to differentiate across different UDP protocols
- Report statistics to the CPU through the DOCA GPUNetIO semaphore
- Report UDP ingress data rate the application can keep up per receive queue

Several packet generators can be used to test this mode like T-Rex or DPDK testpmd.

The application measures the incoming data rate based on the amount of received bytes since the start of the application. To have a meaningful data rate report, the packet generator should start to send packets before starting the GPU packet processing application.

6.11.3.2.1 T-Rex Packet Generator

This is a valuable packet generator that can send or receive Ethernet packets and analyze network traffic metrics and performance.

The recipe to build and run is:

T-Rex

```
wget https://trex-tgn.cisco.com/trex/release/v3.08.tar.gz --no-check-certificate
tar -xvf v3.08.tar.gz
cd v3.08
# Only first time
sudo ./dpdk_setup_ports.py -i
```

This creates a configuration file named **trex_cfg.yaml**. It may be necessary to manually adjust the interface MTU in the config file setting it explicitly. For example:

T-Rex MTU

```
### Config file generated by dpdk_setup_ports.py ###
- version: 2
  interfaces: ['e5:00.0', 'e5:00.1']
  port_mtu: 9000
  port_info:...
```

To launch T-Rex:

T-Rex MTU

```
# On one window/session
sudo ./t-rex-64 -i -c 8 --cfg /etc/trex_cfg.yaml --no-scapy-server
# On second window/session
./trex-console
tui
```

Once the console is started, T-Rex can send or receive packets.

For example:

T-Rex command line

```
# Send 1kB packets at 100Gbps
start -f stl/bench.py -p 0 -m 100gbps --force -t size=1024,vm=var1
# Send 128B packets at 90Gbps
start -f stl/bench.py -p 0 -m 90gbps --force -t size=128,vm=var1
...
```

6.11.3.2.2 DPDK testpmd Packet Generator

Already included in the DPDK shipped with DOCA, an example command line to run **dpdk-testpmd** packet generator.

dpdk-testpmd

```
sudo sh -c 'echo 8192 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages'
sudo /opt/mellanox/dpdk/bin/dpdk-testpmd \
    -a <NIC PCIe address> \
    -l 0-3 \
    --socket-mem=4096 \
    -- \
    --forward-mode=txonly \
    --txpkts=1024 \
    --txq=4 --rxq=4 \
    --nb-cores=3 \
    --eth-peer=0,<DEST MAC ADDRESS> \
    --total-num-mbufs=131072 \
    --txonly-multi-flow \
    -i
```

6.11.3.3 TCP Network Traffic and HTTP Echo Server

By default, the TCP flow management is the same as UDP: Receive TCP packets and analyze their headers to report to the CPU statistics about the types of received packets. This is good for passive traffic analyzers or sniffers but sometimes a packet processing application requires receiving packets directly from TCP peers which implies the establishment of a TCP-reliable connection through the 3-way handshake method. Therefore, it is possible to enable TCP "server" mode through the `-s` command-line flag which enables an "HTTP echo server" mode where the CPU and GPU cooperate to establish a TCP connection and process TCP data packets.

Specifically, in this case there are two different sets of receive queues:

- CPU DPDK receive queues which receive TCP "control" packets (e.g. SYN, FIN or RST)
- DOCA GPUNetIO receive queues to receive TCP "data" packets

This distinction is possible thanks to DOCA Flow capabilities.

The application's flow requires CPU and GPU collaboration as described in the following subsections.

6.11.3.3.1 Step 1: TCP Connection Establishment

A CPU thread through DPDK queues receives a TCP SYN packet from a remote TCP peer. The CPU thread establishes a TCP reliable connection (replies with a TCP SYN-ACK packet) with the peer and uses DOCA Flow to create a new steering rule to redirect TCP data packets to one of the DOCA GPUNetIO receive queues. The new steering rule excludes control packets (e.g., SYN, FIN or RST).

6.11.3.3.2 Step 2: TCP Data Processing

The CUDA kernel responsible for TCP processing receives TCP data packets and performs TCP packet header analysis. If it receives an HTTP GET request, it stores the relevant packet's info in the next item of a DOCA GPUNetIO semaphore, setting it to `READY`.

6.11.3.3.3 Step 3: HTTP Echo Server

A second CUDA kernel responsible for HTTP processing polls the DOCA GPUNetIO semaphore. Once it detects the update of the next item to `READY`, it reads the HTTP GET packet info and crafts an HTTP response packet with an HTML page.

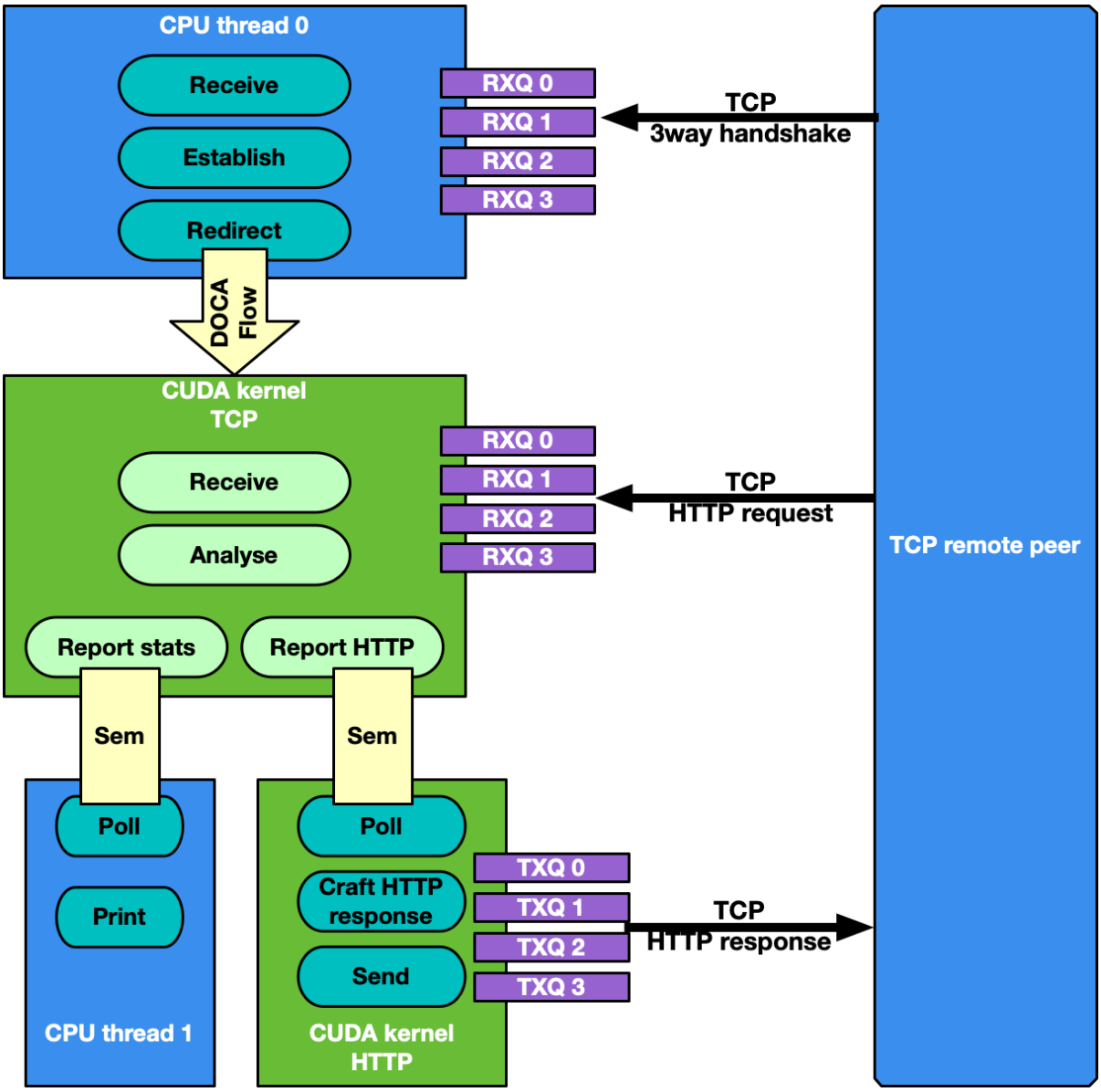
If the request is about **index.html** or **contacts.html**, the CUDA kernel replies with the appropriate HTML page using a **200 OK** code. For all other requests, the it returns a "Page not found" and **404 Error** code.

HTTP response packets are sent by this second HTTP CUDA kernel using DOCA GPUNetIO.

> ⚠ Care must be taken to maintain TCP sequence/ack numbers in the packet headers.

6.11.3.3.4 Step 4: TCP Connection Closure

If the CPU receives a TCP FIN packet through the DPDK queues, it closes the connection with the remote TCP peer and removes the DOCA Flow rule from the DOCA GPUNetIO queues so the CUDA kernel cannot receive anymore packets from that TCP peer.



Motivations for this use case:

- Receiving and analyzing packet headers to differentiate across different TCP protocols
- Processing TCP packets on GPU in passive mode (sniffing) and active mode (reliable connection)
- Having a DOCA-DPDK application able to establish a TCP reliable connection without using any OS socket and bypassing kernel routines
- Having CUDA-kernel-to-CUDA-kernel communication through a DOCA GPUNetIO semaphore
- Showing how to create and send a packet from scratch with DOCA GPUNetIO

Assuming the network interface used to run the application has the IP address `192.168.1.1`, it is possible to test this HTTP echo server mode using simple tools like `curl` or `wget`.

Example with **curl**:

```
$ curl http://192.168.1.1/index.html -ivvv

*   Trying 192.168.1.1:80...
* Connected to 192.168.1.1 (192.168.1.1) port 80 (#0)
> GET /index.html HTTP/1.1
> Host: 192.168.1.1
> User-Agent: curl/7.81.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
HTTP/1.1 200 OK
< Date: Sun, 30 Apr 2023 20:30:40 GMT
Date: Sun, 30 Apr 2023 20:30:40 GMT
< Content-Type: text/html; charset=UTF-8
Content-Type: text/html; charset=UTF-8
< Content-Length: 158
Content-Length: 158
< Last-Modified: Sun, 30 Apr 2023 22:38:34 GMT
Last-Modified: Sun, 30 Apr 2023 22:38:34 GMT
< Server: GPUNetIO
Server: GPUNetIO
< Accept-Ranges: bytes
Accept-Ranges: bytes
< Connection: keep-alive
Connection: keep-alive
< Keep-Alive: timeout=5
Keep-Alive: timeout=5

<
<html>
  <head>
    <title>GPUNetIO index page</title>
  </head>
  <body>
    <p>Hello World, the GPUNetIO server Index page!</p>
  </body>
</html>

* Connection #0 to host 192.168.1.1 left intact
```

6.11.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA GPUNetIO
- DOCA Ethernet
- DOCA Flow

Refer to their respective programming guide for more information on system configuration and requirements.

Refer to the DOCA Installation Guide for Linux for details on how to install DOCA package software.

6.11.5 Dependencies

Before running the application make sure to have the following:

- **gdrdrv** kernel module – active and running on the system
- **dmabuf or nvidia-peermem**
- Network card interface you want to use is up

6.11.6 Compiling the Application

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for the application is located in the following directory:

```
/opt/mellanox/doca/applications/gpu_packet_processing/
```

6.11.6.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_gpu_packet_processing is created under **/tmp/build/gpu_packet_processing/** .

6.11.6.2 Compiling Only the Current Application

To directly build only the GPU packet processing application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_gpu_packet_processing=true
ninja -C /tmp/build
```

doca_gpu_packet_processing is created under **/tmp/build/gpu_packet_processing/** .

Alternatively, users can set the desired flags in the **meson_options.txt** file instead of providing them in the compilation command line:

1. Edit the following flags in **/opt/mellanox/doca/applications/meson_options.txt** :
   - Set **enable_all_applications** to **false**
   - Set **enable_gpu_packet_processing** to **true**
2. Run the following compilation commands:

   ```
   cd /opt/mellanox/doca/applications/
   meson /tmp/build
   ninja -C /tmp/build
   ```

   doca_gpu_packet_processing is created under **/tmp/build/gpu_packet_processing/** .

6.11.7 Running the Application

The GPU packet processing application is provided in source form. Therefore, a compilation is required before the application can be executed.

- Application usage instructions:

```
Usage: doca_gpu_packet_processing [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -g, --gpu <GPU PCIe address>      GPU PCIe address to be used by the app
  -n, --nic <NIC PCIe address>      DOCA device PCIe address used by the app
  -q, --queue <GPU receive queues>  DOCA GPUNetIO receive queue per flow
  -s, --httpserver <Enable GPU HTTP server> Enable GPU HTTP server mode
```

> This usage printout can be printed to the command line using the **-h** (or **--help**) options:
>
> ```
> ./doca_gpu_packet_processing -h
> ```

> For additional information, refer to section "Command Line Flags".

- CLI example for running the application on the host:
  a. Assuming a GPU PCIe address **ca:00.0** and NIC PCIe address **17:00.0** with 2 GPUNetIO receive queues:

```
./doca_gpu_packet_processing -n 17:00.0 -g ca:00.0 -q 2
```

> Refer to section "Running DOCA Application on Host" in the DOCA Virtual Functions.

6.11.7.1 Command Line Flags

6.11.7.1.1 General Flags

Short Flag	Long Flag	Description
-h	**--help**	Prints a help synopsis and exits
-v	**--version**	Prints program version information and exits

Short Flag	Long Flag	Description
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.11.7.1.2 Program Flags

Short Flag	Long Flag	Description
`g`	`gpu`	GPU PCIe address in `<bus>:<device>.<function>` format. This can be obtained using the `nvidia-smi` or `lspci` commands.
`n`	`nic`	Network card port PCIe address in `<bus>:<device>.<function>` format. This can be obtained using the `lspci` command.
`q`	`queue`	Number of receive queues to use in the example. Default is 1, maximum allowed is 4.
`s`	`httpserver`	Enable the TCP HTTP server mode. With this flag, TCP packets are not received by GPUNetIO as regular sniffer as it requires a TCP 3-way handshake to establish a reliable connection first.

6.11.7.2 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.11.8 Application Code Flow

The following explains the application's flow, highlighting main code blocks and functions:

1. Parse application argument.

```
doca_argp_init();
register_application_params();
doca_argp_start();
```

2. Initialize network device as DOCA device, initialize DPDK, and get device DPDK port ID.

```
init_doca_device();
```

   Calls **rte_eal_init()** with empty flags to initialize EAL resources.

3. Initialize a GPU device, creating a DOCA GPUNetIO handle for it.

```
doca_gpu_create();
```

4. Initialize DOCA Flow, starting the DPDK port.

```
init_doca_flow();
```

   Flags to initialize DOCA Flow are VNF, HW steering, and isolated mode (to prevent the default RSS flows from interfering with the GPUNetIO queues).

5. Create RX and TX queue related objects (i.e., Ethernet handlers, GPUNetIO handlers, flow rules, semaphores) to manage UDP, TCP and ICMP flows.

```
create_udp_queues();
create_tcp_queues();
create_icmp_queues();
/* Depending on TCP mode (HTTP server or not) properly connect different DOCA Flow pipes */
create_root_pipe();
```

6. Allocate generic exit flag. All CUDA kernels periodically poll on this flag. If the CPU set it to 1, CUDA kernels exit from their main loop and return.

```
doca_gpu_mem_alloc(gpu_dev, sizeof(uint32_t), alignment, DOCA_GPU_MEM_GPU_CPU, (void
 **)&gpu_exit_condition, (void **)&cpu_exit_condition);
```

7. Launch CUDA kernels, each on a different stream.

```
kernel_receive_udp(rx_udp_stream, gpu_exit_condition, &udp_queues);
kernel_receive_tcp(rx_tcp_stream, gpu_exit_condition, &tcp_queues, app_cfg.http_server);
kernel_receive_icmp(rx_icmp_stream, gpu_exit_condition, &icmp_queues);
if (app_cfg.http_server)
    kernel_http_server(tx_http_server, gpu_exit_condition, &tcp_queues, &http_queues);
```

8. Launch the CPU thread responsible to poll on DOCA GPUNetIO semaphores and print UDP and TCP stats on the console.

```
rte_eal_remote_launch((void *)stats_core, NULL, current_lcore);
```

9. Launch CPU thread responsible for managing TCP 3-way handshake connections.

```
if (app_cfg.http_server) {
    ...
    rte_eal_remote_launch(tcp_cpu_rss_func, &tcp_queues, current_lcore);
}
```

10. Wait for the user to send a signal to quit the application. When this happens, the signal handler function sets the **force_quit** flag to true which causes the main thread to move forward and set the exit condition to 1.

```
while (DOCA_GPUNETIO_VOLATILE(force_quit) == false);
```

```
DOCA_GPUNETIO_VOLATILE(*cpu_exit_condition) = 1;
```

11. Wait for CUDA kernels to exit and finalize all DOCA Flow and GPUNetIO resources.

```
cudaStreamSynchronize(rx_udp_stream);
cudaStreamSynchronize(rx_tcp_stream);
cudaStreamSynchronize(rx_icmp_stream);
if (app_cfg.http_server)
    cudaStreamSynchronize(tx_http_server);
destroy_flow_queue();
doca_gpu_destroy();
```

6.11.9 References

- **/opt/mellanox/doca/applications/gpu_packet_processing/**

6.12 NVIDIA DOCA IP Fragmentation Application Guide

This guide provides an implementation of IP fragmentation application on top of NVIDIA® BlueField®-3 DPU.

6.12.1 Introduction

This IP Fragmentation application is designed to handle IP fragmentation and reassembly efficiently, ensuring minimal processing overhead for non-fragmented packets while maintaining high performance for fragmented packets.

The application operates on a multi-core architecture, uses Receive Side Scaling (RSS) to distribute traffic, and supports configurable modes for flexible port configurations.

Key Features:

- IP Reassembly:
  - Functionality: The application assembles fragmented packets received on input ports based on their fragmentation headers.
  - Workflow: Upon successful reassembly, the complete packets are forwarded to their destination port.
- IP Fragmentation:
  - Functionality: Packets exceeding a configurable Maximum Transmission Unit (MTU) are fragmented into smaller packets.
  - Workflow: Fragments are generated with correct headers and forwarded while maintaining efficient resource utilization.
- Transparent Forwarding: Packets that are neither fragmented nor require reassembly are forwarded directly without additional processing overhead.
- Inner and Outer Fragmentation Handling: The application supports handling fragmentation at both inner (e.g., encapsulated traffic like GRE, VXLAN) and outer IP layers.
- Performance Optimization:
  - Designed for high throughput using multi-core processing.
  - Utilizes RSS to distribute traffic across multiple cores, ensuring efficient CPU utilization and scalability.
- Debuggability with Counters.

- Dual Operating Modes:
    - Dual Port Mode (Bidirectional): Traffic flows bidirectionally between two ports A ↔ B.
    - Quad Port Mode (Multiport): Independent unidirectional forwarding from Port A ↔ Port B and Port C ↔ Port D. In this mode, the direction of the traffic is isolated to go through two ports each time.
    - In both modes, non-fragmented or valid-sized packets follow the same path without additional actions.

6.12.2 System Design

The IP Fragmentation application can either run as a SW on BlueField-3's cores serving as an underlying service for host applications or on the host.

6.12.3 Application Architecture

6.12.3.1 Operational Workflow

- Packet Reception and Classification:
    - Traffic is received on the input ports, with RSS distributing flows to the cores the applications runs on.
    - Packets are classified into three categories:
        - Fragmented (Needs Reassembly)
        - Too Large (Needs Fragmentation)
        - Standard Packets (Direct Forwarding)
- Reassembly:
    - Fragments are buffered and reassembled using a configurable timeout.
    - Once reassembled, the full packet is validated and forwarded to the dedicated port (depending on the mode).
- Fragmentation:
    - Large packets exceeding the MTU are fragmented.
    - Fragments are prepared with correct headers, sequence numbers, and size.
- Direct Forwarding:
    - Standard packets are forwarded with minimal processing

6.12.3.2 Performance and Scalability

- Multi-Core Processing: The application scales horizontally with the number of CPU cores, with each core handling a subset of traffic flows.
- RSS Traffic Distribution: Receive Side Scaling ensures optimal load balancing across cores.
- Minimal Overhead: Processing logic is optimized for low-latency handling of standard packets while ensuring efficient fragmentation and reassembly operations.

6.12.3.3 Debugging and Monitoring

Application provides real-time counters for insights for:

- Packets processed.
- Fragments reassembled or fragmented.

- Errors such as timeout on incomplete fragments.

6.12.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Flow

For additional information about the used DOCA libraries, please refer to the respective programming guides.

6.12.5 Dependencies

- NVIDIA BlueField-3 DPU is required.
- Ubuntu 22.04 host (x86)

6.12.6 Compiling the Application

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for the application is located in the following directory:

```
/opt/mellanox/doca/applications/ip_frag/
```

6.12.6.1 Compiling All Applications

All DOCA applications are defined under a single Meson project. By default, the build process compiles all of them.

To build all applications:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

6.12.6.2 Compiling Only the Current Application

To build the IP fragmentation application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_ip_frag=true
ninja -C /tmp/build
```

doca_ip_frag is created under **/tmp/build/ip_frag/**.

Alternatively, one can set the desired flags in the **meson_options.txt** file instead of providing them in the compilation command line:

1. Edit the following flags in **/opt/mellanox/doca/applications/meson_options.txt**:
    a. Set **enable_all_applications** to **false**
    b. Set **enable_ip_frag** to **true**
2. Execute:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_ip_frag is created under **/tmp/build/ip_frag/**.

6.12.7 Running the Application

6.12.7.1 Prerequisites

- The Fragmentation application is based on DOCA Flow. Therefore, the user is required to allocate huge pages.

```
$ echo '4096' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
$ sudo mkdir /mnt/huge
$ sudo mount -t hugetlbfs -o pagesize=2M nodev /mnt/huge
```

- The Flex Parser Profile is a setting that enables flexible protocol parsing on NVIDIA NICs/DPUs. To enable GTP protocol support, set the Flex Parser Profile to 3. This configuration is mandatory and must be done manually in the system:

```
$ sudo mlxconfig -d <pcie_address> s FLEX_PARSER_PROFILE_ENABLE=3
```

6.12.7.2 Application Execution

The IP Fragmentation application is provided in source form, hence a compilation is required before the application can be executed.

- Application usage instructions:

```
Usage: doca_ip_frag [DPDK Flags] -- [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -a, --device                      device
  -r, --rep                         device representor
  -m, --mode                        Ip_frag application mode. Bidirectional mode forwards packets between a
single reassembly port and a single fragmentation port (two ports in total). Multiport mode forwards
packets between two pairs of reassembly and fragmentation ports (four ports in total). For more information
consult DOCA IP Fragmentation Application Guide. Format: bidir, multiport
  -u, --mtu                         MTU size
  -t, --frag-aging-timeout          Aging timeout of fragments pending packet reassembly in the
fragmentation table (in ms)
  -s, --frag-tbl-size               Frag table size, i.e. maximum amount of concurrent defragmentation
contexts per worker thread
```

```
  -c, --mbuf-chain                Enable mbuf chaining (required for IPv6 fragmentation support)
  -w, --cksum-accel-disable       Disable hardware-accelerated checksum calculation
```

For additional information, please refer to the Command Line Flags section below.
The above usage printout can be printed to the command line using the **-h** (or **--help**) options:

```
/tmp/build/ip_frag/doca_ip_frag -- -h
```

- CLI example for running the application on BlueField-3:

```
/tmp/build/ip_frag/doca_ip_frag -l 3-15 -- -a aux/2,dv_flow_en=2,sft_en=1 -a aux/3,dv_flow_en=2,sft_en=1 -a
aux/4,dv_flow_en=2,sft_en=1 -a aux/5,dv_flow_en=2,sft_en=1 -l 50 -m multiport
```

- CLI example for running the application on the host:

```
/tmp/build/ip_frag/doca_ip_frag -l 0-7 -- -a pci/0000:08:00.0,dv_flow_en=2 -a pci/0000:08:00.1,dv_flow_en=2
 -l 60 -m bidir -t 1000
```

For additional information regarding the command line syntax of the device and device representor identifiers, please referr to the matching section in the DOCA Arg Parser Guide.

6.12.7.3 Command Line Flags

6.12.7.3.1 General Flags

Short Flag	Long Flag	Description
-h	**--help**	Prints a help synopsis and exits
-v	**--version**	Prints program version information and exits
-l	**--log-level**	Sets the numeric log level for the application: • **10** – DISABLE • **20** – CRITICAL • **30** – ERROR • **40** – WARNING • **50** – INFO • **60** – DEBUG • **70** – TRACE (requires compilation with TRACE support)
N/A	**--sdk-log-level**	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	**--log-filter**	Filters logs from specific modules (comma-separated list)
-j	**--json**	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.12.7.3.2 Program Flags

Short Flag	Long Flag	Description
a	device	Device identifier
r	rep	Device representor identifier
m	mode	Execution mode: bidir, multiport This is a mandatory flag.
u	mtu	MTU for fragmentation
t	frag-aging-timeout	Fragmentation table aging timeout (in milliseconds)
s	frag-tbl-size	Fragmentation table size
c	mbuf-chain	Enable mbuf chaining support on packet reassembly

6.12.7.4 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

6.12.8 References

- `/opt/mellanox/doca/applications/ip_frag/`

6.13 DOCA IPsec Security Gateway Application Guide

This document provides an IPsec security gateway implementation on top of NVIDIA® BlueField® DPU.

> **Important note for NVIDIA® BlueField®-2 DPUs**
>
> If your target application utilizes 100Gb/s or higher bandwidth, where a substantial part of the bandwidth is allocated for IPsec traffic, please refer to the *NVIDIA BlueField-2 DPUs Product Release Notes* to learn about a potential bandwidth limitation. To access the relevant product release notes, please contact your NVIDIA sales representative.

6.13.1 Introduction

DOCA IPsec Security Gateway leverages the DPU's hardware capability for secure network communication. The application demonstrates how to insert rules related to IPsec encryption and decryption based on the DOCA Flow library.

The application demonstrates how to insert rules to create an IPsec tunnel.

An example for configuring the Internet Key Exchange (IKE) can be found under section "Keying Daemon Integration (StrongSwan)" but is not considered part of the application.

The application can be configured to receive IPsec rules in one of the following ways:

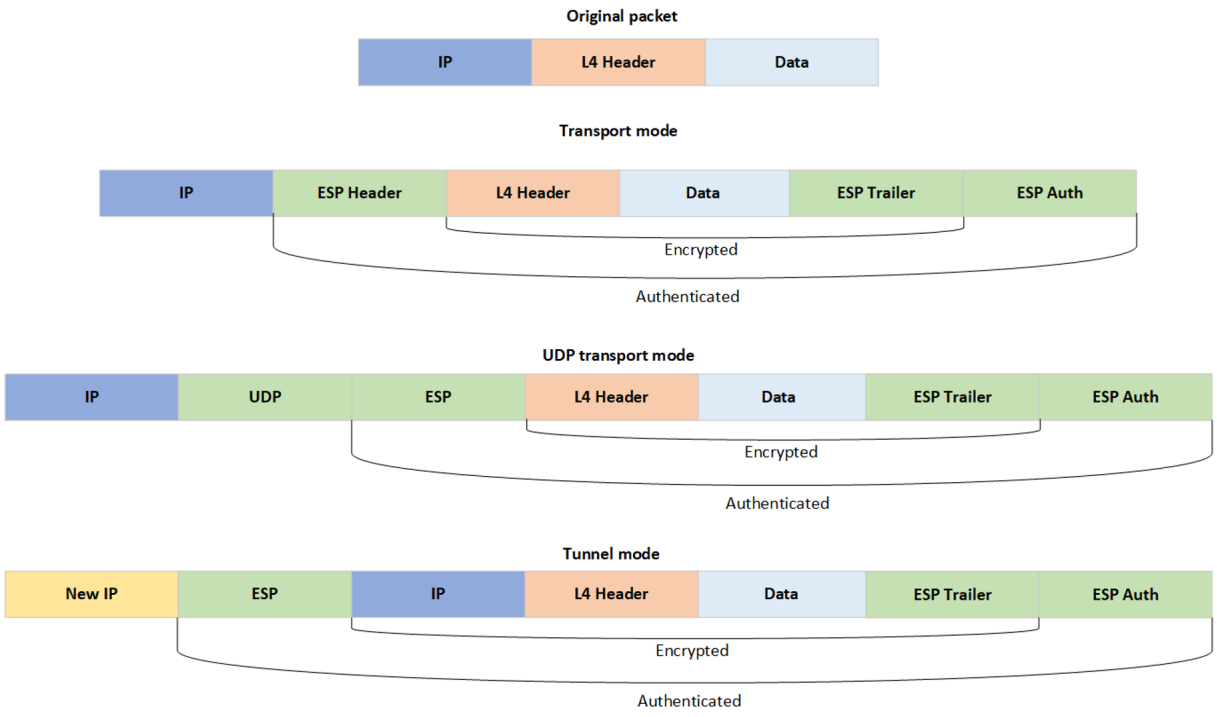
- Static configuration – (default) receives a fixed list of rules for IPsec encryption and decryption

  When creating the security association (SA) object, the application gets the key, salt, and other SA attributes from the JSON input file.

- Dynamic configuration – receives IPsec encryption and decryption rules during runtime through a Unix domain socket (UDS) which is enabled when providing a socket path to the application

  You may find an example of integrating a rules generator with the application under strongSwan project (DOCA plugin).

The application supports the following IPsec modes: Tunnel, transport, UDP transport.

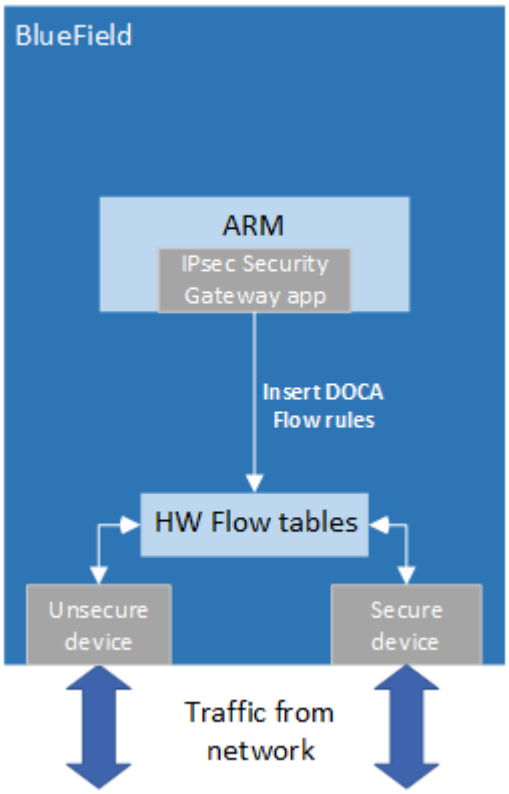


6.13.2 System Design

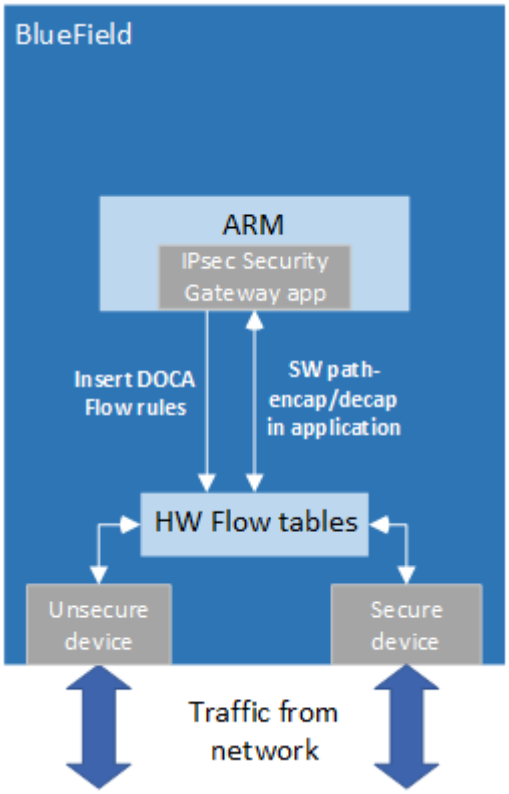
DOCA IPsec Security Gateway is designed to run with 2 ports, secured and unsecured:

- Secured port – BlueField receives IPsec encrypted packets and, after decryption, they are sent through the unsecured port
- Unsecured port – BlueField receives regular (plain text) packets and, after encryption, they are sent through the secured port

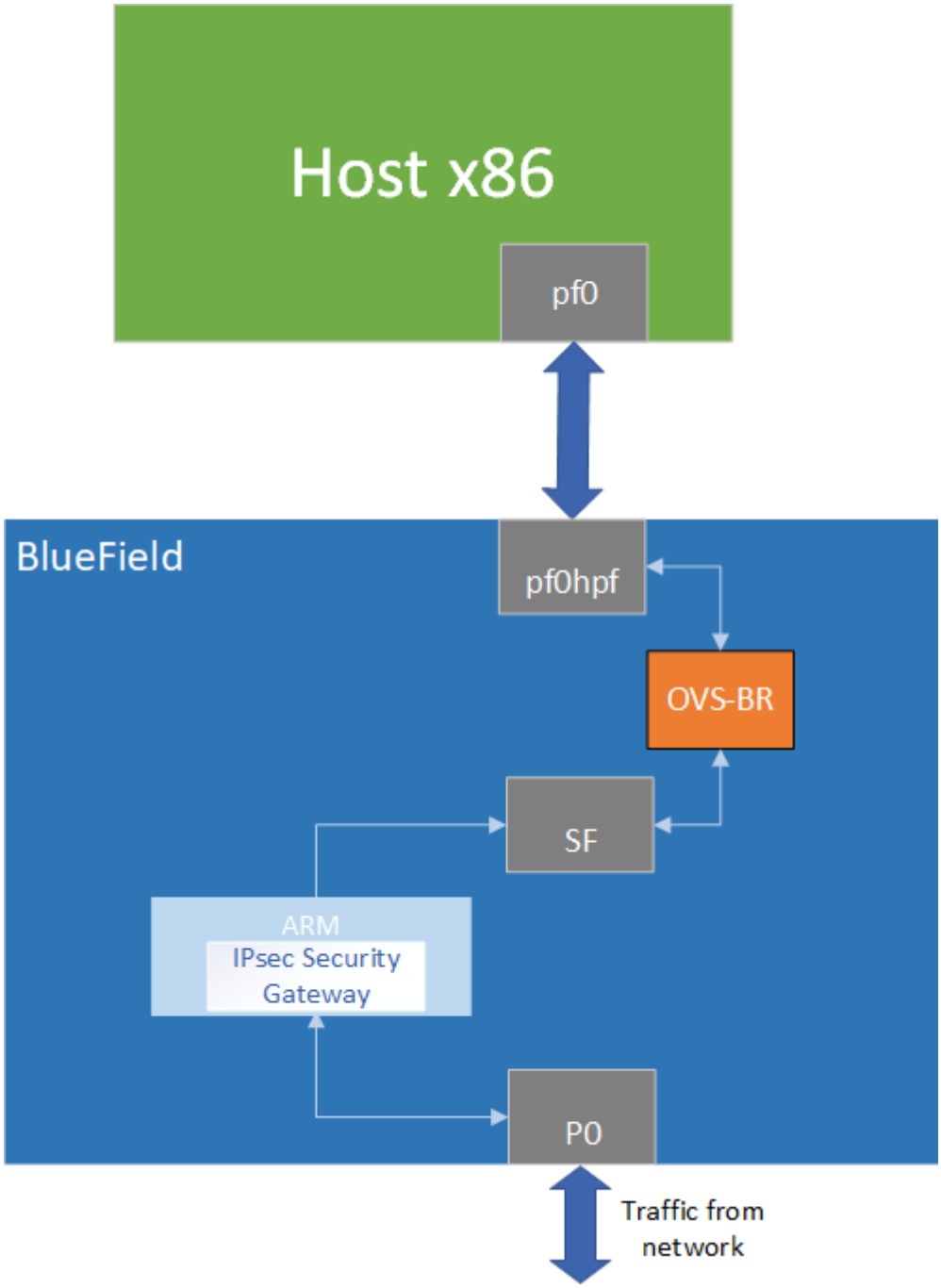
Example packet path for hardware (HW) offloading:



Example packet path for partial software processing (handling encap/decap in software):



Using the application with SF:



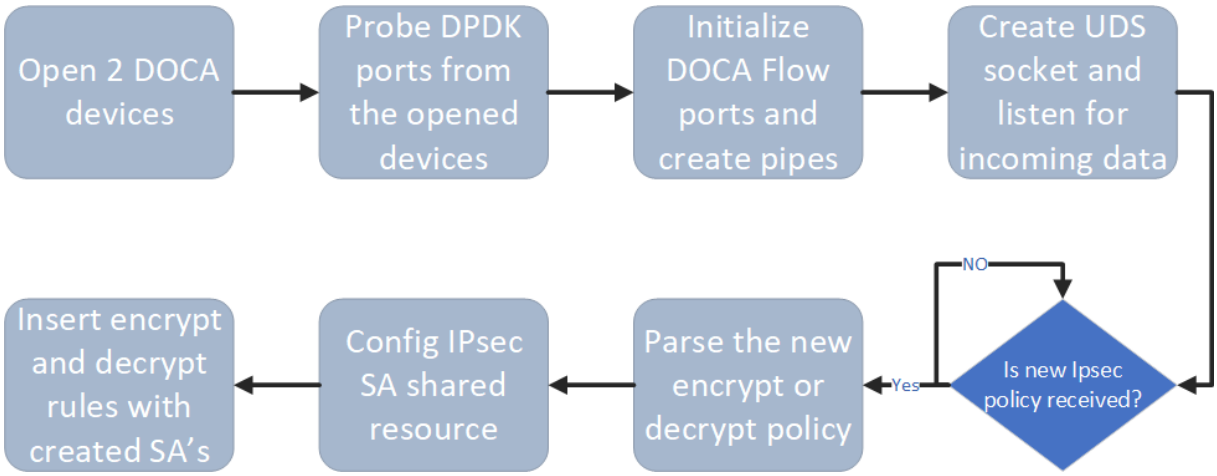
6.13.3 Application Architecture

6.13.3.1 Static Configuration

1. Open two DOCA devices, one for the secured port and another for the unsecured port.
2. With the open DOCA devices, the application probes DPDK ports and initializes DOCA Flow and DOCA Flow ports accordingly.

3. On the created ports, build DOCA Flow pipes.
4. In a loop according to the JSON rules:
    a. Create IPSec SA shared resource for the new rule.
    b. Insert encrypt or decrypt rule to DOCA Flow pipes.

6.13.3.2 Dynamic Configuration



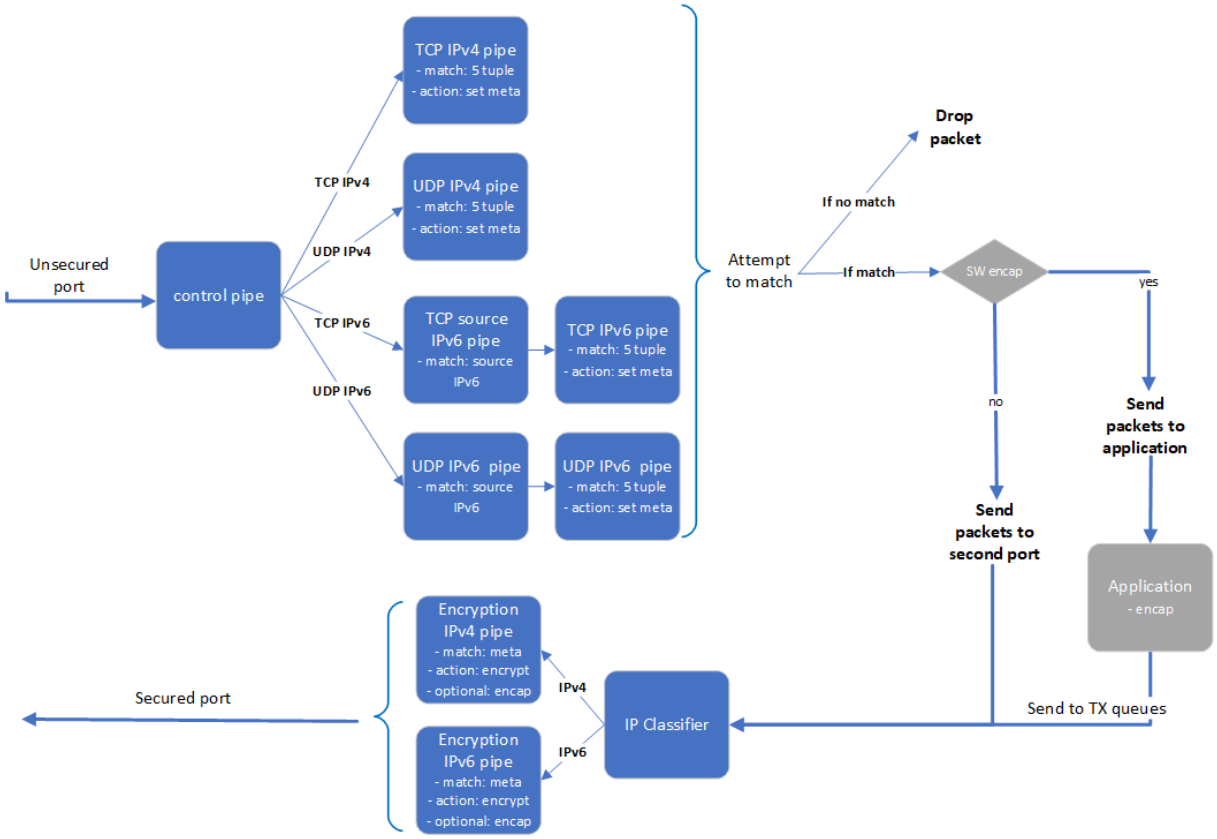
1. Open two DOCA devices, one for the secured port and another for the unsecured port.
2. With the open DOCA devices, the application probes DPDK ports and initializes DOCA Flow and DOCA Flow ports accordingly.
3. On the created ports, build DOCA Flow pipes.
4. Create UDS socket and listen for incoming data.
5. While waiting for new IPsec policies to be received in a loop, if a new IPsec policy is received:
    a. Parse the policy whether it is an encryption or decryption rule.
    b. Create IPSec SA shared resource for the new rule.
    c. Insert encrypt or decrypt rule to DOCA Flow pipes.

6.13.3.3 DOCA Flow Modes

The application can run in two modes, `vnf` and `switch`. For more information about the modes, please refer to "Pipe Mode" in the DOCA Flow.

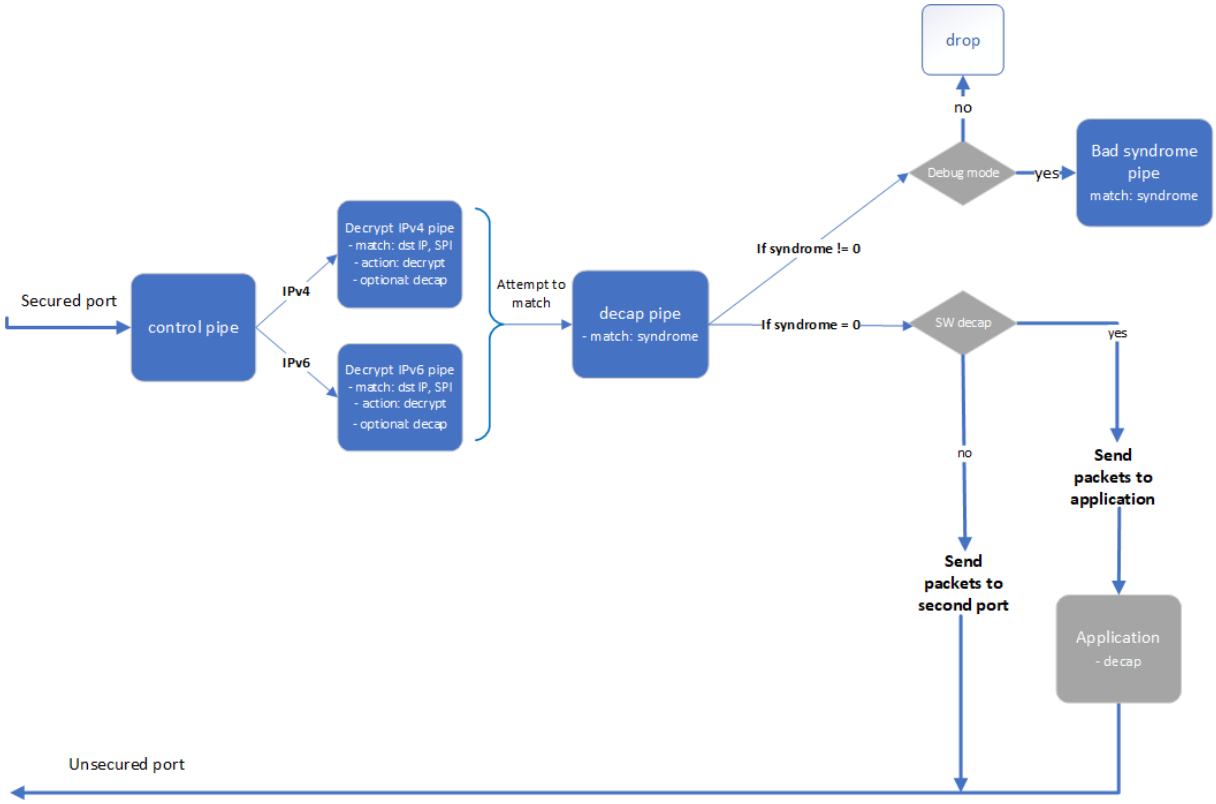
6.13.3.3.1 VNF Mode

6.13.3.3.1.1 Encryption



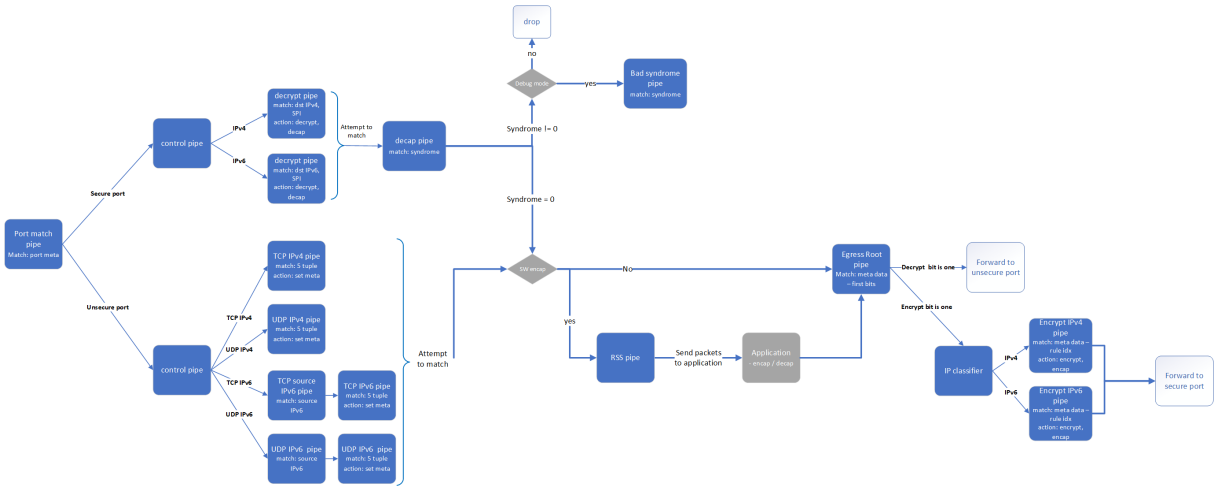
1. The application builds pipes for encryption. Control pipe as root with four entries that match L3 and L4 types and forward the traffic to the relevant pipes.
    a. IPv6 pipes – match the source IP address and forward the traffic to a pipe that matches 5-tuple excluding the source IP.
    b. In the 5-tuple match pipes set action of "set meta data", the metadata would be the rule's index in the JSON file.
    c. The matched packet is forwarded to the second port.
2. In the secured egress domain, the IP classifier pipe sends the packets to the correct encryption pipe (IPv4 or IPv6) which has a shared IPsec encrypt action. According to the metadata match, the packet is encrypted with the encap destination IP and SPI as defined in the user's rules.

6.13.3.3.1.2 Decryption



1. The application builds pipes for decryption. Control pipe as root with two entries that match L3 type and forward the traffic to the relevant decrypt pipe.
2. The decrypt pipe matches the destination IP and SPI according to the rule files and has a shared IPsec action for decryption.
3. After decryption, the matched packets are forwarded to the decap pipe and, if the syndrome is non-zero, the packets are dropped. Otherwise, the packets decap the ESP header and forward to the second port.
    a. In debug mode, if syndrome is non-zero, then it sends to bad syndrome pipe to match on the syndrome, count and drop/send to application.

6.13.3.3.2 Switch Mode



In switch mode, an ingress root pipe matches the source port to decide what the next pipe is:

- Based on the port, the packet passes through almost the same path as VNF mode and the metadata is set. Afterwards, the packet moves to egress root pipe.

In egress root pipe, the match is on encrypt and decrypt bits that were set in the packet meta:

- Decrypt bit is 1 – packet finishes the decrypt path and must be sent to the unsecure port
- Encrypt bit is 1 – packet almost finishes the encrypt path and must be sent to the encrypt pipe on the secure egress domain and to the secure port from there

Additional information about the application's pipeline can be found in the detailed layered view as shown by the DOCA Flow Tune tool.

6.13.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Flow

Refer to their respective programming guide for more information.

6.13.5 Compiling the Application

> Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for the application is located in the following directory:

```
/opt/mellanox/doca/applications/ipsec_security_gw/
```

6.13.5.1 Prerequisites

The application relies on the **json-c** open source, hence, requires the following installation:

- Ubuntu/Debian:

```
$ sudo apt install libjson-c-dev
```

- CentOS/RHEL:

```
$ sudo yum install json-c-devel
```

6.13.5.2 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_ipsec_security_gw is created under **/tmp/build/ipsec_security_gw/**.

6.13.5.3 Compiling Only the Current Application

To directly build only the IPsec Security Gateway application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_ipsec_security_gw=true
ninja -C /tmp/build
```

doca_ipsec_security_gw is created under **/tmp/build/ipsec_security_gw/**.

Alternatively, users can set the desired flags in the **meson_options.txt** file instead of providing them in the compilation command line:

1. Edit the following flags in **/opt/mellanox/doca/applications/meson_options.txt**:
    - Set **enable_all_applications** to **false**

- Set **enable_ipsec_security_gw** to **true**

2. Run the following compilation commands:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

> **doca_ipsec_security_gw** is created under **/tmp/build/ipsec_security_gw/**.

6.13.6 Running the Application

6.13.6.1 Prerequisites

1. The IPsec security gateway application is based on DOCA Flow. Therefore, the user is required to allocate huge pages.

```
$ echo '2048' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
$ sudo mkdir /mnt/huge
$ sudo mount -t hugetlbfs -o pagesize=2M nodev /mnt/huge
```

2. VNF mode – the IPsec security gateway application requires disabling some of the hardware tables:

```
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode legacy
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode legacy

echo none > /sys/class/net/p0/compat/devlink/encap
echo none > /sys/class/net/p1/compat/devlink/encap

/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode switchdev
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode switchdev
```

   To restore the old configuration:

```
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode legacy
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode legacy

echo basic > /sys/class/net/p0/compat/devlink/encap
echo basic > /sys/class/net/p1/compat/devlink/encap

/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode switchdev
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode switchdev
```

3. Switch mode – the IPsec security gateway application requires configuring the ports to run in switch mode:

```
sudo mlxconfig -d /dev/mst/mt41686(mt41692)_pciconf0 s LAG_RESOURCE_ALLOCATION=1
# power cycle the host to apply this setting

/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode legacy
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode legacy

sudo devlink dev param set pci/0000:03:00.0 name esw_pet_insert value false cmode runtime
sudo devlink dev param set pci/0000:03:00.1 name esw_pet_insert value false cmode runtime

/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode switchdev
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode switchdev

sudo devlink dev param set pci/0000:03:00.0 name esw_multiport value true cmode runtime
sudo devlink dev param set pci/0000:03:00.1 name esw_multiport value true cmode runtime
```

> Make sure to perform BlueField system reboot prior to power cycling the host.

To restore the old configuration:

```
sudo devlink dev param set pci/0000:03:00.0 name esw_multiport value false cmode runtime
sudo devlink dev param set pci/0000:03:00.1 name esw_multiport value false cmode runtime
```

6.13.6.2 Application Execution

The IPsec Security Gateway application is provided in source form. Therefore, a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_ipsec_security_gw [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -s, --secured                     secured port device identifier
  -u, --unsecured                   unsecured port device identifier
  -c, --config                      Path to the JSON file with application configuration
  -m, --mode                        ipsec mode - {tunnel/transport/udp_transport}
  -i, --ipc                         IPC socket file path
  -ur, --unsecured-rep              unsecured port representor device identifier
  -n, --nb-cores                    number of cores
  --debug                           Enable debug counters
```

> This usage printout can be printed to the command line using the **-h** (or **--help**) options:
>
> ```
> ./doca_ipsec_security_gw -h
> ```

> For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the BlueField or host:
   - Static Configuration:

     ```
     ./doca_ipsec_security_gw -s pci/03:00.0 -u pci/03:00.1 -c ./ipsec_security_gw_config.json -m
     transport
     ```

     > Both the PCIe address identifiers (**-s** and **-u** flags) should match the addresses of the desired PCIe devices.

   - Dynamic Configuration:

     ```
     ./doca_ipsec_security_gw -s pci/03:00.0 -u pci/03:00.1 -c ./ipsec_security_gw_config.json -m
     transport -i /tmp/rules_socket
     ```

Both the PCIe address identifiers (`-s` and `-u` flags) should match the addresses of the desired PCIe devices.

For additional information regarding the command line syntax of the device and device representor identifiers, please referr to the matching section in the DOCA Arg Parser Guide.

6.13.6.3 Command Line Flags

6.13.6.3.1 General Flags

Short Flag	Long Flag	Description
`-h`	`--help`	Prints a help synopsis and exits
`-v`	`--version`	Prints program version information and exits
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.13.6.3.2 Program Flags

Short Flag	Long Flag	Description
`c`	`config`	Path to JSON file with configurations
`s`	`secured`	Device identifier for the secured port
`u`	`unsecured`	Device identifier for the unsecured port

Short Flag	Long Flag	Description
`ur`	`unsecured-rep`	Device representor identifier for the unsecured port (switch mode)
`m`	`mode`	IPsec mode. Possible values: `tunnel` , `transport` , `udp_transport`
`i`	`ipc`	IPC socket file path for receiving IPsec rules during runtime
`n`	`nb-cores`	Number of cores
N/A	`debug`	Add counters to all the entries

6.13.6.4 Static Configuration IPsec Rules

IPsec rules and other configuration can be added with a JSON config file which is passed using the `--config` parameter.

Section	Field	Type	Description	Example
`config`	`switch`	bool	Configures whether DOCA Flow runs in VNF (`false`) or switch (`true`) mode	`"switch": true`
	`esp-header-offload`	string	Decap and encap offloading: `both` , `encap` , `decap` , or `none` . Default is `both` (offloading both encap and decap).	`"esp-header-offload": "none"`
	`sw-sn-inc-enable`	bool	Increments sequence number of ESP in software if set to `true` . Default is false. ⚠ Available only if `esp_header_offload` is `decap` or `none` .	`"sw-sn-inc-enable": true`
	`sw-antireplay-enable`	bool	Enables anti-replay mechanism in software if set to `true` . Default is false. ⚠ Available only if `esp_header_offload` is `encap` or `none` . ⚠ Window size is 64. Not ESN. Supports non-zero `sn_initial` .	`"sw-antireplay-enable": true`
	`sn-initial`	uint	Initial sequence number for ESP header. Used also when `sw_antireplay_enable` is true. Default is 0.	`"sn-initial": 0`
	`debug`	bool	Set debug counter for all entries when `true` . Default is `false` . This parameter is also used from CLI, will be taken as true if was sent in one of them.	`"debug": false`

Section	Field	Type	Description	Example
	`fwd-bad-syndrome`	string	Forward packets that has bad syndrome: `drop`, `RSS`. Default is `drop`. ⚠ Only available in debug mode.	`"fwd-bad-syndrome": "drop"`
	`perf-measurements`	string	Possible values: `none`, `insertion-rate`, `bandwidth`, `both`. Default is `none`. • `insertion-rate` – print the total time it took to add the entries • `bandwidth` – optimize the pipe to improve pps for IPv6	`"perf-measurements": "both"`
	`vxlan-encap`	bool	When `true`, preform `vxlan-encap` after encryption and decap before decryption. Default is `false`.	`"vxlan-encap": false`
	`vni`	uint	When `vxlan-encap` is true, use this `vni` value in the VXLAN tunnel.	`"vni": 5`
	`marker-encap`	bool	When `true`, add an extra non-ESP marker of 8 bytes. Default is `false`.	`"marker-encap": false`
	`icv-length`	int	ICV length value: `8`, `12`, or `16`. Default is `16`.	`"icv-length": 12`
`encrypt_rules`	`ip-version`	int	Source and destination IP version. Possible values: `4`, `6`. Optional; default is `4`.	`"ip-version": 6`
	`src-ip`	string	Source IP to match	`"src-ip": "1.2.3.4"`
	`dst-ip`	string	Destination IP to match	`"dst-ip": "101:101:101:101:101:101:101:101"`
	`protocol`	string	L4 protocol: TCP or UDP	`"protocol"`
	`src-port`	int	Source port to match	
	`dst-port`	int	Destination port to match	`"dst-port": 55`
	`encap-ip-version`	int	Encap IP version: `4` or `6`. Optional; default is `4`.	`"ip-version": 4`

Section	Field	Type	Description	Example
	`encap-dst-ip`	string	Encap destination IP ⚠ Mandatory for tunnel mode only.	`"encap-dst-ip": "1.1.1.1"`
	`spi`	int	SPI integer to set in the ESP header	`"spi": 5`
	`key`	string	Key for creating the SA (in hex format)	`"key": "112233445566778899aabbccdd"`
	`key-type`	int	Key size: `128` or `256`. Optional; default is `256`.	`"key-type": 128`
	`iv`	string	Initial vector (IV) for creating the SA (in hex format). Optional; default is an IV of zero bytes.	`"iv": "0102030405060708"`
	`salt`	int	Salt value for creating the SA. Default ICS `6`.	`"salt": 1212`
	`lifetime-threshold`	int	Set IPsec lifetime threshold. Ignored if `sw-sn-inc-enable` is true. Default is 0.	`"lifetime-threshold": 1000000`
	`esn_en`	bool	Enables extended sequence number. Default is `false`.	`"esn_en" : true`
`decrypt_rules`	`ip-version`	int	Destination IP version: `4` or `6`. Optional; default is `4`.	`"ip-version": 6`
	`dst-ip`	string	Destination IP to match	`"dst-ip": "1122:3344:5566:7788:99aa:bbcc:ddee:ff00"`
	`inner-ip-version`	int	Inner IP version: `4` or `6`. Optional; default is `4`. ⚠ Mandatory for tunnel mode only.	`"inner-ip-version": 4`
	`spi`	int	SPI to match in the ESP header	`"spi": 5`
	`key`	string	Key for creating the SA (in hex format)	`"key": "112233445566778899aabbccdd"`

Section	Field	Type	Description	Example
	`key-type`	int	Key size: `128` or `256`. Optional; default is `256`.	`"key-type": 128`
	`iv`	string	Initial vector (IV) for creating the SA (in hex format). Optional; default is an IV of zero bytes.	`"iv": "0102030405060708"`
	`salt`	int	Salt value for creating the SA. Default is `6`.	`"salt": 1212`
	`lifetime-threshold`	int	Set IPsec lifetime threshold. Ignored if `sw-antireplay-enable` is true. Default is 0.	`"lifetime-threshold": 1000000`
	`esn_en`	bool	Enables extended sequence number. Default is `false`.	`"esn_en" : true`

6.13.6.5 Dynamic Configuration IPsec Rules

The application listens on the UDS socket for receiving a predefined structure for the IPsec policy defined in the `policy.h` file.

The client program or keying daemon should connect to the socket with the same socket file path provided to the application by the `--ipc` / `-i` flags, and send the policy structure as packed to the application through the same socket.

> ⚠ In the dynamic configuration, the application uses the `config` section from the JSON config file and ignores the `encrypt_rules` and `decrypt_rules` sections.

The IPsec policy structure:

```
struct ipsec_security_gw_ipsec_policy {
    /* Protocols attributes */
    uint16_t src_port;                  /* Policy inner source port */
    uint16_t dst_port;                  /* Policy inner destination port */
    uint8_t l3_protocol;                /* Policy L3 proto {POLICY_L3_TYPE_IPV4, POLICY_L3_TYPE_IPV6} */
    uint8_t l4_protocol;                /* Policy L4 proto {POLICY_L4_TYPE_UDP, POLICY_L4_TYPE_TCP} */
    uint8_t outer_l3_protocol;          /* Policy outer L3 type {POLICY_L3_TYPE_IPV4, POLICY_L3_TYPE_IPV6} */

    /* Policy attributes */
    uint8_t policy_direction;           /* Policy direction {POLICY_DIR_IN, POLICY_DIR_OUT} */
    uint8_t policy_mode;                /* Policy IPSEC mode {POLICY_MODE_TRANSPORT, POLICY_MODE_TUNNEL} */

    /* Security Association attributes */
    uint8_t esn;                        /* Is ESN enabled? */
    uint8_t icv_length;                 /* ICV length in bytes {8, 12, 16} */
    uint8_t key_type;                   /* AES key type {POLICY_KEY_TYPE_128, POLICY_KEY_TYPE_256} */
    uint32_t spi;                       /* Security Parameter Index */
    uint32_t salt;                      /* Cryptographic salt */
    uint8_t enc_key_data[MAX_KEY_LEN];  /* Encryption key (binary) */

    /* Policy inner and outer addresses */
    char src_ip_addr[MAX_IP_ADDR_LEN + 1];      /* Policy inner IP source address in string format */
    char dst_ip_addr[MAX_IP_ADDR_LEN + 1];      /* Policy inner IP destination address in string format */
    char outer_src_ip[MAX_IP_ADDR_LEN + 1];     /* Policy outer IP source address in string format */
    char outer_dst_ip[MAX_IP_ADDR_LEN + 1];     /* Policy outer IP destination address in string format */
};
```

The policy type, whether it is encrypted or decrypted, is classified according to the **policy_direction** attribute:

- **POLICY_DIR_IN** – decryption policy
- **POLICY_DIR_OUT** – encryption policy

6.13.6.6 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.13.7 Application Code Flow

1. Parse application argument.
    a. Initialize arg parser resources and register DOCA general parameters.

    ```
    doca_argp_init();
    ```

    b. Register the application's parameters.

    ```
    register_ipsec_security_gw_params();
    ```

    c. Parse the arguments.

    ```
    doca_argp_start();
    ```

        i. Parse app parameters.
2. DPDK initialization.

    ```
    rte_eal_init();
    ```

    Call **rte_eal_init()** to initialize EAL resources with the provided EAL flags for not probing the ports.
3. Parse config file.

    ```
    ipsec_security_gw_parse_config();
    ```

4. Initialize devices and ports.

    ```
    ipsec_security_gw_init_devices();
    ```

    a. Open DOCA devices with input PCIe addresses / interface names.
    b. Probe DPDK port from each opened device.
5. Initialize and start DPDK ports.

    ```
    dpdk_queues_and_ports_init();
    ```

    a. Initialize DPDK ports, including mempool allocation.
    b. Initialize hairpin queues if needed.
    c. Binds hairpin queues of each port to its peer port.

6. Initialize DOCA Flow.

```
ipsec_security_gw_init_doca_flow();
```

   a. Initialize DOCA Flow library.
   b. Find the indices of the DPDK-probed ports and start DOCA Flow ports with them.
7. Insert rules.
   a. Insert encryption rules.

```
ipsec_security_gw_insert_encrypt_rules();
```

   b. Insert decryption rules.

```
ipsec_security_gw_insert_decrypt_rules();
```

8. Wait for traffic.

```
ipsec_security_gw_wait_for_traffic();
```

   a. wait in a loop until the user terminates the program.
9. IPsec security gateway cleanup:
   a. DOCA Flow cleanup; destroy initialized ports.

```
doca_flow_cleanup();
```

   b. SA destruction.

```
ipsec_security_gw_destroy_sas();
```

   c. IPsec objects destruction.

```
ipsec_security_gw_ipsec_ctx_destroy();
```

   d. Destroy DPDK ports and queues.

```
dpdk_queues_and_ports_fini();
```

   e. DPDK finish.

```
dpdk_fini();
```

   Calls **rte_eal_destroy()** to destroy initialized EAL resources.
   f. Arg parser destroy.

```
doca_argp_destroy()
```

6.13.8 Keying Daemon Integration (StrongSwan)

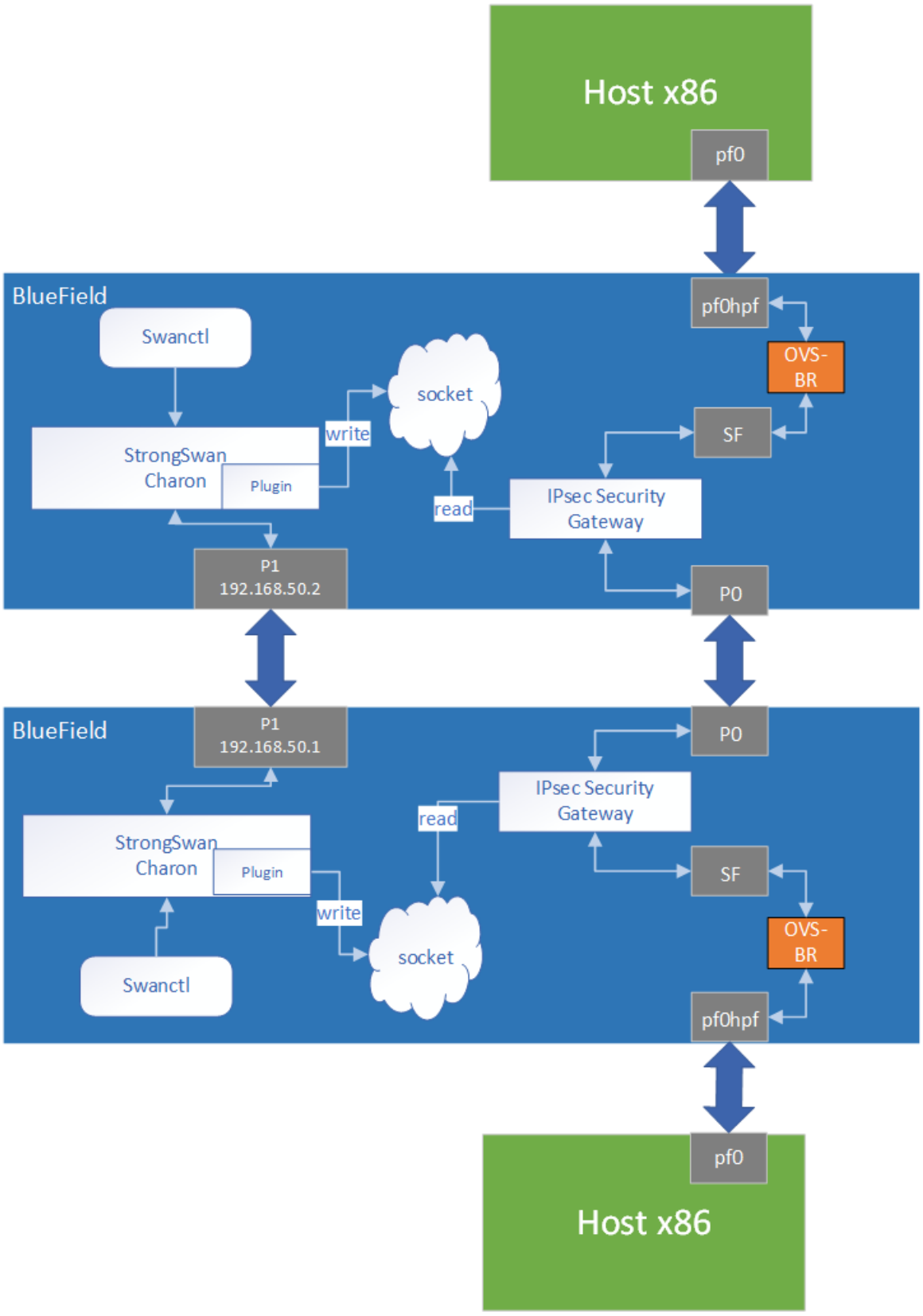
strongSwan is a keying daemon that uses the Internet Key Exchange Version 2 (IKEv2) protocol to establish SAs between two peers. strongSwan includes a DOCA plugin that is part of the strongSwan package in BFB. The plugin is loaded only if the DOCA IPsec Security Gateway is

triggered. The plugin connects to UDS socket and sends IPsec policies to the application after the key exchange completes.

For more information about the key daemon, refer to strongSwan documentation.

6.13.8.1 End-to-end Architecture

The following diagram presents an architecture where two BlueField DPUs are connected to each other with DOCA IPsec Security Gateway running on each.


Host x86
pf0
BlueField
pf0hpf
Swanctl
OVS-
BR
socket
SF
write
StrongSwan
Charon
Plugin
read
IPsec Security
Gateway
P1
192.168.50.2
P0
BlueField
P1
192.168.50.1
P0
IPsec Security
Gateway
read
StrongSwan
Charon
Plugin
SF
write
OVS-
BR
socket
Swanctl
pf0hpf
pf0
Host x86

swanctl is a command line tool used for strongSwan IPsec configuration:

1. Run DOCA IPsec Security Gateway on both sides in a dynamic configuration.
2. Start strongSwan service.
3. Configure strongSwan IPsec using the **swanctl.conf** configuration file on both sides.
4. Start key exchange between the two peers. At the end of the flow, the result arrives to the DOCA plugin, populates the policy-defined structure, and sends it to the socket.
5. DOCA IPsec Security Gateway on both sides reads new policies from the socket, performs the parsing, creates a DOCA SA object, and adds flow decrypt/encrypt entry.

This architecture uses P1 uplink on both BlueField DPUs to run the strongSwan key daemon. To configure the uplink:

1. Configure an IP addresses for the PFs of both DPUs:
   a. On BF1:

   ```
   ip addr add 192.168.50.1/24 dev p1
   ```

   b. On BF2:

   ```
   ip addr add 192.168.50.2/24 dev p1
   ```

   It is possible to configure multiple IP addresses to uplinks to run key exchanges with different policy attributes.

2. Verify the connection between two BlueField DPUs.

   ```
   BF1> ping 192.168.50.2
   ```

   Make sure that the uplink is not in OVS bridges.

3. Configure the **swanctl.conf** files for each machine. The file should be located under **/etc/swanctl/conf.d/** .

   Adding **swanctl.conf** file examples:

   - Transport mode
     - **swanctl.conf** example for BF1:

     ```
     connections {
        BF1-BF2 {
           local_addrs  = 192.168.50.1
           remote_addrs = 192.168.50.2
           rekey_time = 0

           local {
             auth = psk
             id = host1
           }
           remote {
             auth = psk
             id = host2
           }

           children {
              bf {
                 local_ts = 192.168.50.1/32 [udp/60]
                 remote_ts = 192.168.50.2/32 [udp/90]
                 esp_proposals = aes128gcm128-x25519-esn
                 mode = transport
                 policies_fwd_out = yes
                 life_time = 0
              }
     ```

```
        }
        version = 2
        mobike = no
        reauth_time = 0
        proposals = aes128-sha256-x25519
    }
}

secrets {
    ike-BF {
        id-host1 = host1
        id-host2 = host2
        secret = 0sv+NkxY9LLZvwj4qCC2o/gGrWDF2d21jL
    }
}
```

- **swanctl.conf** example for BF2:

```
connections {
    BF2-BF1 {
        local_addrs  = 192.168.50.2
        remote_addrs = 192.168.50.1
        rekey_time = 0

        local {
          auth = psk
          id = host2
        }
        remote {
          auth = psk
          id = host1
        }

        children {
            bf {
                local_ts = 192.168.50.2/32 [udp/90]
                remote_ts = 192.168.50.1/32 [udp/60]
                esp_proposals = aes128gcm128-x25519-esn
                mode = transport
                life_time = 0
            }
        }
        version = 2
        mobike = no
        reauth_time = 0
        proposals = aes128-sha256-x25519
    }
}

secrets {
    ike-BF {
        id-host1 = host1
        id-host2 = host2
        secret = 0sv+NkxY9LLZvwj4qCC2o/gGrWDF2d21jL
    }
}
```

- Tunnel mode

```
connections {
    BF1-BF2 {
        local_addrs  = 192.168.50.2
        remote_addrs = 192.168.50.1
        rekey_time = 0

        local {
          auth = psk
          id = host2
        }
        remote {
          auth = psk
          id = host1
        }

        children {
            bf {
                local_ts = 2001:db8:85a3::8a2e:370:7334/128 [udp/3030]
                remote_ts = 2001:db8:85a3::8a2e:370:7335/128 [udp/55]
                esp_proposals = aes128gcm128-x25519-esn
                life_time = 0
            }
        }
        version = 2
        mobike = no
        proposals = aes128-sha256-x25519
    }
}

secrets {
    ike-BF {
        id-host1 = host1
        id-host2 = host2
        secret = 0sv+NkxY9LLZvwj4qCC2o/gGrWDF2d21jL
    }
}
```

> ⚠ **local_ts** and **remote_ts** must have a netmask of /32 for IPv4 addresses and /128 for IPv6 addresses.

> ⚠ SA rekey is not supported in DOCA plugin. **connection.rekey_time** must be set to 0 and **connection.child.life_time** must be set to 0.

DOCA IPsec only supports ESP headers, AES-GCM encryption algorithm, and key sizes 128 or 256. Therefore, when setting ESP proposals in the **swanctl.conf**, please adhere to the values provided in the following table:

ESP Proposal	Algorithm Type Including ICV Length	Key Size
aes128gcm8	ENCR_AES_GCM_ICV8	128
aes128gcm64	ENCR_AES_GCM_ICV8	128
aes128gcm12	ENCR_AES_GCM_ICV12	128
aes128gcm96	ENCR_AES_GCM_ICV12	128
aes128gcm16	ENCR_AES_GCM_ICV16	128
aes128gcm128	ENCR_AES_GCM_ICV16	128
aes128gcm	ENCR_AES_GCM_ICV16	128
aes256gcm8	ENCR_AES_GCM_ICV8	256
aes256gcm64	ENCR_AES_GCM_ICV8	256
aes256gcm12	ENCR_AES_GCM_ICV12	256
aes256gcm96	ENCR_AES_GCM_ICV12	256
aes256gcm16	ENCR_AES_GCM_ICV16	256
aes256gcm128	ENCR_AES_GCM_ICV16	256
aes256gcm	ENCR_AES_GCM_ICV16	256

6.13.8.2 Building strongSwan

To enable the DOCA plugin in strongSwan:

1. Verify that the dependencies listed here are installed in your environment. **libgmp-dev** is missing from that list so make sure to install that as well.
2. Git clone https://github.com/Mellanox/strongswan.git.
3. Git checkout BF-5.9.10 branch.
4. Add your changes in the plugin located under **src/libcharon/plugins/doca**.
5. Run **autogen.sh** within the strongSwan repo.
6. Run the following:

```
./configure --enable-openssl --disable-random --prefix=/usr/local --sysconfdir=/etc --enable-systemd --
enable-doca
make
make install
systemctl daemon-reload
systemctl restart strongswan.service
```

6.13.8.3 Running the Solution

Run the following commands on both BlueField peers.

1. Run DOCA IPsec Security Gateway in dynamic configuration, assuming the socket location is **/tmp/rules_socket**.

   ```
   doca_ipsec_security_gw -s 03:00.0 -u <sf_net_dev> -c ./ipsec_security_gw_config.json -m transport -i /tmp/
   rules_socket
   ```

   > ⚠ DOCA IPsec Security Gateway application should be run first.

2. Edit the **/etc/strongswan.d/charon/doca.conf** file and add the UDS socket path. If the **socket_path** is not set, the plugin uses the default path **/tmp/strongswan_doca_socket**.

   ```
   doca {

   # Whether to load the plugin
   load = yes

   # Path to DOCA socket
   socket_path = /tmp/rules_socket
   }
   ```

   > ⚠ You must provide the application with this path as well.

3. Restart the strongSwan server:

   ```
   systemctl restart strongswan.service
   ```

   > ⚠ If the application has been run with log level debug, you can see that the connection has been done successfully, and the application is waiting for new IPsec policies.

4. Verify that the **swanctl.conf** file exists in **/etc/swanctl/conf.d/**. directory.

   > ⚠ It is recommended to remove any unused conf files under **/etc/swanctl/conf.d/**.

5. Load IPsec configuration:

   ```
   swanctl --load-all
   ```

6. Start IKE protocol on either the initiator or the target side:

   ```
   swanctl -i --child <child_name>
   ```

In the example above, the child's name is `bf`.

6.13.9 References

- `/opt/mellanox/doca/applications/ipsec_security_gw/`

6.14 DOCA NVMe Emulation Application Guide

This document provides an NVMe emulation implementation on top of the NVIDIA® BlueField® DPU.

6.14.1 Introduction

The NVMe emulation application exhibits how to use the DOCA DevEmu PCI Generic API along with SPDK to emulate an NVMe PCIe function using hardware acceleration to fully emulate the storage device.

6.14.2 Application Architecture

6.14.2.1 Brief Introduction to NVMe

NVMe (Non-Volatile Memory Express) is a high-performance storage protocol designed for accessing non-volatile storage media. NVMe operates over the PCI Express bus, which provides a direct and high-speed connection between the CPU and storage. It enables significantly lower latency and higher input/output operations per second compared to older storage protocols. NVMe achieves this through its scalable, multi queue architecture, allowing thousands of I/O commands to be processed in parallel.

NVMe emulation is a mechanism that allows NVMe device behavior to be simulated in a virtualized or development environment without the need for physical NVMe hardware.

6.14.2.1.1 Controller Registers

NVMe controllers feature several memory-mapped registers located within memory regions defined by the Base Address Registers (BARs). These registers convey the controller's status, enable the host to configure operational settings, and facilitate error reporting.

Key NVMe controller registers include:

- CC (Controller Configuration) – Configures the controller's operational parameters, including enabling or disabling it and specifying I/O command sets.
- CSTS (Controller Status) – Reports the controller's status, including its readiness and any fatal error.
- CAP (Capabilities) – Details the capabilities of the controller.
- VS (Version) – Indicates the supported NVMe version.
- AQA (Admin Queue Attributes) – Specifies the sizes of the admin queues.

6.14.2.1.1.1 Initialization

Initializing an NVMe controller involves configuring the controller registers and preparing the system to communicate with the NVMe device. The process typically follows these steps:

1. The host clears the enable bit in the CC register (sets it to 0) to reset the controller.
2. The host configures the controller by setting initial parameters in the CC register and enables it by setting the enable bit.
3. The host sets up the admin submission and completion queues for handling administrative commands.
4. Administrative commands are issued to retrieve namespace information and perform other setup tasks.
5. The host creates I/O submission and completion queues, preparing the controller for I/O operations.

6.14.2.1.1.2 Reset and Shutdown

Reset and shutdown operations ensure the controller is properly handled and data integrity is maintained.

- The reset process sets the enable bit in CC register to 0, stopping all I/O operations and clearing the controller's state to ensure it returns to a known state.
- Shutdown is initiated by setting the Shutdown Notification (SHN) field in the CC register. This allows the controller to halt operations, flush caches, and ensure data in flight is safely handled before powering off or resetting.

6.14.2.1.2 Completion Queue (CQ)

The Completion Queue in NVMe stores entries that the controller writes after processing commands. Each entry in the CQ corresponds to a command submitted through a Submission Queue, and the host checks the CQ to track the status of these commands.

CQs are implemented as circular buffers in host memory. The host can either poll the CQ or use interrupts to be notified when new entries are available.

6.14.2.1.2.1 Completion Queue Element (CQE)

Each completion queue element in an NVMe Completion Queue (CQ) is an individual entry that contains status information about a completed command, including:

- CID – Identifier of the completed command.
- SQID – ID of the Submission Queue from which the command was issued.
- SQHD – Marks the point in the Submission Queue up to which commands have been completed.
- SF – Indicates the status of the completed command.
- P – Flags whether the completion entry is new.

6.14.2.1.3 Submission Queue

The submission queue (SQ) is where the host places admin and I/O commands for the controller to execute. It operates as a circular buffer, and each SQ is paired with a Completion Queue (CQ). NVMe supports multiple SQs, with each one assigned to a specific CQ.

The controller is notified of new commands via the doorbell mechanism.

6.14.2.1.3.1 Submission Queue Element (SQE)

A Submission Queue Element (SQE) is an individual entry in a Submission Queue that represents a command for the NVMe controller to process. Each SQE contains:

- CID – A unique identifier for the command
- OPC – The opcode that specifies the operation to be performed
- PSDT – Specifies whether PRPs (Physical Region Pages) or SGLs (Scatter-Gather Lists) are used for data transfer associated with the command
- NSID – The identifier of the target namespace for the command

Additionally, the SQE includes fields for command-specific details, such as logical block addresses and transfer sizes.

6.14.2.1.4 Admin Queue Pair (Admin QP)

The Admin Queue Pair consists of an Admin Submission Queue (SQ) and an Admin Completion Queue (CQ), both assigned a command identifier (CID) of 0. There is only one Admin Queue Pair (QP) for each NVMe controller, and it is created asynchronously during the initialization phase. Unlike I/O queues, this QP is dedicated solely to controller management tasks, facilitating the processing of administrative commands.

6.14.2.1.4.1 Admin Commands

- Identify Command (**SPDK_NVME_OPC_IDENTIFY 0x06**)

  This command allows the host to query information from the NVMe controller, retrieving a data buffer that describes attributes of the NVMe subsystem, the controller, or the namespace. This information is essential for host software to properly configure and utilize the storage device effectively.

- Create I/O Submission Queue (**SPDK_NVME_OPC_CREATE_IO_SQ 0x01**)

  This command allows the host to instruct the NVMe controller to establish a new queue for submitting I/O commands.

  Key parameters:

  - SQID – Identifies the specific I/O Submission Queue being created.
  - Queue Depth – Specifies the number of entries the queue can hold.
  - CQID – The identifier of the associated Completion Queue.

  Once the host sends the command with the necessary parameters, the controller allocates the required resources and returns a status indicating success or failure.

- Delete I/O Submission Queue (**SPDK_NVME_OPC_DELETE_IO_SQ 0x00**)

  This command is used to remove an I/O Submission Queue when it is no longer required.

  Key parameters:
    - SQID – The identifier of the I/O Submission Queue to be deleted.

  Once the host issues the command, the controller releases all resources associated with the queue and returns a status confirming the deletion. After the queue is deleted, no additional I/O commands can be submitted to it.
- Create I/O Completion Queue (**SPDK_NVME_OPC_CREATE_IO_CQ 0x05**)

  This command is issued by the host to set up an I/O completion queue in the NVMe controller.

  Key parameters:
    - CQID – The identifier of the I/O Completion Queue to be created.
        - Queue Depth – The number of entries the completion queue can hold.
    - PRP1/PRP2 – Pointers to the memory location where the CQ entries are stored.
    - MSIX – The Interrupt vector associated with this CQ.

  Once the host issues the command, the controller allocates the necessary resources, links the CQ to the specified interrupt vector, and returns a status confirming the creation.
- Delete I/O Completion Queue (**SPDK_NVME_OPC_DELETE_IO_CQ 0x04**)

  This command is issued by the host to remove an existing I/O Completion Queue from the NVMe controller.

  Key parameters:
    - CQID – The identifier of the I/O Completion Queue to be deleted.

  Upcoming receiving this command, The NVMe controller removes the specified CQ and frees all associated resources. Before a CQ can be deleted, all SQs linked to it must be either deleted or reassigned to another CQ. The Controller returns a status code indicating whether the deletion was successful or if an error occurred. Once deleted, the CQ no longer processes completions entries from any linked SQ.
- Get Feature (**SPDK_NVME_OPC_GET_FEATURES 0x0A**)
  This command is issued by the host to query specific features supported by the NVMe controller.
  Key parameters:
    - FID (Feature ID) – Specifies which feature the host wants to retrieve. The controller returns information based on the requested feature. Common features include:
        - Arbitration (FID 0x01)
        - Power Management (FID 0x02)
        - Temperature Threshold (FID 0x04)
        - Error Recovery (FID 0x05)
        - Volatile Write Cache (FID 0x06)
        - Number of Queues (FID 0x07)
        - Interrupt Coalescing (FID 0x08)

Depending on the FID, the feature information may be returned in the Completion Queue Entry (CQE) or written to an output buffer in host memory. If an output buffer is used, the host provides a memory region that the controller accesses via PRP entries or SGL.

- Select – Determines which version of the feature value to return. There are four options:
  - Current (0x0) – Returns the active value of the feature
  - Default (0x1) – Returns the default value of the feature
  - Saved (0x2) – Returns the saved value from non-volatile memory
  - Supported Capabilities (0x3) – Returns the capabilities supported by the controller for the feature

After executing the command, the controller returns a status code in the CQE indicating whether the query was successful.

- Set Feature (**SPDK_NVME_OPC_SET_FEATURES 0x09**)
  This command is issued by the host to modify specific features on the NVMe controller.
  Key parameters:
  - FID (Feature ID): Specifies which feature the host intends to modify. Common features include:
    - Arbitration (FID 0x01)
    - Power Management (FID 0x02)
    - Temperature Threshold (FID 0x04)
    - Error Recovery (FID 0x05)
    - Volatile Write Cache (FID 0x06)
    - Number of Queues (FID 0x07)
    - Interrupt Coalescing (FID 0x08)
  - Data location: depending on the FID, the new value can be provided directly in the SQE or stored in an input buffer located in host memory and is accessible by the controller via PRP or SGL.
  - Save: this field allows the host to specify whether the modification should persist after a controller reset, if set the modified value is saved in the controller non-volatile memory.

After the command is issued and the controller modifies the feature as requested, it returns a status code in the CQE, indicating whether the modification was successful or if an error occurred.

- Log Page (**SPDK_NVME_OPC_GET_LOG_PAGE 0x02**)
  This command is issued by the host to retrieve various types of log pages from the NVMe controller for monitoring and diagnosing the state of an NVMe device.
  Key parameters:
  - LID: log page identifier that specifies the type of log page to retrieve. some common log pages include:
    - SMART / Health Information (LID 0x02): provides device health metrics, temperature, available spare, and more.
    - Error Information (LID 0x01): contains details about errors encountered by the controller.

        - Firmware Slot Information (LID 0x03): information on the firmware slots and active firmware.
        - Telemetry Host-Initiated (LID 0x07): contains telemetry data about device performance.
    - NUMD: the number of DWORDs of log data to return. this allows partial or full-page retrieval.
    - Log page data location: the retrieved log data is written into the output buffer provided by the host. The buffer is accessible by the controller via PRP or an SGL.

When the host issues the get log command, the controller retrieves the requested log information and writes it to the host-provided memory. The controller then returns a status code in the CQE, indicating the success or failure of the operation.

6.14.2.1.5 Input/Output Queue Pair (I/O QP)

An I/O Queue Pair consists of one Submission Queue and its corresponding Completion Queue, which are used to perform data transfers (I/O operations). Multiple I/O Queue Pairs can be created to enable parallel I/O operations. Each queue pair functions independently, maximizing the use of multi-core processors.

- I/O Submission Queue: where the host places read/write/flush commands
- I/O Completion Queue: where the controller posts completion entries after processing the commands

6.14.2.1.5.1 NVM Commands

- Flush (`SPDK_NVME_OPC_FLUSH 0x00`)

The NVMe Flush Command is issued by the host to ensure that any data residing in volatile memory is securely written to permanent storage. If no volatile write cache is present or enabled, the Flush command completes successfully without any effect. Once the Flush operation is finished, the controller updates the associated I/O Completion Queue with a completion entry.

- Read (`SPDK_NVME_OPC_READ 0x02`)

The NVMe read command is one of the core I/O operations in NVMe. This command is issued by the host to retrieve data from a specified namespace and transfer it to the host memory. Key parameters:

- NSID – the identifier of the namespace from which the data is being read.
- LBA – the starting address of the data to be read within the namespace.
- NLB (Number of LBAs) – specifies the size of the read operation in terms of the number of logical blocks to be read.
- Destination buffer – The controller gets the read data from the namespace and sends it to the host memory, where the destination buffer is specified using PRP or SGL.

Upon completion, the controller posts a completion entry to the I/O Completion Queue which includes a status code indicating success or failure.

- Write (`SPDK_NVME_OPC_WRITE 0x01`)

The write command is also one of the core I/O operations in NVMe. this command is issued by the host to write data to a specific namespace at a given logical block address (LBA). Key parameters:

- NSID – The identifier of the namespace where the data is being written
- LBA – The destination address within the namespace where data is written
- NLB – Specifies the size of the write operation in terms of the number of logical blocks to be written
- Source buffer – The data to be written is located in host memory, and the controller reads this data from the source buffer, which is provided using PRP or SGL.

Upon completion, the controller posts a completion entry to the I/O Completion Queue, which includes a status code indicating the success or failure of the write operation.

6.14.2.2 Brief Introduction To SPDK

The Storage Performance Development Kit (SPDK) is an open-source framework that offers tools and libraries for building high-performance, scalable storage solutions, particularly for NVMe devices. SPDK allows applications to implement transport protocols for NVMe over Fabrics (NVMe-oF) by bypassing the kernel and using user-space drivers, enabling direct interaction with storage hardware. This approach significantly reduces latency and overhead, making it ideal for demanding storage environments.

A key component of SPDK is its highly optimized NVMe driver, which operates entirely in user space. By allowing direct communication with NVMe devices without involving the kernel, this driver minimizes I/O latency and boosts performance, supporting both local NVMe storage and remote NVMe devices over NVMe-oF.

6.14.2.2.1 SPDK Threading Model

SPDK's threading model is designed for high concurrency, scalability, and low-latency I/O processing. It operates on a cooperative multitasking model where SPDK threads are assigned to pollers and tasks are executed entirely in user space without kernel involvement. Each SPDK thread runs on a dedicated CPU core, ensuring minimal context switching and enabling tight control over workloads in a non-preemptive environment.

6.14.2.2.1.1 Reactor Threads

At the core of this model are reactor threads, which are SPDK's main execution threads responsible for handling I/O processing and application logic. Each reactor thread is bound to a specific core and runs in polling mode, meaning it continuously polls for tasks and I/O requests rather than relying on interrupts.

Example:

```
struct spdk_thread *thread = spdk_get_thread();
```

This function retrieves the current SPDK thread, which is mapped to a specific core.

SPDK provides the ability to register pollers to reactor threads, which are periodic functions designed to complete asynchronous I/O operations, manage RPC servers, or perform custom

operations. Once a poller completes an operation, it can trigger a user-defined callback to finalize the task.

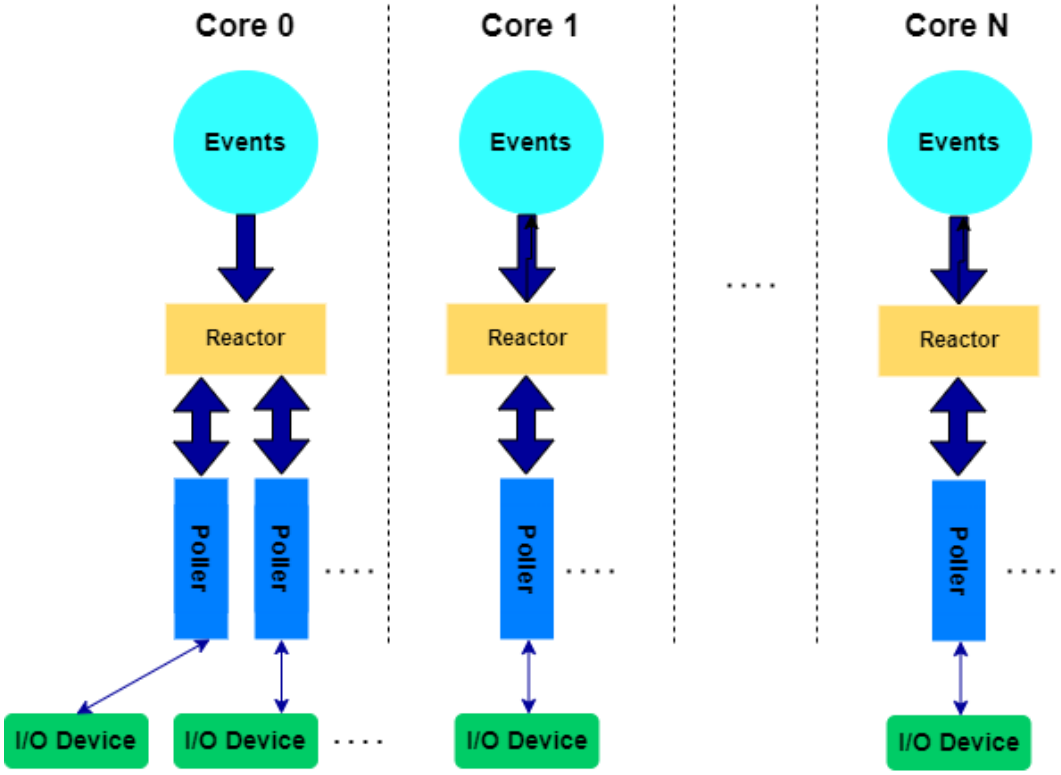
Example of registering a poller:

```
struct spdk_poller *my_poller = spdk_poller_register(my_poll_function, arg, poll_interval_us);
```

Here, **my_poll_function** is invoked repeatedly by the reactor thread.

6.14.2.2.1.2 Poll Groups

Poll Groups consist of multiple SPDK threads working together to manage I/O across multiple devices or connections. A poll group allows a set of threads to coordinate and handle shared workloads, ensuring efficient distribution of tasks across available cores with minimal latency.



6.14.2.2.1.3 Thread Synchronization

In SPDK's cooperative threading model, thread synchronization is designed to be efficient and minimal, as threads do not experience preemptive context switching like traditional kernel threads. This allows for fine-grained control over when tasks are executed. However, there are situations where coordination between threads becomes necessary, such as when handling shared resources or passing tasks between cores.

Instead of relying on traditional locking mechanisms, which can introduce performance bottlenecks due to contention, SPDK uses message passing as the primary method for thread communication. This involves sending events or tasks between threads through a lockless event ring, which allows for coordination without the overhead associated with locks.

Example of sending a message between threads:

```
void send_message(struct spdk_thread *target_thread, spdk_msg_fn fn, void *arg) {
    spdk_thread_send_msg(target_thread, fn, arg);
}
```

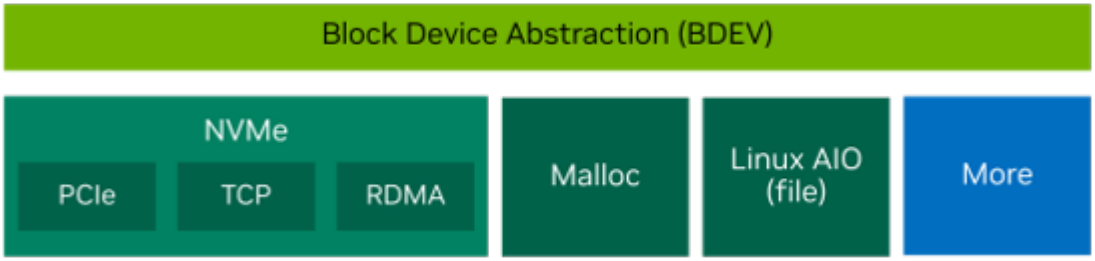
target thread is the SPDK we want to send a message to, **fn** is the function to be executed in the context of the target thread, and **arg** is the argument passed to the function.

6.14.2.2.2 Block Device

SPDK provides a flexible system for working with different types of storage devices, like NVMe SSDs, virtual block devices, AIO devices, and RAM disks. It uses high-performance, user-space APIs to let applications bypass the operating system's kernel, reducing delays and improving performance.

SPDK's block device (bdev) layer gives applications a unified way to perform read and write operations on these devices. It supports popular devices out of the box and allows users to create custom block devices. Advanced features like RAID and acceleration with technologies like DPDK are also supported.

Block devices can be easily created or destroyed using SPDK's RPC server. In NVMe-oF environments, a block device represents a namespace, which helps manage storage across different systems. This makes SPDK ideal for building fast, scalable storage solutions.



6.14.2.2.3 NVMe-oF Target

The NVMe-oF target is a user-space application within the SPDK framework that exposes block devices over network fabrics such as Ethernet, InfiniBand, or Fibre Channel. It typically uses transport protocols like TCP, RDMA, or vfio-user to enable clients to access remote storage.

To use the NVMe-oF target, it is required to configure it to work with one of these transport protocols:

- TCP
- RDMA (over InfiniBand or RoCE)
- vfio-user (mainly for virtual machines)

- FC-NVMe (Fibre Channel, less common in SPDK environments)

6.14.2.2.3.1 NVMe-oF Transport

Each NVMe-oF transport plays a crucial role in facilitating communication between the NVMe-oF initiator (the client) and the target. The transport handles how NVMe commands are transmitted over the network fabric. For example:

- TCP uses IP-based addressing over Ethernet.
- RDMA leverages the low-latency, high-throughput characteristics of InfiniBand or RoCE.
- vfio-user provides virtualization support, allowing virtual machines to access NVMe devices.
- FC-NVMe uses Fibre Channel, often found in enterprise SAN environments.

Additionally, SPDK is designed to be flexible, allowing developers to create custom transports and extend the functionality of the NVMFS target beyond the standard transports provided by SPDK.

The NVMe-oF transport is responsible for:

- Establishing the connection between the initiator and the target.
- Translating network-layer commands into SPDK NVMe commands.
- Managing data transfer across the network fabric.

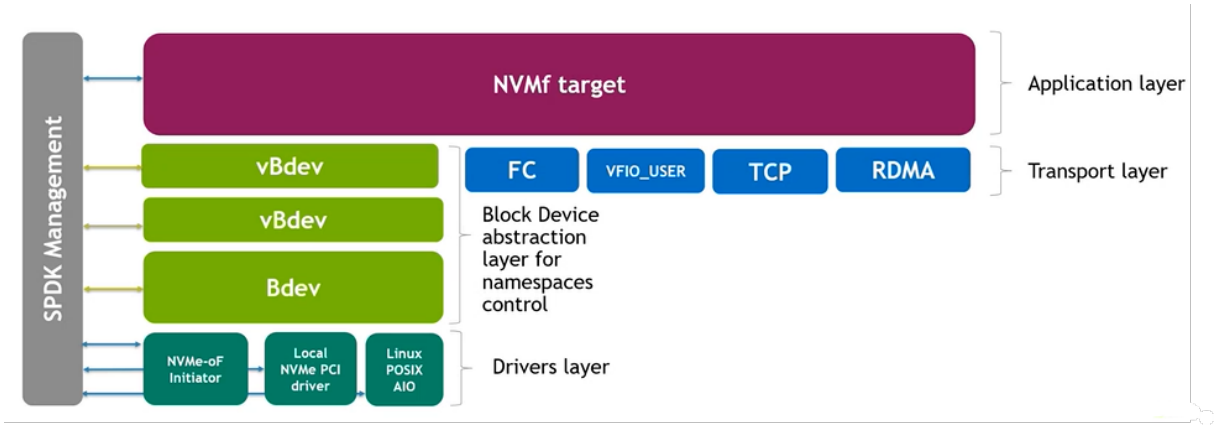
Once a connection is established through the transport, the NVMe-oF target processes the NVMe commands sent by the initiator.

6.14.2.2.3.2 Application Layer

Above the transport layer is the SPDK application layer, which is transport-agnostic. This means that regardless of the transport being used (TCP, RDMA, etc.), the application layer handles NVMe commands uniformly. It is responsible for:

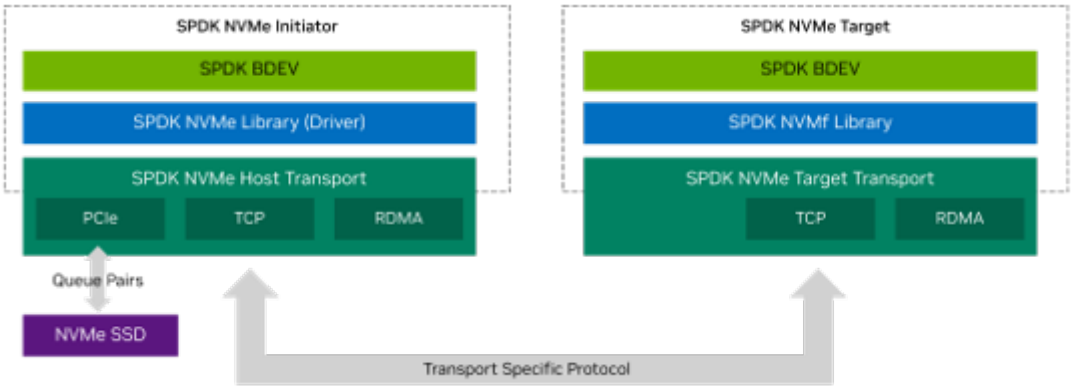
- Managing subsystems, namespaces, and controllers.
- Processing NVMe commands received over the network.
- Mapping these commands to the appropriate storage devices (such as NVMe SSDs or virtual devices like SPDK's **`malloc`** or **`null`** devices).

This uniform application layer ensures that the transport layer interacts with the same logic for processing and responding to NVMe commands, regardless of the underlying network fabric.



6.14.2.2.3.3 NVMe Driver in SPDK

In the context of NVMe storage, the initiator is the host system that needs to access an NVMe storage device to send data or retrieve data from an NVMe storage device, and the driver is responsible for generating and sending the appropriate commands such as read and write. the driver uses the transport to communicate with the NVMe target. the transport sends those commands either over a network in case of remote target or directly to a local NVMe device through the PCIe bus. The target receives these requests from the initiator, processes them and responds back with the data or completion status.



6.14.2.2.4 RPC Server

SPDK's Remote Procedure Call (RPC) server provides a flexible interface for clients to interact with various SPDK services, such as NVMe-oF, block devices (bdevs), and other storage subsystems. The server is based on JSON-RPC and runs within the SPDK application, allowing external clients to dynamically configure, control, and manage SPDK components without the need for application restarts. Through RPC requests, users can create, delete, or query subsystems, configure network storage layers, and manage NVMe-oF targets. The main functionality of RPC server is to process

incoming RPC commands that are usually in JSON format, to execute the functions based on the RPC requests and to return the result back to the client.

The RPC server runs within the SPDK application.

6.14.2.2.5 RPC Client

The RPC client in SPDK interacts with the RPC server to issue commands for configuring and managing various SPDK subsystems. It sends a command, typically in JSON format, asking the server to perform a certain task or retrieve data. The main functionality of RPC client is to construct the request message that it wants the server to process sends the request to the RPC server and receives the response from the server,

The RPC client runs on the user or application side.

6.14.2.2.5.1 Transport RPCs

SPDK provides several transport RPCs are used to configure and manage the transport layer for NVMe-oF. Here are the primary transport RPCs:

- **nvmf_create_transport**
- **nvmf_get_transports**
- **nvmf_subsystem_add_listener**
- **nvmf_get_subsystems**
- **nvmf_delete_listener**
- **nvmf_get_stats**
- **nvmf_delete_transport**

6.14.2.2.5.2 Block Device RPCs

SPDK provides several transport RPCs are used to configure and manage the transport layer for NVMe-oF. Here are the primary transport RPCs:

- **nvmf_create_transport**
- **nvmf_get_transports**
- **nvmf_subsystem_add_listener**
- **nvmf_get_subsystems**
- **nvmf_delete_listener**
- **nvmf_get_stats**
- **nvmf_delete_transport**

6.14.2.2.5.3 Namespace RPCs

Namespace RPCs are used to manage NVMe namespaces within NVMe-oF subsystems or NVMe controllers, here is a list of Namespaces RPCs available:

- **nvmf_subsystem_add_ns**
- **nvmf_subsystem_remove_ns**
- **nvmf_subsystem_get_ns**
- **nvmf_subsystem_get_ns_stats**

- `nvmf_subsystem_get_namespaces`

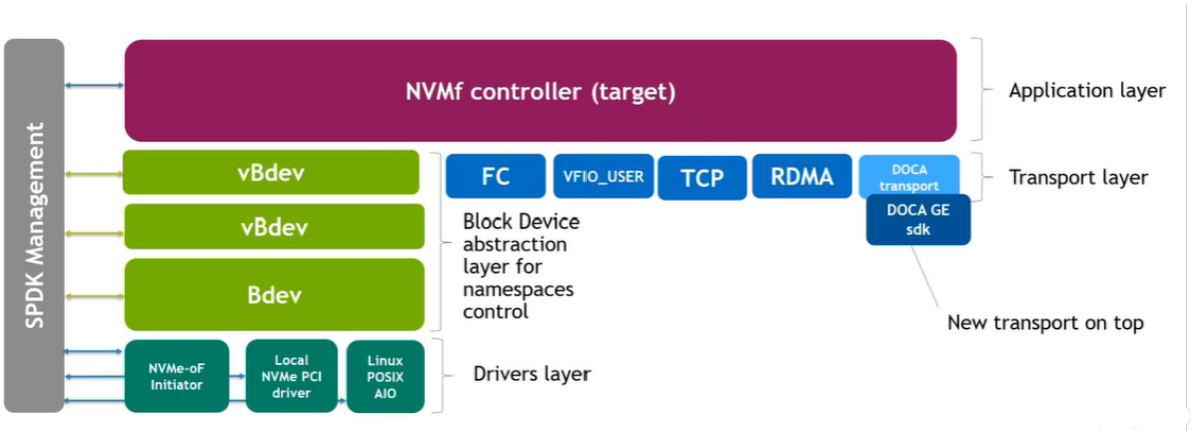
6.14.2.3 Solution Overview

6.14.2.3.1 Integration with SPDK

Using the DOCA Generic PCI Emulation SDK, the BlueField DPU can emulate NVMe devices through PCIe endpoints. This allows the DPU to appear as a physical NVMe device to the host system, enabling the host to send NVMe commands. While the DPU hardware can handle data movement and basic I/O tasks, managing the complete NVMe protocol, including admin commands, queue management, and register operations, requires additional software support. This is where SPDK comes in.

With SPDK, the DPU can offload the complex handling of NVMe commands, making it a fully functional and high-performance NVMe device without the need to develop custom firmware for each command.

6.14.2.3.1.1 DOCA Generic Device Emulation as NVMe-oF Transport



While NVMe-oF is designed for remote transports like TCP or RDMA, SPDK enables us to treat PCIe as another transport option by adding a memory-based transport. This allows the DPU to function as if it's communicating with a remote NVMe-oF target, even though it's local to the host system.

To implement this, we use a DOCA transport, a custom transport layer that acts as a connection tunnel and provides NVMe-oF with generic emulation capabilities. By leveraging SPDK's RPCs for NVMe-oF, we can create a DPU that effectively emulates an NVMe device. The DOCA transport ensures efficient routing and processing of NVMe commands, while SPDK takes care of the software-based emulation.

(The application utilizes the NVMe-oF application transport layer to implement an NVMe emulation solution, inspired by SPDK blogpost.)

6.14.2.3.1.2 Emulated Function as NVMe Controller

In the DOCA transport, the NVMe controller is mapped to a PCIe DOCA device known as the emulation manager. In this context, the emulation manager serves as the hardware interface that provides access to the NVMe controller, which exposes its capabilities through specific PCIe registers and memory-mapped regions.

When connecting a device to a controller, the transport is responsible for providing the controller's unique ID through the Connect command, as specified by the NVMe-oF protocol.

To make the core NVMe-oF target logic work with our DOCA transport, we need to implement specific operations in the **spdk_nvmf_transport_ops** structure. These operations handle tasks like managing connections, transferring data, and processing NVMe commands for DOCA. This structure provides a standard way to connect different transports to SPDK, so the core NVMe-oF logic can work with any transport without needing to know its specific details.

```
const struct spdk_nvmf_transport_ops spdk_nvmf_transport_doca = {
    .name = "DOCA",
    .type = SPDK_NVME_TRANSPORT_CUSTOM,
    .opts_init = nvmf_doca_opts_init,
    .create = nvmf_doca_create,
    .dump_opts = nvmf_doca_dump_opts,
    .destroy = nvmf_doca_destroy,

    .listen = nvmf_doca_listen,
    .stop_listen = nvmf_doca_stop_listen,
    .listen_associate = nvmf_doca_listen_associate,

    .poll_group_create = nvmf_doca_poll_group_create,
    .get_optimal_poll_group = nvmf_doca_get_optimal_poll_group,
    .poll_group_destroy = nvmf_doca_poll_group_destroy,
    .poll_group_add = nvmf_doca_poll_group_add,
    .poll_group_remove = nvmf_doca_poll_group_remove,
    .poll_group_poll = nvmf_doca_poll_group_poll,

    .req_free = nvmf_doca_req_free,
    .req_complete = nvmf_doca_req_complete,

    .qpair_fini = nvmf_doca_close_qpair,
    .qpair_get_listen_trid = nvmf_doca_qpair_get_listen_trid,
};
```

6.14.2.3.1.3 New SPDK RPCs

Since the DOCA transport requires specific configurations that are not covered by the existing SPDK RPCs and differ from other transports, such as managing emulation managers, we need to implement custom RPCs to expose these options to users:

RPC	Description	Details	Arguments	Output	Example

nvmf_doca_get_managers	Provides the ability to list all emulation managers, which are equivalent to DOCA devices	Returns the names of all available local DOCA devices with management capabilities.	None.	If successful, the RPC returns a list of device names for the emulation managers. If it fails, it returns an error code.	`/usr/bin/spdk_rpc.py --plugin rpc_nvmf_doca nvmf_doca_get_managers`
					`[`
					`  {`
					`  "name": "mlx5_0"`
					`  }`
					`]`

RPC	Description	Details	Arguments	Output	Example

`nvmf_doca_create_function`	Provides the ability to create an emulated function under a specified device name.	Creates a new representor device, retrieves its VUID, and then closes the device.	• `-d` – Device name (string)	If successful, the RPC returns the VUID of the newly created function. In case of failure, it returns an error code.	`/usr/bin/spdk_rpc.py --plugin rpc_nvmf_doca nvmf_doca_get_managers`
					`[`
					`  {`
					`  "name": "mlx5_0"`
					`  }`
					`]`

`nvmf_doca_destroy_function`	Provides the ability to destroy an emulated function.	Destroys a DOCA device representor.	• `-d` – The device name associated with the created function (string) • `-v` – The VUID of the function to be destroyed (string)	On success, the RPC returns nothing. On failure, it returns an error code.	`/usr/bin/spdk_rpc.py --plugin rpc_nvmf_doca nvmf_doca_destroy_function -d mlx5_0 -v MT2306XZ00AYGES2D0F0`

`nvmf_doca_list_functions`	Lists all the emulated functions under the specified device name.	Lists all the available representor devices.	• `-d` – Device name (string)	If successful, the RPC returns a list containing the VUID and PCIe address of all the emulated functions under the specified device name. In case of failure, it returns an error code.	`/usr/bin/spdk_rpc.py --plugin rpc_nvmf_doca nvmf_doca_list_functions -d mlx5_0`
					`[`
					`  {`
					`  "Function VUID: ": "MT2306XZ00AYGES1D0F0",`
					`  "PCI Address: ": "0000:00:00.0"`
					`  },`
					`  {`
					`  "Function VUID: ": "MT2306XZ00AYGES2D0F0",`
					`  "PCI Address: ": "0000:00:00.0"`
					`  }`
					`]`

⚠ `/usr/bin/spdk_rpc.py` is the Python script that sends RPC commands to SPDK. The `spdk_rpc.py` script is responsible for handling SPDK commands via the JSON-RPC interface.

6.14.2.3.1.4 Extended RPCs

Some existing SPDK RPCs need to be modified because certain configurations or capabilities of the DOCA transport are not supported by the default RPCs:

RPC	Description	Details	Arguments	Output	Example
nvmf_create_transport	Creates a new NVMe-oF transport by defining the transport type and configuration parameters, allowing the SPDK target to communicate with hosts using the specified transport.	Creates DOCA transport and its resources.	• **-t** – The transport type (string) • **-u** – The number of I/O queue pairs per system. (Optional) • **-i** – The maximum I/O size in bytes. (Optional) • **-c** – The maximum of inline data size in bytes. (Optional)	None	`/usr/bin/spdk_rpc.py nvmf_create_transport -t doca -u 8192 -i 131072 -c 819`
nvmf_subsystem_add_listener	Adds a listener to an NVMe-oF subsystem, enabling it to accept connections over a specified transport.	Hot-plugs the device, allowing the host to interact with it as an NVMe device.	• The subsystem NQN (NVMe qualified name) to which to add the listener. • **-t** – The transport type for this listener. • **-a** – The VUID of the function we want to use to create the emulated device.	None	`/usr/bin/spdk_rpc.py nvmf_subsystem_add_listener nqn.2016-06.io.spdk:cnode1 -t doca -a MT2306XZ00AYGES1D0F0`

/usr/bin/spdk_rpc.py is the Python script that sends RPC commands to SPDK. The **spdk_rpc.py** script is responsible for handling SPDK commands via the JSON-RPC interface.

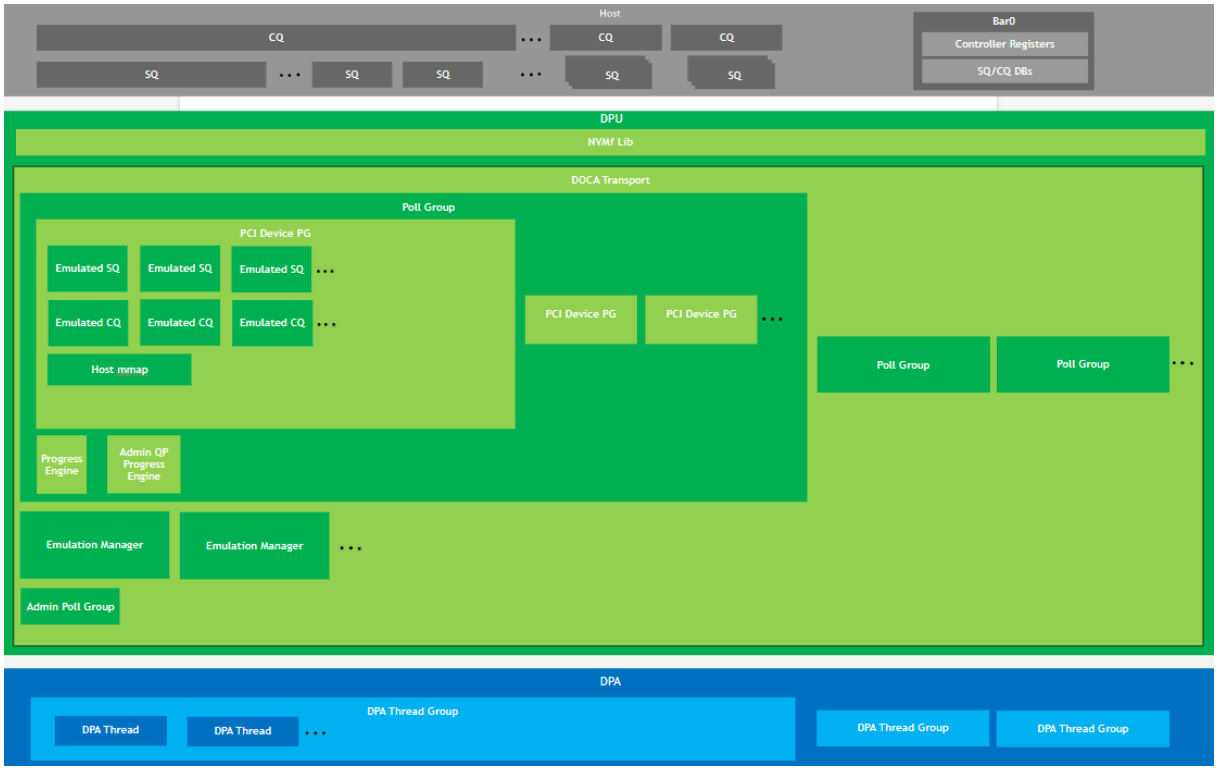
6.14.2.4 Data Structures

To implement the APIs for `spdk_transport_ops` mentioned above, we created transport-specific data structures that efficiently interact with these APIs. These structures are designed to manage the transport's state, connections, and operations.

The upper layer represents the host, followed by the DPU, and further down is the DPA, which is part of the DPU. Within the DPU, the NVMe-oF application is running, divided into two parts: the NVMe-oF library, which we use as a black box, and the DOCA transport, which we implement.

In the DOCA transport, there are several poll groups, each representing a thread. In addition to the poll group list, the transport maintains an emulation managers list, consisting of all devices managed by this transport. There is also a special poll group instance dedicated to polling PCIe events.

If we dive into the poll group, there are two progress engines: one that polls the I/O queues and another that handles the admin queues. Additionally, there is a list of PCIe device poll groups, each of which manages the completion queues, submission queues, and host memory mappings for a specific device.



The following sections provide more information on each of the main structures.

6.14.2.4.1 Emulation Manager

The first structure to address is the emulation manager context. During initialization, the DOCA transport scans for all available DOCA devices that can serve as emulation managers. For each of

these devices, it creates a PCIe type, initializes a DPA instance, assigns a DPA application, and starts the DPA process. It also opens the DOCA device (emulation manager). All of these devices are stored within the emulation manager context, where they are tracked for as long as the transport is active.

DOCA Emulation Manager
DOCA device
PCI type
DPA

6.14.2.4.2 PCIe Device Admin

This is the representor of the emulated device. and it contains the following:

DOCA PCI Device Admin
Doca transport
Emulation managers
PCI device
SPDK subsystem
Device representor
Transport ID
Doca Listener state
Controller ID
Stateful region values
SDPK NVMF controller
Doca admin QP
Admin QP poll group
FLR flag
Destroy flow falg

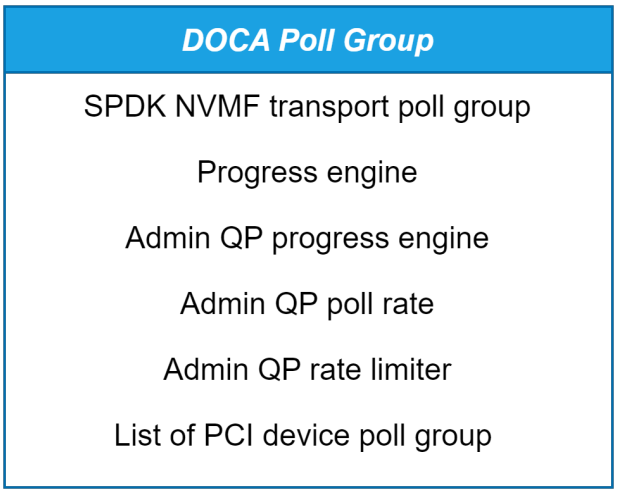
- A pointer to the DOCA transport this device belongs to.
- The emulation manager described previously
- The emulated PCIe device
- A pointer to the SPDK subsystem this device belongs to
- The transport ID this device belongs to
- A pointer to SPDK NVMe-oF controller
- The stateful region values and are updated after each query
- Admin Queue Pair context
- Admin Queue Pair poll group that manages the admin QP, and it is selected from the system using the round-robin method
- The FLR flag Indicates if an FLR event has occurred
- Destroy flag that indicates if PCIe device should be destroyed

Once the user issues the add listener RPC, this context is established to facilitate the hot-plug of the emulated device to the host, enabling it to start listening for interactions from the host.

6.14.2.4.3 DOCA Poll Group

Each poll group in SPDK is associated with a thread running on a specific CPU core. For each core, a reactor thread is responsible for executing pollers, and the poll group is one of those pollers. When an NVMe-oF is created, it is assigned to each poll group so that the transport can handle I/O and connection management for the devices across multiple CPU cores, so each transport has a representative within each poll group.

DOCA poll group Fields:



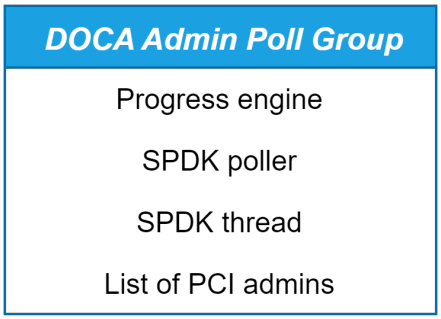
- SPDK NVMe-oF transport poll group (**struct spdk_nvmf_transport_poll_group**): refers to a structure that is part of the NVMe-oF subsystem and is responsible for handling transport specific I/O operations at the poll group level.
- Progress engine (**struct doca_pe**): When each poll group runs its set of pollers, it also invokes the DOCA progress engine to manage transport operations. On the DOCA side, the

doca_pe_progress function is called to drive the progress engine within each poll group. This is how DOCA's PE integrates into SPDK's poll group mechanism.

- Admin QP progress engine (**struct doca_pe**): Another progress engine (**struct doca_pe**) dedicated to handling admin queues while the previous one is dedicated to handling the I/O queues. this separation allows for more control over the polling rate of each queue type, which helps optimize performance.
- Admin QP poll rate and Admin QP rate limiter: They determine how often the system checks the admin queue for new commands.
- List of PCIe device poll group: List of devices that this poll group typically polls for their queues.

6.14.2.4.4 Admin Poll Group

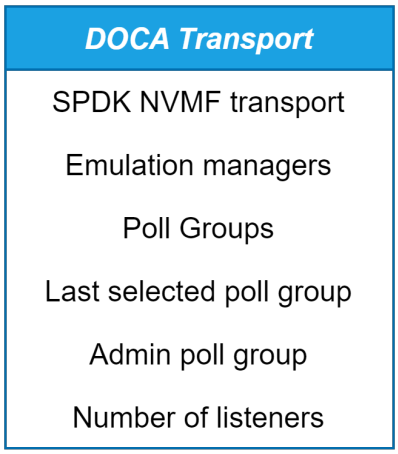
This object is a per-transport entity, functioning as a dedicated unit for polling PCIe events and managing Admin Queue activities.



- Progress engine – The DOCA used by the poller.
- SPDK Poller – This poller is used on SPDK application thread. It continuously monitors for PCIe events such as FLR, stateful region and hot plug events.
- SPDK thread – The application thread associated with the currently executing SPDK thread.
- PCIe admins – List of all the PCIe device admins.

6.14.2.4.5 DOCA Transport

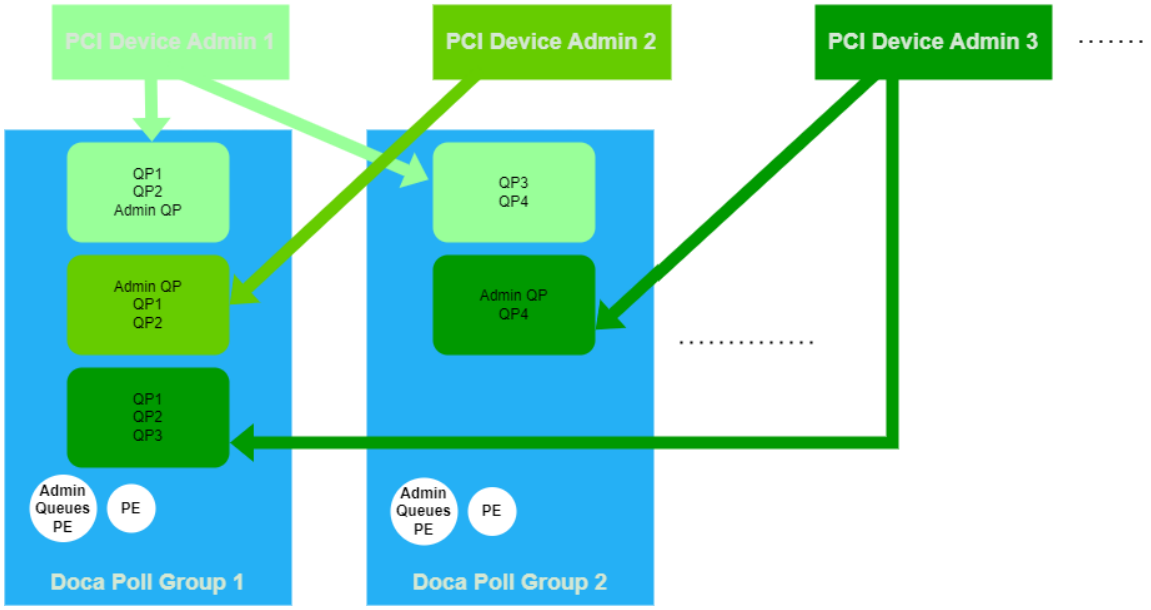
This structure holds the overall state and configuration of the transport as it includes:



- SPDK NVMe-oF transport – Defines the transport layer within SPDK NVMe-oF framework. It holds essential data for managing the transport, such as configuration parameters, operational states, and connections.
- Emulation managers – Includes all the devices managed by this transport
- Poll groups – Contains all the poll groups actively polling for this transport
- Last selected poll group – Used to assist with round-robin selection of poll groups, ensuring even distribution of workload across poll groups during transport operations
- Admin poll group – described previously
- Number of listeners – Number of devices within this transport

6.14.2.4.6 PCIe Device Poll Group

Based on the previous descriptions of the transport structures, the following diagram illustrates the relationship between PCIe devices and the poll groups within a transport. Each PCIe device contains I/O and admin queues, which are distributed across the poll groups.



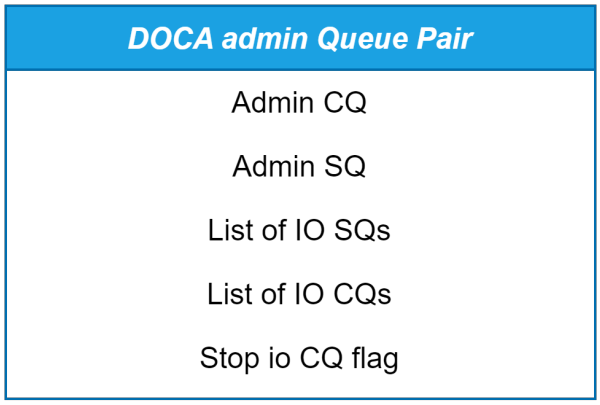
This relationship is managed by a structure called the PCIe dev poll group (**struct nvmf_doca_pci_dev_poll_group**), which holds the device's memory map (mmap), the PCIe device's admin details, admin QPs (if applicable), a list of I/O queues, and the poll group responsible for polling those queues.

DOCA PCI Dev Poll Group
Host mmap
PCI device admin
Admin QP
List of IO CQs
Parent Poll group

When creating a new I/O or admin queue for a specific device and poll group, we first check if a PCIe dev poll group structure already exists that links the two. If not, we create a new **struct nvmf_doca_pci_dev_poll_group** to combine them.

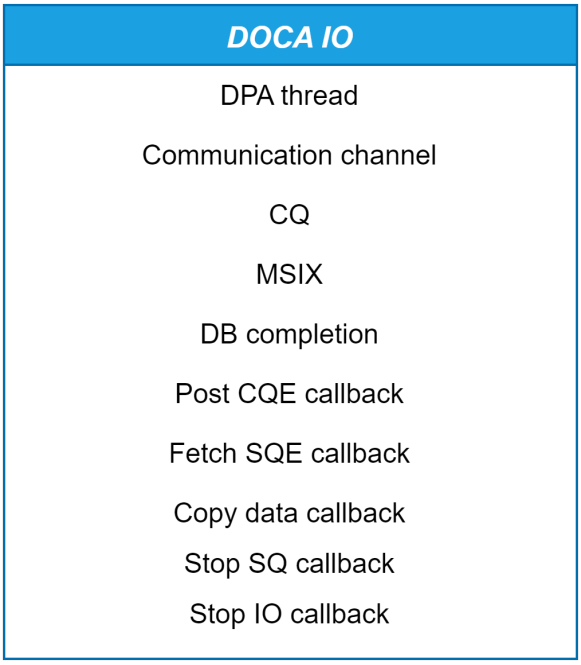
6.14.2.4.7 Admin QP

This structure manages the queues for a specific device. It holds an admin completion queue and admin submission queue to handle admin command operations. Additionally, it includes lists for I/O completion queues and I/O submission queues, which manage data-related operations. The structure also contains a flag `stopping_all_io_cqs` to indicate whether all completion queues should be stopped, used for gracefully halting the device's queue processing when needed.



6.14.2.4.8 DOCA IO

The I/O structure is responsible for managing I/O operations, including receiving doorbells on the completion queue (CQ) and its associated submission queues (SQs). It handles reading SQ entries (SQEs) from the host, writing SQEs back to the host, and raising MSI-X interrupts. This structure contains a single CQ, along with an MSI-X vector index that is raised by the DPA, and a doorbell completion that is polled by the DPA thread.



It also holds several callbacks:

- Post CQE – Invoked once CQE is posted to the host CQ, freeing resources afterward
- Fetch CQE – Invoked when an SQE is fetched from the host, parsing and executing the request
- Copy data – Invoked after data is copied to the host, completing the request and freeing resources
- Stop SQ – Invoked when an admin or I/O SQ is stopped, to fully release resources
- Stop IO – Invoked when an admin or I/O CQ is stopped, to complete resource cleanup

The function `nvmf_doca_io_create` synchronously creates an emulated I/O.

6.14.2.4.8.1 DOCA CQ

The CQ's main task is to write cookies to host. The main fields are:

DOCA CQ
SQ ID
DOCA queue
Doorbell
Consumer index (CI)
Producer index (PI)

- In case it is an Admin CQ then the CQ ID is zero.
- The DOCA queue is a shared structure that acts as both the completion queue (admin and I/O CQ) and submission queue (admin and I/O SQ), mirroring the host's queues with the same size. It is responsible for fetching Submission Queue Entries (SQEs) from the host and posting Completion Queue Entries (CQEs) back to the host. The NVMe driver issues commands through its submission queues and receives their completions via the completion queues. To facilitate efficient processing by the DPU, the DOCA queue leverages DMA to handle data transfers between the host and DPU in both directions. Each queue is equipped with a pointer to a DMA structure that contains the necessary resources for these operations. During initialization, DMA resources and local buffers are allocated based on the queue's size. The DOCA queue also maintains an array of tasks, with each task at index idx corresponding to and synchronized with the task at the same index in the host's queue. When DMA operations are required, these resources are utilized for data transfer. The below outlines its main fields:

DOCA Queue
Buffer Inventory
DMA
Local queue MMAP
Local queue address
Elements (DMA tasks)
Number of elements

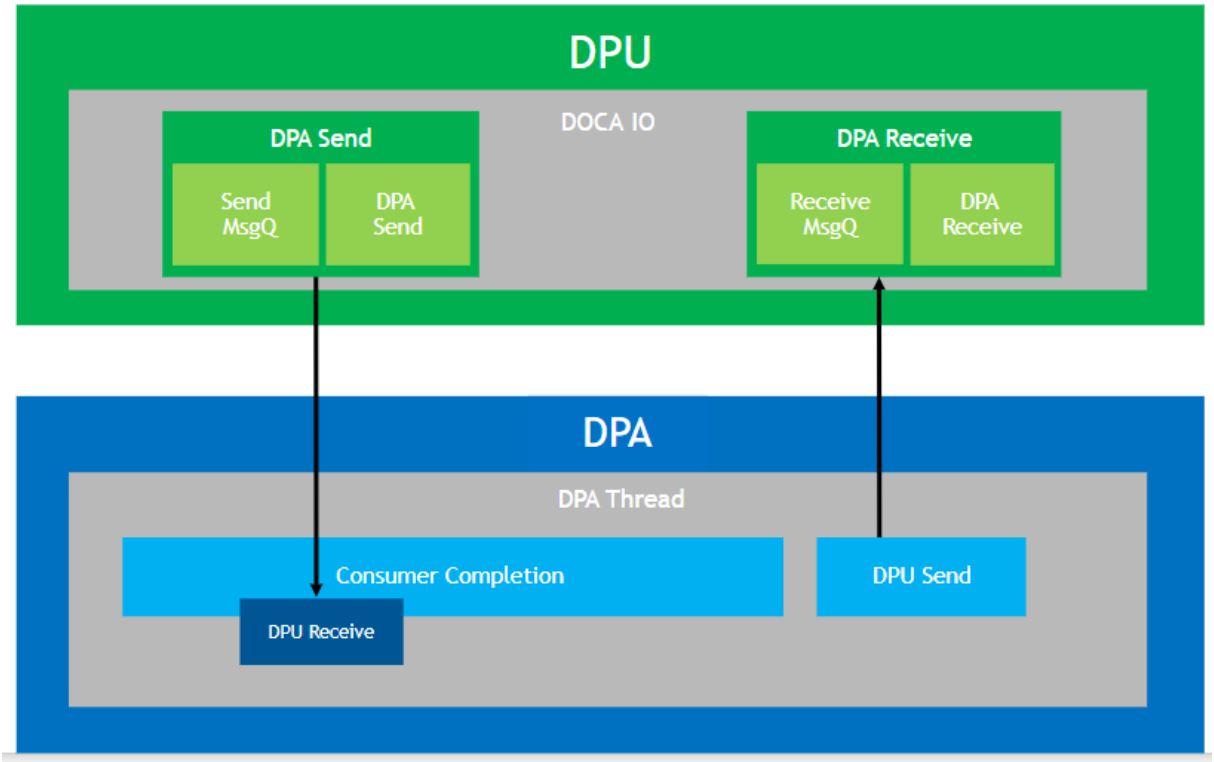
  - The buffer inventory is for allocating the queue elements
  - DMA context handles data transfer to and from the host

- The queue MMAP represents the local memory where elements are stored for copying
- The local address for copying the elements to or from the host
- The elements themselves are DMA tasks for copy/write
- The maximum number of elements the queue can hold

6.14.2.4.8.2 DOCA Comch

A full-duplex communication channel is used to facilitate message passing between the DPA and DPU in both directions. This channel is contained within the DOCA IO and consists of two key components:

- A send message queue (**nvmf_doca_dpa_msgq**) with a producer completion (**doca_dpa_completion**) context.
- A receive message queue (**nvmf_doca_dpa_msgq**) with a consumer completion (**doca_comch_consumer_completion**) context.

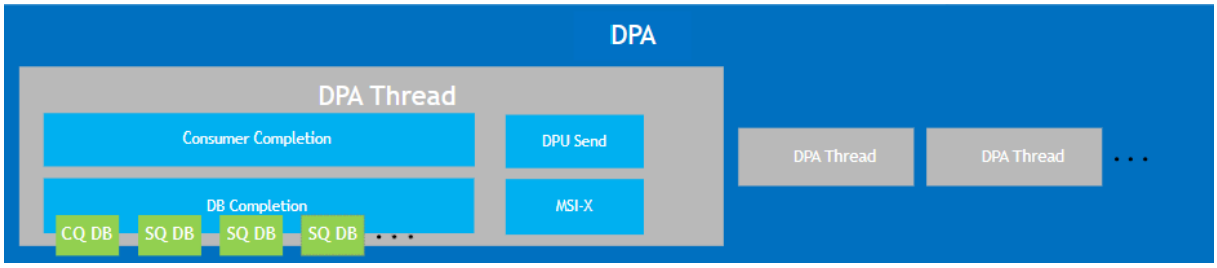


6.14.2.4.8.3 DOCA DPA Thread

In addition to the communication channel, the I/O structure also includes a DPA thread context, (with each thread dedicated to a single CQ). It consists of:

- A pointer to the DPA
- A pointer to the DPA thread
- The necessary arguments for the DPA thread, include:

- Consumer Completion: Continuously polled by the DPA thread to detect new messages from the host.
- Producer Completion: Monitored to verify if messages have been successfully sent.
- DB Completion Context: Provides the doorbell values and connects to both CQ and SQ doorbells.
- MSIX: Allows the device to send interrupts to the host.

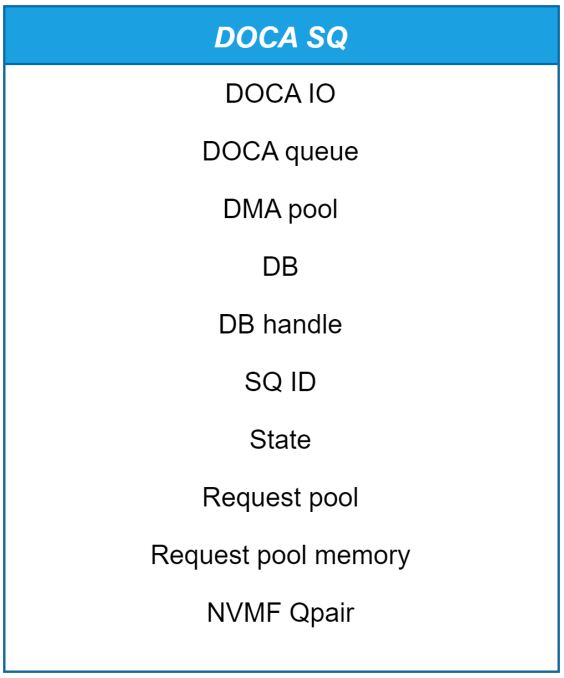


The DPA thread performs three main operations:

- **nvmf_doca_dpa_thread_create** – Creates a DPA thread by providing the DOCA DPA, the DPA handle, and the size of the arguments to be passed to the DPA. It also allocates memory on the device.
- **nvmf_doca_dpa_thread_run** – Copies the arguments to the DPA and runs the thread.
- **nvmf_doca_dpa_thread_destroy** – Deletes the thread and frees the allocated arguments."

6.14.2.4.9 DOCA SQ

The main fields that construct the DOCA submission queue (**nvmf_doca_sq**) include:

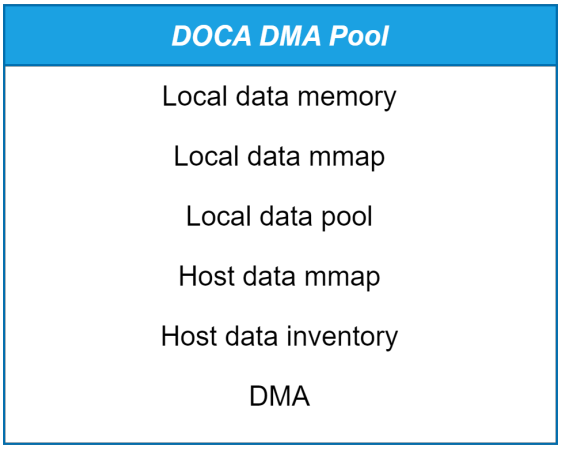


- A reference to the DOCA I/O to which this submission queue belongs, and where its completion is posted. Multiple SQs can belong to a single I/O.
- The DOCA queue, which was previously described, that handles copying Submission Queue Entries (SQEs) from the host.
- A pool of DMA (**nvmf_doca_dma_pool**) data copy operations (to be defined shortly).
- The doorbell associated with this submission queue.
- The DPA handle of the doorbell.
- The submission queue identifier (SQID).
- The state of the submission queue, used for monitoring purposes.
- NVMe-oF request pool memory (to be defined shortly).
- A list of NVMe-oF DOCA empty requests, used whenever a new SQE is received and a request needs to be prepared.
- The Queue Pair (QP) created for this submission queue by the NVMe-oF target, which is used to execute commands.

The submission queue creation is asynchronous because, when we create the completion queue (CQ), we also initialize the DPA. However, when creating the SQ, the DPA is already running, so we need to update the DPA about the newly added SQ. Directly modifying its state could lead to synchronization issues, which is why we use a communication channel, making the process asynchronous.

6.14.2.4.9.1 DOCA DMA Pool

As previously noted, a Submission Queue (SQ) includes the **nvmf_doca_dma_pool** structure, which manages data transfer operations between the host and the DPU, in both directions. It consists of the following elements:



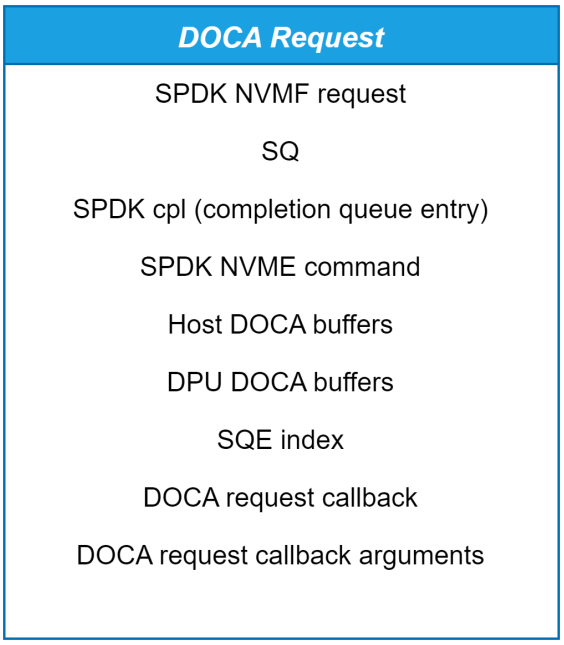
- Memory allocated for local data buffers
- Memory-mapped region for the local data buffers
- A pool of local data buffers
- Memory mapping that provides access to host data buffers
- An inventory for allocating host data buffers
- A DMA context used for transferring data between the host and the DPU

This structure is initialized whenever the SQ is created. The size of the local data memory is determined by multiplying the maximum number of DMA copy operations by the maximum size in bytes for each DMA copy operation. All these local buffers are allocated during the creation of the SQ. The key operations performed on the DMA are:

- **nvmf_doca_sq_get_dpu_buffer** – Retrieves a buffer in DPU memory, enabling data transfers between the host and the DPU.
- **nvmf_doca_sq_get_host_buffer** – Retrieves a buffer pointing to host memory, also used for data transfers between the host and the DPU.
- **nvmf_doca_sq_copy_data** – Copies data between the host and the DPU. This operation is asynchronous, and upon completion, it invokes the **nvmf_doca_io::copy_data_cb** callback function.

6.14.2.4.9.2 DOCA NVMe-oF Request

The NVMe-oF target utilizes requests to handle incoming commands. When a new command is received by any transport (not limited to the DOCA transport), it creates an instance of the struct **spdk_nvmsf_request** . This structure contains various elements, including the NVMe command, the Submission Queue (SQ) to which the command belongs, the Queue Pair (qpair), the IO Vector (IOV), and other relevant data. In our design, we introduce a new wrapper structure called **nvmf_doca_request** , which encapsulates the NVMe-oF request structure along with additional fields specific to the DOCA transport. The main fields included in this structure:



Where:

- SPDK request is and instance of struct **spdk_nvmf_request**, which represents the command being processed.
- Host and DPU DOCA buffers are pointers to data buffers located at the host or DPU, containing data associated with this command.
- DOCA request callback is the function invoked upon request completion, receiving the appropriate DOCA request callback arguments.

The key operations performed on requests include:

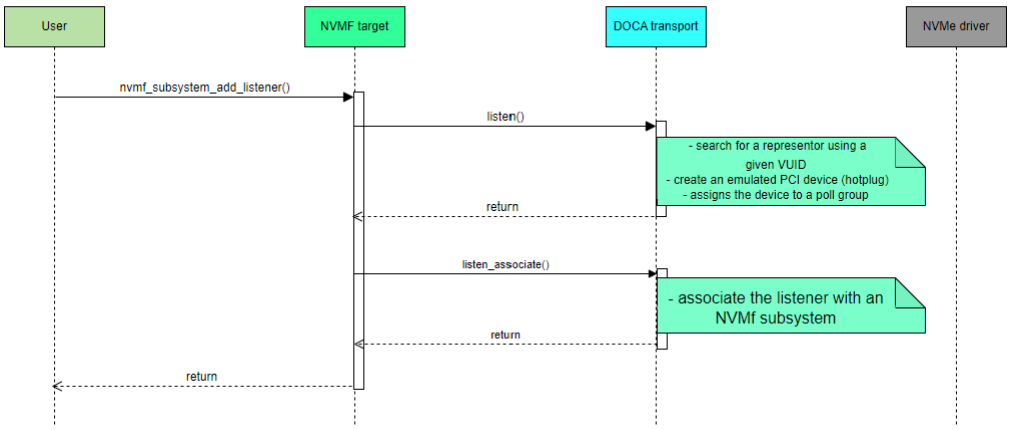
- **nvmf_doca_request_pool_create** – When the SQ is created, a request pool is allocated with a size matching the SQ depth.
- **nvmf_doca_request_pool_destroy** – This function destroys the request pool when the SQ is removed.
- **nvmf_doca_request_get** – Retrieves an NVMe-oF request object from the pool associated with a specific SQ, and is called after fetching an SQE from the host.
- **nvmf_doca_request_complete** – Completes an NVMe-oF request by invoking its callback and then releasing the request back to the pool.

6.14.2.5 Control Path Flows

6.14.2.5.1 DOCA Transport Listen

6.14.2.5.1.1 Hotplug and Hotunplug

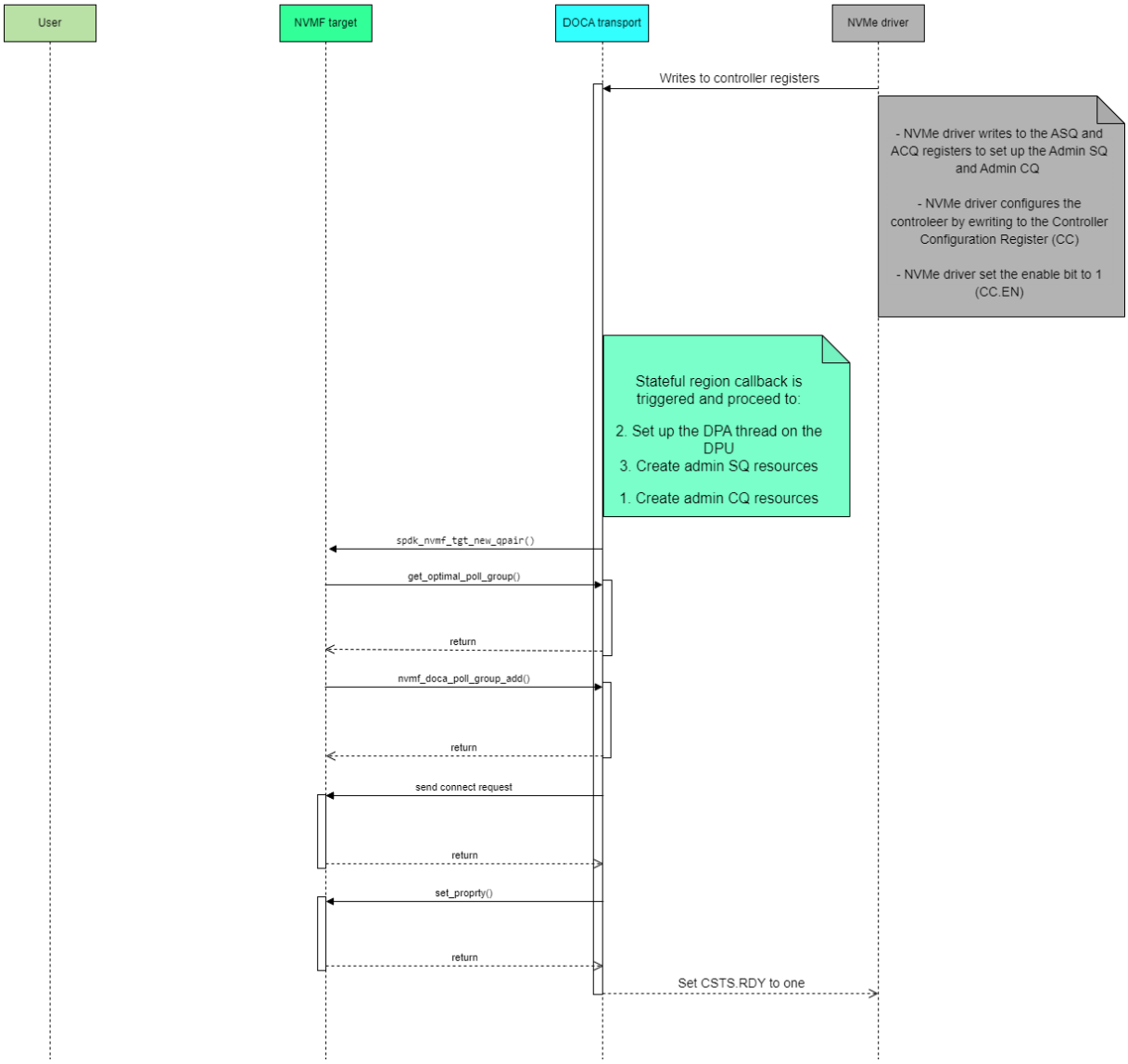
1. Start with **add_listener** RPC trigger – The flow begins when the **add_listener** RPC is called. SPDK then pauses all poll groups.
2. Lookup for representor by VUID – The transport searches for a representor using a given VUID.
3. Create emulated PCIe device – Once the representor is found, the transport creates an emulated PCIe device and assigns it to a poll group.
4. Initialize memory mapping (mmap) – For each poll group, the transport sets up a memory-mapped area representing the host memory.



At this stage, the NVMe driver detects the newly hot-plugged device.

6.14.2.5.2 Controller Register Events

6.14.2.5.2.1 Initialization



- Controller Initialization – The process begins by configuring the controller registers:
  - The NVMe driver writes to the ASQ and ACQ registers to set up the Admin Submission Queue and Admin Completion Queue.
  - The driver configures the controller by writing to the controller configuration (CC) register, setting parameters like memory page size, arbitration, and timeout values.
  - The driver sets the CC.EN bit in the CC register to 1, transitioning the controller from **Disabled** to **Enabled**.

    - The NVMe driver waits for the CC.RDY (Controller Status Register - Ready bit) to become 1. This indicates that the controller has successfully completed its initialization and is ready to process commands.
- Stateful Region Callback Trigger – At this point, the PCI-emulated device triggers a callback to the stateful region. This callback detects the host's initialization process by checking the changes in the enable bit. This callback may occur multiple times, but it only proceeds if the enable bit has been altered.
- CQ and DPU Setup – The callback proceeds to create the Completion Queue (CQ) resources and sets up the DPA thread on the DPU. The DPA thread is equipped with two message queues: one for sending and one for receiving.
- Binding CQ Doorbell to DB Completion – An RPC is sent to bind the CQ doorbell to the DB completion context. This is done while the DPA is not yet active, preventing synchronization issues.
- SQ Resource Creation – The Submission Queue (SQ) resources are created, including the SQE pools and the local buffer size needed for copying SQEs from the host to the DPU. The DOCA DMA is used for the data transfer operations.
- SQ Bind DB Message – The SQ sends a "bind DB" message to the DPA.
- DPA Receives Bind DB Message – The DPA processes the "bind DB" message and sends the SQ's doorbell information to the DB completion context.
- SQ Sends Bind Done Message – The SQ sends a "bind done" message to the DPU.
- Start SQ DB – The DPU receives the "bind done" message and starts the SQ DB.
- NVMe-oF QPair and Request Pool Creation – An NVMe-oF QPair and an NVMe-oF request pool are created.
- Asynchronous QPair Creation – The NVMe-oF library starts an asynchronous operation to create the QPair.
- NVMe-oF Library Calls –
    - The library creates the QPair and calls the transport to get the optimal poll group.
    - It then calls **poll_group_add** on the selected thread.
- NVMe-oF Connect Request – The transport sends an NVMe-oF connect request.
- Set Property Request – After the connect request is complete, a **set_property** request is sent to update the controller configuration as provided by the host during initialization.
- Callback Triggered – Once the **set_property** request is finished, the NVMe-oF triggers callbacks.
- Update Stateful Region – The transport updates the stateful region, setting the **CSTS.RDY** bit to 1.
- Host Polling Unblocked – With the CSTS.RDY set to 1, the host polling is now unblocked, completing the initialization process.

6.14.2.5.2.2 Reset and Shutdown Flow

The reset flow in NVMe using SPDK is crucial for maintaining the integrity and stability of the storage subsystem, to allow the system afterwards to recover gracefully.

The reset process can be initiated by:

- The Host: The host can initiate a shutdown or reset of the NVMe controller by configuring specific registers in the controller's register space. in this case the **handle_controller_register_events()** function is triggered.

- Configuring Shutdown Notification (SHN): The host can write to the CC (Controller Configuration) register, specifically the SHN field, to specify how the controller should handle a shutdown:
  - Normal Shutdown – **SPDK_NVME_SHN_NORMAL** – Allows for a graceful shutdown where the controller can complete outstanding commands.
  - Abrupt Shutdown – **SPDK_NVME_SHN_ABRUPT** – Forces an immediate shutdown without completing outstanding commands.
- Resetting the NVMe controller by setting the CC.enablebit to zero. Also **handle_controller_register_events()** function is triggered in this case.
- The host can initiate a reset the NVMe controller by using Function-Level Reset (FLR). In this case the **flr_event_handler_cb()** is triggered.

- DOCA transport can initiate reset flow through **nvmf_doca_on_initialization_error()** when it detects an internal error condition during initialization flow.

Once a shutdown or reset is requested, the transport proceeds to destroy all resources associated with the controller. If it is a shutdown request, the shutdown status is updated accordingly. When the host performs a function-level reset (FLR) or a controller reset, the transport must take several actions: it destroys all submission and completion queues (SQs and CQs) across all poll groups for the specified device, destroys the admin SQ and CQ via the admin thread, stops the PCIe device, and then restart it.

Regardless of the reason for the reset flow, it all starts from **nvmf_doca_pci_dev_admin_reset()**. This function marks the beginning of the asynchronous process for resetting the PCIe device NVMe-oF context. The flow consists of callbacks that are triggered in sequence to track the completion of each process and to proceed to the next one. Let us now illustrate the rest of the flow:

- If the admin QP exists, the process first checks for any I/O submission queues (SQs).
- If I/O SQs are found, an asynchronous flow begins to stop all I/O SQs.
- For each I/O SQ associated with the admin queue, it retrieves the corresponding poll group responsible for destroying its specific SQ, as no other poll group can perform this action.
- Once all I/O SQs are stopped, if any I/O completion queues (CQs) remain, a message is sent to each poll group instructing them to delete their I/O CQs.
- After all I/O queues are destroyed, the flow proceeds to destroy the admin CQ and SQ.

After this flow is done, it moves to **nvmf_doca_pci_dev_admin_reset_continue()** to finalize the reset flow:

- If a reset is issued by configuring the NVMe controller registers, then set **CSTS.BITS.SHST** to **SPDK_NVME_SHST_COMPLETE** and **CSTS.BITS.RDY** to 0
- If the reset is triggered by a FLR, then stop the PCIe device context: **doca_ctx_stop(doca_devemu_pci_dev_as_ctx())**

6.14.2.5.3 I/O QP Create/Destroy

The process of creating and destroying I/O Queue Pairs (QPairs) begins with the initiator (host) sending an NVMe-oF connect command to the NVMe-oF target following the completion of transport initialization. The host sends an NVMe-oF connect command to the NVMe-oF target after the transport has completed its initialization. The NVMe-oF target receives an admin

command and begins processing it through **nvmf_doca_on_fetch_sqe_complete()**. Based on the command opcode, the following steps are executed:

1. Creating an I/O Completion Queue – **SPDK_NVME_OPC_CREATE_IO_CQ** – **handle_create_io_cq()**:
    a. First, the system selects a poll group that is responsible for creating the CQ.
    b. The target searches for the **nvmf_doca_pci_dev_poll_group** entity within the selected poll group. If it is not found, it indicates that this is the first queue associated with the specific device managed by this poll group, necessitating the creation of a new entity.
    c. Next, the target allocates a DOCA IO (**nvmf_doca_io**) using the attributes and data provided in the command, such as CQ size, CQ ID, CQ address, and CQ MSI-X.
    d. Create the DMA context, the message queues, consumer and producer handle and the DPA thread.
    e. Once the asynchronous allocation and setup are complete, the target posts a Completion Queue Entry (CQE) to indicate that the operation has succeeded. it is done through static void **nvmf_doca_poll_group_create_io_cq_done()**.
2. Creating an I/O Submission Queue – **SPDK_NVME_OPC_CREATE_IO_SQ** – **handle_create_io_sq()**:
    a. Based on the CQ ID argument provided in the command, the system first identifies the I/O entity to which the new Submission Queue (SQ) should be added. From this entity, it retrieves the associated **nvmf_doca_pci_dev_poll_group**.
    b. Next, a DOCA Submission Queue **nvmf_doca_sq** is allocated using the attributes and data specified in the command, including the SQ size, SQ ID, and SQ address. This allocation is handled within the **nvmf_doca_poll_group_create_io_sq()** function.
    c. The DMA context and the DB completions are also created.
    d. Once the asynchronous process of creating the SQ is complete, a Completion Queue Entry (CQE) is posted via the **nvmf_doca_poll_group_create_io_sq_done()** function.
    e. Next, the target begins the QPair allocation, which creates a new Queue Pair comprising a submission queue (SQ) and a completion queue (CQ).
    f. Then, it determines the optimal poll group for the new QPair by calling **get_optimall_poll()** group ensuring that both the CQ and SQ attached to it runs on the same poll group.
    g. After identifying the appropriate poll group, the target adds the newly created QPair to it using **nvmf_doca_poll_group_add()**, enabling management of the QPair's events and I/O operations.
    h. After the connection is established, the initiator can start sending I/O commands through the newly created QPair.
3. Destroying an I/O Completion Queue – **SPDK_NVME_OPC_DELETE_IO_CQ** – **handle_delete_io_cq()**:
    a. The process starts by fetching the identifier of the queue that needs to be deleted from the NVMe request.
    b. Once the identifier is retrieved, the corresponding **nvmf_doca_io** entity is located.
    c. The associated poll group is then extracted from the I/O, as it is responsible for destroying the CQ associated with it. The thread retrieved schedules the execution of **nvmf_doca_pci_dev_poll_group_stop_io_cq()** using **spdk_thread_send_msg**.

d. The **nvmf_doca_io_stop()** function is called to initiate the stopping process. If there are CQs in this I/O that are not idle, it triggers **nvmf_doca_io_stop_continue()** to advance the sequence. This flow then executes a series of asynchronous callback functions in order, ensuring that each step completes fully before the next begins, performing the following actions:
    i. Stop the DOCA devemu PCIe device doorbell to prevent triggering completions on the associated doorbell completion context
    ii. Stop the NVMe-oF DOCA DMA pool
    iii. Stop the DMA context associated with the Completion Queue and frees all elements of the NVMe-oF DOCA Completion Queue.
    iv. Stop the NVMe-oF DOCA DPA communication channel, halting both the receive and send message queues.
    v. Stop and destroy the PCIe device DB.
    vi. Stop the MSI-X.
    vii. Stop and destroy the DB completion.
    viii. Destroy the communication channel.
    ix. Destroy the DPA thread.
e. Finally, the target posts a Completion Queue Entry (CQE) to indicate that the operation has succeeded.

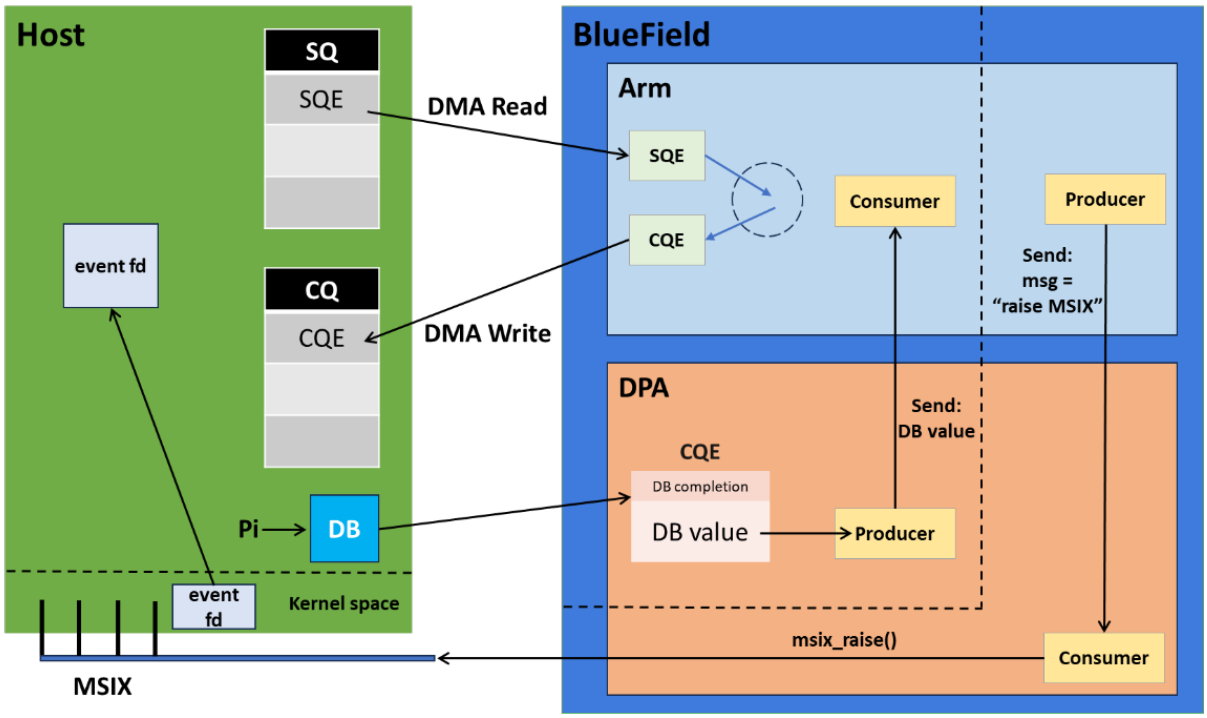
4. Destroying an I/O Submission Queue – **SPDK_NVME_OPC_DELETE_IO_SQ** – **handle_delete_io_sq()** :
    a. The process starts by fetching the identifier of the queue that needs to be deleted from the NVMe request.
    b. Once the identifier is retrieved, the corresponding **nvmf_doca_sq** entity is located.
    c. The associated poll group is then extracted from the I/O, as it is responsible for destroying the SQ associated with it. The thread retrieved schedules the execution of **nvmf_doca_pci_dev_poll_group_stop_io_sq()** using **spdk_thread_send_msg** .
    d. The **nvmf_doca_sq_stop()** function is called to initiate the stopping process.
    e. The stopping process begins by calling **spdk_nvmf_qpair_disconnect()** To disconnect an NVMe-oF queue pair (QP), clean up associated resources and terminate the connection.
    f. Once this step is complete, **nvmf_doca_sq_stop_continue()** is triggered to proceed with a sequence of asynchronous callback functions, ensuring that each step completes before moving to the next, performing the following actions:
        i. Disconnect an NVMe-oF queue pair (QP), cleaning up associated resources and terminate the connection.
        ii. Stop the DOCA devemu PCIe device doorbell to prevent triggering completions on the associated doorbell completion context.
        iii. Send unbind SQ doorbell message to DPA.
        iv. Stop the NVMe-oF DOCA DMA pool
        v. Stop the DMA context associated with the Submission Queue and frees all elements of the NVMe-oF DOCA Submission Queue.
        vi. Destroy the resources associated with the SQ: the DMA pool, the queue and the request pool.
        vii. Finally, the target posts a Submission Queue Entry (CQE) to indicate that the operation has succeeded.

6.14.2.6 Data Path Flows

From the host's perspective, it is communicating with a standard NVMe device. To create this experience, the DOCA transport uses the available NVMe-oF APIs to mimic the behavior of a real NVMe device effectively.

The data path flow involves a series of steps that handle the transfer and processing of commands between the host and the NVMe-oF target. It begins with the host writing commands to the Submission Queue (SQ) entries, specifying operations like read or write requests that the NVMe-oF target processes.

The diagram below provides a holistic view and data path steps:



1. The host writes the Submission Queue Entry (SQE).
2. The host rings the doorbell (DB), and the DB value is received by the DPA.
3. The producer forwards the DB value to the ARM processor.
4. The system reads the SQE from the host.
5. The SQE is processed.
6. A Completion Queue Entry (CQE) is written back.
7. If MSI-X is enabled, the producer triggers the MSI-X interrupt.
8. The DPU's consumer raises the MSI-X interrupt to the host.

Each step is described in depth below:

6.14.2.6.1 Retrieve Doorbell

The process starts by retrieving the doorbell values, which indicate new commands submitted by the host, allowing the system to identify and process pending commands:

The DPA wakes up and checks the reason for activation—it could either be that the DPU has posted something for the DPA consumer, or a new doorbell value needs to be passed to the DPU. In this case, the DPA detects a new doorbell value and sends it to the DPU via message queues. The DPU then calculates the number of Submission Queue Entries (SQEs) that need to be fetched from the host's submission queue and retrieves the commands using DMA via the **nvmf_doca_sq_update_pi()** function.

6.14.2.6.2 Fetch SQE From Host

After retrieving the Submission Queue Entry (SQE), the system must translate this command into a request object that the NVMe-oF target can understand and process. The SQE contains crucial command details that need to be executed, such as read or write operations. This process involves populating a **spdk_nvmf_request** structure, which includes:

- Command parameters extracted from the SQE.
- Associated data buffer locations (if any data is to be read from or written to).
- Metadata and additional information necessary for processing the command.

For admin commands, the SQEs are handled by the **nvmf_doca_on_fetch_sqe_complete()** function, while I/O NVMe-oF commands are managed by the **nvmf_doca_on_fetch_nvm_sqe_complete()** function. Both functions are responsible for filling the **nvmf_doca_request** structure.

A request is obtained from the SQ request pool, as described previously, and it is populated by setting various attributes based on the specific command being issued. These attributes may include the Namespace ID, the length of the request, and the queue to which the request belongs.

Next, there are three options for data direction handled in the command:

1. No data transfer:
    a. After preparing the request, the system sets the callback function to **post_cqe_from_response()**
    b. It then executes the request using **spdk_nvmf_request_exec()**
    c. Finally, the system posts the CQE to indicate completion
2. Data transfer from host to DPU:
    a. After preparing the request, the system retrieves buffers from the SQ pool and initializes them with the details of the data to be copied from the host
    b. It invokes **nvmf_doca_sq_copy_data()**, which performs a DMA copy from the host to the DPU
    c. Once the asynchronous copy completes, **spdk_nvmf_request_exec()** is called to continue processing
    d. Finally, the system posts the CQE to signal completion
3. Data transfer from DPU to host:
    a. After preparing the request, the system retrieves buffers from the SQ pool and initializes them with the data to be copied and the destination address on the host.

b. It invokes **nvmf_doca_sq_copy_data()** to perform a DMA copy from the DPU to the host
c. Once the asynchronous copy finishes, **spdk_nvmf_request_exec()** is called to complete processing
d. The system then posts the CQE to indicate the operation's completion

While the overall flow for NVMe commands and admin commands is similar, there are subtle differences in the transport implementation to address the unique requirements of each command type. For I/O commands like read and write, the system may involve large data blocks transfer. Here the PRPs (Physical Region Pages) come into play, as they are used to describe the memory locations for the data to be read or written. The PRPs provide a list of physical addresses that the NVMe device uses to access the host's memory directly.

In this scenario, there could be multiple DMA operations required for copying the data. Therefore, after preparing the request, the function **nvme_cmd_map_prps()** is invoked to iterate over the entire PRP list, preparing the retrieved buffers from the pool and initializing them with the corresponding data and destination addresses. Once the buffers are properly set up, the function **buffer_ready_copy_data_host_to_dpu()** is called, which iterates through all the buffers and invokes **nvmf_doca_sq_copy_data()** for each one. Only after all asynchronous copy tasks for the buffers are completed does the function **nvmf_doca_request_complete()** get called to signal the end of the request processing.

6.14.2.6.3 Dispatch Command to NVMe-oF Target

Once the request is built, the function **spdk_nvmf_request_exec()** is called to execute it. **spdk_nvmf_request_exec()** initiates the processing of the request by determining the command type and dispatching it to the appropriate handler within the NVMe-oF target for processing. The NVMe-oF target interprets the command, allocates necessary resources, and performs the requested operation.

When a request is dispatched to the NVMe-oF target for execution via **spdk_nvmf_request_exec()**, a completion callback function is usually configured. This callback is invoked once the request has been fully processed, indicating to the transport layer that the NVMe-oF target has completed handling the request.

6.14.2.6.4 Post CQE to Host

After the command has been processed, we create a Completion Queue Entry (CQE) and we posted back to the host to indicate the completion status of the command. This entry includes details about the operation's success or any errors encountered.

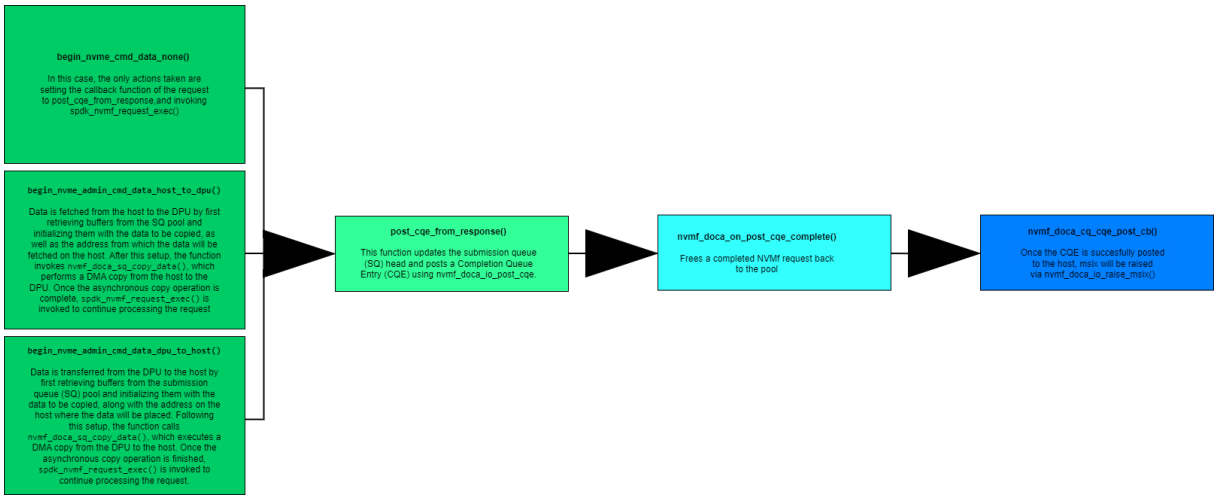
6.14.2.6.5 Raise RMSI-X

If the DMA for posting the CQE is completed successfully, MSIX is raised to the host, to inform the host that a completion event has occurred allowing to it read the CQE and process the result of the request.

In this scenario, the DPU calls **nvmf_doca_io_raise_msix()**, which in turn sends a message through **nvmf_doca_dpa_msgq_send()**. This action prompts the DPA to wake up and attempt to retrieve the

consumer completion context. Then, DPA receives a message from the DPU instructing it to raise MSIX.

The flow diagram illustrates the steps from fetching SQEs and preparing the request to posting the CQE, highlighting the three possible data scenarios (case where PRP is not involved):



6.14.2.7 Limitations

6.14.2.7.1 Supported SPDK Versions

The supported SPDK version is 23.01.

6.14.2.7.2 Supported Admin Commands

Currently, not all admin commands are supported. The transport supports the following commands: **SPDK_NVME_OPC_CREATE_IO_CQ**, **SPDK_NVME_OPC_DELETE_IO_CQ**, **SPDK_NVME_OPC_CREATE_IO_SQ**, **SPDK_NVME_OPC_DELETE_IO_SQ**, **SPDK_NVME_OPC_ASYNC_EVENT_REQUEST**, **SPDK_NVME_OPC_IDENTIFY**, **SPDK_NVME_OPC_GET_LOG_PAGE**, **SPDK_NVME_OPC_GET_FEATURES**, **SPDK_NVME_OPC_SET_FEATURES**.

6.14.2.7.3 Supported NVM Commands

Currently, the transport supports only the following NVMe commands: **SPDK_NVME_OPC_FLUSH**, **SPDK_NVME_OPC_WRITE**, **SPDK_NVME_OPC_READ**.

6.14.2.7.4 SPDK Stop Listen Flow

The stop listener flow (**spdk_nvmf_stop_listen**) can be initiated through the **remove_listener** RPC. The current version of SPDK has limitations regarding the asynchronous handling of **remove_listener** requests. For this reason, it is recommended that the stop listener function be

called only after freeing up resources like memory buffers and queue pairs. This can be accomplished on the host side by issuing the unbind script:

```
python3 samples/doca_devemu/devemu_pci_vfio_bind.py --unbind 0000:62:00.0
```

6.14.3 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Compress
- DOCA Comch

For additional information about the used DOCA libraries, please refer to the respective programming guides.

6.14.4 Dependencies

- BlueField-3 DPU is required
- SPDK version 23.0 is required

6.14.5 Compiling the Application

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for the application is located in the following directory:

```
/opt/mellanox/doca/applications/nvme_emulation/
```

6.14.5.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

1. To build all the applications together, run:

   ```
   cd /opt/mellanox/doca/applications/
   meson /tmp/build
   ninja -C /tmp/build
   ```

   doca_nvme_emulation is created under **/tmp/build/applications/nvme_emulation/**

2. Alternatively, one can set the desired flags in the **meson_options.txt** file instead of providing them in the compilation command line:

a. Edit the following flags in **/opt/mellanox/doca/applications/meson_options.txt**:
   - Set **enable_all_applications** to **false**
   - Set **enable_nvme_emulation** to **true**

b. The same compilation commands should be used, as were shown in the previous section:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

6.14.5.2 Compiling With Custom SPDK

If you plan to use a custom or alternative SPDK version, update the paths in the following variables via Meson:

- **spdk_lib_path**
- **spdk_incl_path**
- **spdk_dpdk_lib_path**
- **spdk_isal_prefix**

6.14.6 Running the Application

6.14.6.1 Prerequisites

- From the server on the DPU – the user is required to allocate hugepages, and then run the application that runs continuously, remaining active and processing incoming RPC requests.

```
$ echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages $ sudo mount -t hugetlbfs -o
pagesize=2M nodev /mnt/huge
$ sudo /tmp/build/applications/nvme_emulation/doca_nvme_emulation
```

- From the client on the DPU – the user can send various RPC requests to the application during its execution. For example, to remove a listener, the user could send the following command:

```
$ sudo PYTHONPATH=/doca/applications/nvme_emulation/ /usr/bin/spdk_rpc.py nvmf_subsystem_remove_listener
nqn.2016-06.io.spdk:cnode1 -t doca -a MT2306XZ00AYGES1D0F0
```

6.14.6.2 Application Execution

The NvMR emulation application is provided in source form, hence a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_nvme_application [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
```

```
  --sdk-log-level                  Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                     Filter logs from specific modules, separated by comma
  -j, --json <path>                Parse command line flags from an input json file

Program Flags:
  -p, --pci-addr                   DOCA Comm Channel device PCIe address
  -r, --rep-pci                    DOCA Comm Channel device representor PCIe address
  -f, --file                       File to send by the client / File to write by the server
  -t, --timeout                    Application timeout for receiving file content messages, default is 5
 sec

 -c, --config <config>     JSON config file (default none)
     --json <config>       JSON config file (default none)
     --json-ignore-init-errors
                           don't exit on invalid config entry
 -d, --limit-coredump      do not set max coredump size to RLIM_INFINITY
 -g, --single-file-segments
                           force creating just one hugetlbfs file
 -h, --help                show this usage
 -i, --shm-id <id>         shared memory ID (optional)
 -m, --cpumask <mask or list>    core mask (like 0xF) or core list of '[]' embraced (like [0,1,10]) for
 DPDK
 -n, --mem-channels <num>  channel number of memory channels used for DPDK
 -p, --main-core <id>      main (primary) core for DPDK
 -r, --rpc-socket <path>   RPC listen address (default /var/tmp/spdk.sock)
 -s, --mem-size <size>     memory size in MB for DPDK (default: 0MB)
     --disable-cpumask-locks    Disable CPU core lock files.
     --silence-noticelog   disable notice level logging to stderr
     --msg-mempool-size <size>  global message memory pool size in count (default: 262143)
 -u, --no-pci              disable PCIe access
     --wait-for-rpc        wait for RPCs to initialize subsystems
     --max-delay <num>     maximum reactor delay (in microseconds)
 -B, --pci-blocked <bdf>
                           PCIe addr to block (can be used more than once)
 -R, --huge-unlink         unlink huge files after initialization
 -v, --version             print SPDK version
 -A, --pci-allowed <bdf>
                           PCIe addr to allow (-B and -A cannot be used at the same time)
     --huge-dir <path>     use a specific hugetlbfs mount to reserve memory from
     --iova-mode <pa/va>   set IOVA mode ('pa' for IOVA_PA and 'va' for IOVA_VA)
     --base-virtaddr <addr>      the base virtual address for DPDK (default: 0x200000000000)
     --num-trace-entries <num>   number of trace entries for each core, must be power of 2, setting 0 to
disable trace (default 32768)
     --rpcs-allowed    comma-separated list of permitted RPCS
     --env-context         Opaque context for use of the env implementation
     --vfio-vf-token       VF token (UUID) shared between SR-IOV PF and VFs for vfio_pci driver
 -L, --logflag <flag>    enable log flag (all, accel, aio, app_config, app_rpc, bdev, bdev_concat,
bdev_ftl, bdev_group, bdev_malloc, bdev_null, bdev_nvme, bdev_raid, bdev_raid0, bdev_raid1, bdev_raid5f,
blob, blob_esnap, blob_rw, blobfs, blobfs_bdev, blobfs_bdev_rpc, blobfs_rw, ftl_core, ftl_init, gpt_parse,
json_util, log, log_rpc, lvol, lvol_rpc, notify_rpc, nvme, nvme_vfio, nvmf, nvmf_tcp, opal, rdma, reactor,
rpc, rpc_client, sock, sock_posix, thread, trace, uring, vbdev_delay, vbdev_gpt, vbdev_lvol, vbdev_opal,
vbdev_passthru, vbdev_split, vbdev_zone_block, vfio_pci, vfio_user, virtio, virtio_blk, virtio_dev,
virtio_pci, virtio_user, virtio_vfio_user, vmd)
 -e, --tpoint-group <group-name>[:<tpoint_mask>]
                           group_name - tracepoint group name for spdk trace buffers (bdev, nvmf_rdma,
nvmf_tcp, blobfs, thread, nvme_pcie, nvme_tcp, bdev_nvme, nvme_nvda_tcp, all)
                           tpoint_mask - tracepoint mask for enabling individual tpoints inside a
tracepoint group. First tpoint inside a group can be enabled by setting tpoint_mask to 1 (e.g. bdev:0x1).
                            Groups and masks can be combined (e.g. thread,bdev:0x1).
                            All available tpoints can be found in /include/spdk_internal/trace_defs.h
```

The above usage printout can be printed to the command line using the **-h** (or **--help**) options:

```
/tmp/build/applications/nvme_emulation/doca_nvme_emulation -h
```

6.14.6.3 Command Line Flags

The application uses the same command-line flags as SPDK, allowing for configuration and behavior control similar to standard SPDK applications.

For more details refer to SPDK's official Storage Performance Development Kit Documentation.

6.14.6.4 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.14.6.5 References

- **/opt/mellanox/doca/applications/nvme_emulation/**
- **/opt/mellanox/doca/applications/nvme_emulation/build_device_code.sh**
- **/opt/mellanox/doca/applications/nvme_emulation/dependencies**
- **/opt/mellanox/doca/applications/nvme_emulation/device**
- **/opt/mellanox/doca/applications/nvme_emulation/host**
- **/opt/mellanox/doca/applications/nvme_emulation/meson.build**
- **/opt/mellanox/doca/applications/nvme_emulation/nvme_emulation.c**
- **/opt/mellanox/doca/applications/nvme_emulation/rpc_nvmf_doca.py**

6.15 DOCA PCC Application Guide

This document provides a DOCA PCC implementation on top of NVIDIA® BlueField® networking platform.

6.15.1 Introduction

Programmable Congestion Control (PCC) allows users to design and implement their own congestion control (CC) algorithm, giving them the flexibility to work out an optimal solution to handle congestion in their clusters. On BlueField-3 networking platforms, PCC is provided as a component of DOCA.

The application leverages the DOCA PCC API to provide users the flexibility to manage allocation of DPA resources according to their requirements.

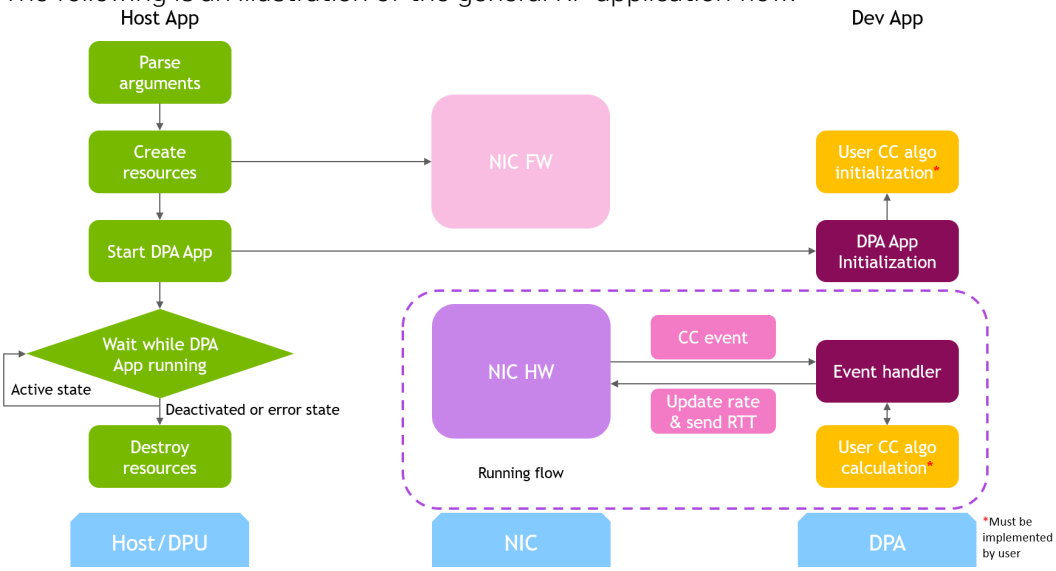
Typical DOCA application includes App running on host/Arm and App running on DPA. Developers are advised to use the host/Arm application with minimal changes and focus on developing their algorithm and integrating it into the DPA application.

6.15.2 System Design

DOCA PCC application consists of two parts:

- Host/Arm app is the control plane. It is responsible for allocating all resources and handover to the DPA app initially, then destroying everything when the DPA app finishes its operation. The host app must always be alive to stay in control while the device app is working.
- Device/DPA app is the data plane.
    - The default mode of the data plane is running as a reaction point (RP). When the first thread is activated, DPA App initialization is done in the DOCA PCC library by calling the algorithm initialization function implemented by the user in the app. Moreover, the user algorithm execution function is called when a CC event arrives. The user algorithm takes event data as input and performs a calculation, using per-flow context, and replies with the updated rate value and a flag to send an RTT request.

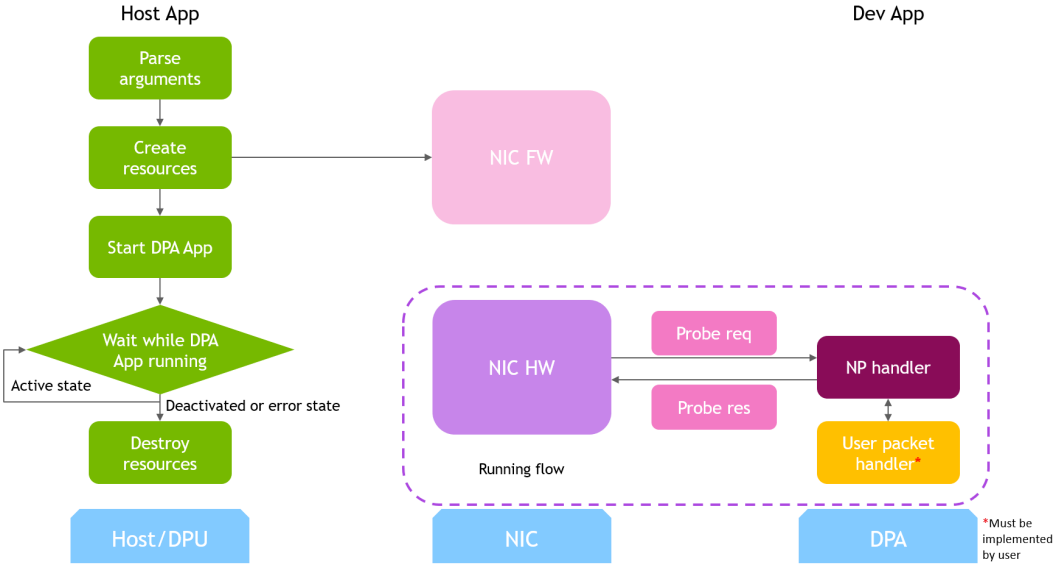
The following is an illustration of the general RP application flow:



The host/Arm application sends a command to the BlueField platform firmware when allocating or destroying resources. CC events are generated by the BlueField platform hardware automatically when sending data or receiving ACK/NACK/CNP/RTT packets, then the device application handles these events by calling the user algorithm. After the DPA application replies to hardware, handling of current event is done, and the next event can arrive.

> The device/DPA app can also run different algorithms for the RP program, which users can configure as a runtime option.

- The device/DPA app can function as a notification point (NP). When a new probe request packet arrives, the user handler can read and analyze the data and send a probe response back. The following is an illustration of the general NP application flow:



> The device/DPA app is as well capable of functioning as a telemetry program for a NP switch operation, which users can configure as a runtime option.

6.15.3 Application Architecture

```
/opt/mellanox/doca/applications/pcc/
    host
         pcc.c
         pcc_core.c
         pcc_core.h
    device
         pcc_common_dev.h
         rp
              rtt_template
                        algo
                             rtt_template.h
                             rtt_template_algo_params.h
                             rtt_template_ctxt.h
                             rtt_template.c
                        rp_rtt_template_dev_main.c
              switch_telemetry
                        algo
                             telem_template.h
                             telem_template_algo_params.h
                             telem_template_ctxt.h
                             telem_template.c
                        rp_switch_telemetry_dev_main.c
         np
             switch_telemetry
                        np_switch_telemetry_dev_main.c
```

The main content of the reference DOCA PCC application files are the following:

- **host/pcc.c** – entry point to entire application
- **host/pcc_core.c** – host functions to initialize and destroy the PCC application resources, parsers for PCC command line parameters
- **device/pcc_common_dev.h** – common util calls and definitions for device programs
- **device/rp/rtt_template/rp_rtt_template_dev_main.c** – callbacks for user CC algorithm initialization, user CC algorithm calculation and algorithm parameter change notification of the RTT template algorithm reference
- **device/rp/rtt_template/algo/*** – user CC algorithm reference for RTT template. Put user algorithm code here
- **device/rp/switch_telemetry/rp_switch_telemetry_dev_main.c** – callbacks for user CC algorithm initialization, user CC algorithm calculation, and algorithm parameter change notification of the switch telemetry algorithm reference
- **device/rp/switch_telemetry/algo/*** – user CC algorithm reference for switch telemetry. Put user algorithm code here.
- **device/np/switch_telemetry/np_switch_telemetry_dev_main.c** – callback for user NP handling, implemented as a switch telemetry program to observe last hop switch metadata

6.15.4 DOCA Libraries

This application leverages the following DOCA library:

- DOCA PCC

Refer to its respective programming guide for more information.

6.15.5 Dependencies

- NVIDIA BlueField-3 Platform is required
- Firmware 32.38.1000 and higher
- MFT 4.25 and higher

6.15.6 Compiling the Application

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for the application is located in the following directory:

```
/opt/mellanox/doca/applications/pcc/
```

6.15.6.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_pcc is created under **/tmp/build/pcc/**.

6.15.6.2 Compiling Only the Current Application

To directly build only the PCC application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_pcc=true
ninja -C /tmp/build
```

doca_pcc is created under **/tmp/build/pcc/**.

Alternatively, one can set the desired flags in the **meson_options.txt** file instead of providing them in the compilation command line:

1. Edit the following flags in **/opt/mellanox/doca/applications/meson_options.txt**:
   - Set **enable_all_applications** to **false**
   - Set **enable_pcc** to **true**
2. Run the following compilation commands:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

> **doca_pcc** is created under **/tmp/build/pcc/**.

6.15.6.3 Compilation Options

The application offers specific compilation flags which one can set for a desired behavior in the device/DPA program.

In the **meson_options.txt** file, one can find the following options:

- **enable_pcc_application_tx_counter_sampling**: set to **true** to use TX counters sampled at runtime in the RP CC handling algorithm.

6.15.7 Running the Application

6.15.7.1 Prerequisites

Enable **USER_PROGRAMMABLE_CC** in **mlxconfig**:

```
mlxconfig -y -d /dev/mst/mt41692_pciconf0 set USER_PROGRAMMABLE_CC=1
```

Perform a BlueField system reboot for the **mlxconfig** settings to take effect.

6.15.7.2 Application Execution

The PCC application is provided in source form. Therefore, a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_pcc [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -d, --device <RDMA device names> RDMA device name that supports PCC (mandatory).
```

```
  -rp-st, --rp-switch-telemetry <PCC Reaction Point Switch Telemetry> Flag to indicate running as a
Reaction Point Switch Telemetry (optional). The application will generate IFA2 probe packets. By default
 the flag is set to false.
  -np-st, --np-switch-telemetry <PCC Notification Point Switch Telemetry> Flag to indicate running as a
Notification Point Switch Telemetry (optional). The application will generate IFA2 probe packets. By
default the flag is set to false.
  -t, --threads <PCC threads list>  A list of the PCC threads numbers to be chosen for the DOCA PCC context
to run on (optional). Must be provided as a string, such that the number are separated by a space.
  -w, --wait-time <PCC wait time>   The duration of the DOCA PCC wait (optional), can provide negative
values which means infinity. If not provided then -1 will be chosen.
  -r-handler, --remote-sw-handler <CCMAD remote SW handler> CCMAD remote SW handler flag (optional). If not
provided then false will be chosen.
  -hl, --hop-limit <IFA2 hop limit> The IFA2 probe packet hop limit (optional). If not provided then 0XFE
 will be chosen.
  -gns, --global-namespace <IFA2 global namespace> The IFA2 probe packet global namespace (optional). If
not provided then 0XF will be chosen.
  -gns-ignore-mask, --global-namespace-ignore-mask <IFA2 global namespace ignore mask> The IFA2 probe
packet global namespace ignore mask (optional). If not provided then 0 will be chosen.
  -gns-ignore-val, --global-namespace-ignore-value <IFA2 global namespace ignore value> The IFA2 probe
packet global namespace ignore value (optional). If not provided then 0 will be chosen.
  -f, --coredump-file <PCC coredump file> A pathname to the file to write coredump data in case of
unrecoverable error on the device (optional). Must be provided as a string.
 --dpa-resources <DPA resources file> Path to a DPA resources .yaml file (optional). Must be provided
together with DPA application key.
 --dpa-app-key <DPA application key> Application key in specified DPA resources .yaml file (optional). Must
be provided together with DPA resources file.
  -tlm-ex, --enable-telemetry-export <Telemetry exporter enablement> Flag to enable exporting telemetry
data gathered by pcc algorithem (optional). By default the flag is set to false.
```

This usage printout can be printed to the command line using the **-h** (or **--help**) options:

```
./doca_pcc -h
```

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the BlueField Platform or the host:

```
./doca_pcc -d mlx5_0
```

The RDMA device identifier (**mlx5_0**) should match the identifier of the desired RDMA device.

6.15.7.3 Command Line Flags

6.15.7.3.1 General Flags

Short Flag	Long Flag	Description
-h	**--help**	Prints a help synopsis and exits
-v	**--version**	Prints program version information and exits

Short Flag	Long Flag	Description
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.15.7.3.2 Program Flags

Short Flag	Long Flag	Description
`d`	`device`	RDMA device name that supports PCC
`rp-st`	`rp-switch-telemetry`	(Optional) Flag to indicate running as a RP switch telemetry. The DOCA PCC application can run as a RP switch telemetry program. If this flag is used, the application loads a program to run on the DPA of a switch telemetry algorithm which receives metadata from the last hop switch congestion point from the NP node.
`np-st`	`np-switch-telemetry`	(Optional) Flag to indicate running as a NP switch telemetry. The DOCA PCC application can run as a NP switch telemetry program. If this flag is used, the application loads a program to run on the DPA to sample metadata from the last hop switch congestion point and send them in response packet.
`t`	`threads`	(Optional) A list of the PCC EU indexes to be chosen for the DOCA PCC event handler threads to run on. Must be provided as a string, such that the numbers are separated by a space. The placement of the PCC threads per core can be controlled using the EU indexes. Utilizing a large number of EUs, while limiting the number of threads per core, gives the best event handling rate and lowest event latency. The last EU is used for communication with the BlueField Platform while all others are for data path CC event handling.
`w`	`wait-time`	(Optional) In seconds, the duration of the DOCA PCC wait. Negative values mean infinity.

Short Flag	Long Flag	Description
`r-handler`	`remote-sw-handler`	(Optional) CCMAD remote SW handler flag. Relevant for RP contexts. This flag indicates whether the expected CCMAD probe packet responses are generated by a remote DOCA NP process or not. ⚠ If using other probe types than CCMAD, probe packet responses are always expected to be generated from a remote DOCA NP process.
`hl`	`hop-limit`	(Optional) The IFA2 probe packet hop limit ℹ Relevant for RP contexts.
`gns`	`global-namespace`	(Optional) The IFA2 probe packet global namespace ℹ Relevant for RP contexts.
`gns-ignore-mask`	`global-namespace-ignore-mask`	(Optional) The IFA2 probe packet global namespace ignore mask ℹ Relevant for NP contexts.
`gns-ignore-val`	`global-namespace-ignore-value`	(Optional) The IFA2 probe packet global namespace ignore value ℹ Relevant for NP contexts.
`f`	`coredump-file`	(Optional) A pathname to the file to write core dump data if an unrecoverable error occurs on the device
N/A	`dpa-resources`	(Optional) Path to a DPA resources `.yaml` file. Must be provided together with DPA application key.
N/A	`dpa-app-key`	(Optional) Application key in specified DPA resources `.yaml` file. Must be provided together with DPA resources file.

6.15.7.4 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.15.8 Application Code Flow

This section lists the application's configuration flow, explaining the different DOCA function calls and wrappers.

1. Parse application argument.
    a. Initialize arg parser resources and register DOCA general parameters.

```
doca_argp_init();
```

b. Register PCC application parameters.

```
register_pcc_params();
```

c. Parse the arguments.

```
doca_argp_start();
```

i. Parse DOCA flags.

ii. Parse DOCA PCC parameters.

2. PCC initialization.

```
pcc_init();
```

a. Open DOCA device that supports PCC.

b. Create DOCA PCC context.

c. Configure affinity of threads handling CC events.

3. Start DOCA PCC.

```
doca_pcc_start();
```

a. Create PCC process and other resources.

b. Trigger initialization of PCC on device.

c. Register the PCC in the BlueField Platform hardware so CC events can be generated and an event handler can be triggered.

4. Process state monitor loop.

```
doca_pcc_get_process_state();
doca_pcc_wait();
```

a. Get the state of the process:

State	Description
DOCA_PCC_PS_ACTIVE = 0	The process handles CC events (only one process is active at a given time)
DOCA_PCC_PS_STANDBY = 1	The process is in standby mode (another process is already **ACTIVE**)
DOCA_PCC_PS_DEACTIVATED = 2	The process has been deactivated by the BlueField Platform firmware and should be destroyed
DOCA_PCC_PS_ERROR = 3	The process is in error state and should be destroyed

b. Wait on process events from the device.

5. PCC destroy.

```
doca_pcc_destroy();
```

a. Destroy PCC resources. The process stops handling PCC events.

b. Close DOCA device.

6. Arg parser destroy.

```
doca_argp_destroy()
```

6.15.9 Port Programmable Congestion Control Register

The Port Programmable Congestion Control (PPCC) register allows the user to configure and read PCC algorithms and their parameters/counters.

It supports the following functionalities:

- Enabling different algorithms on different ports
- Querying information of both algorithms and tunable parameters/counters
- Changing algorithm parameters without compiling and reburning user image
- Querying or clearing programmable counters

6.15.9.1 Usage

The PPCC register can be accessed using a string similar to the following:

```
sudo mlxreg -d /dev/mst/mt41692_pciconf0 -y --get --op "cmd_type=0" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=0,algo_param_index=0,target_app=0"
sudo mlxreg -d /dev/mst/mt41692_pciconf0 -y --set "cmd_type=1" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=0,algo_param_index=0,target_app=0"
```

Where you must:

- Set the **cmd_type** and the indexes
- Give values for **algo_slot** , **algo_param_index**
- Keep **local_port=1** , **pnat=0** , **lp_msb=0**
- Set **target_app = 0** for PCC
- Keep **doca_pcc** application running

Command (**cmd_type**)	Description	Method	Index	Input (in **--set**)	Output
0x0	Get algorithm info	Get	**algo_slot**	N/A	**value** : 32-bit **algo_num** (or 0 if no algorithm is available at this index). **text** : Algorithm name and description (ASCII). **sl_bitmask_support** : Indicates if the device supports **sl_bitmask** logic.

Command (`cmd_type`)	Description	Method	Index	Input (in `--set`)	Output
`0x1`	Enable algorithm	Set	`algo_slot`	`sl_bitmask`, `trace_en`, `counter_en`. Optional: • `cfg_counter_group_en=1` • `counter_group_id` (assigns counter group to algo slot).	N/A
`0x2`	Disable algorithm	Set	`algo_slot`	N/A	N/A
`0x3`	Get algorithm enabling status	Get	`algo_slot`	N/A	`value`: 0 (disabled) or 1 (enabled). `sl_bitmask`: Applies to specific SLs based on the bitmask. `sl_bitmask_support`: Indicates if the device supports `sl_bitmask` logic.
`0x4`	Get number of parameters	Get	`algo_slot`	N/A	`value`: Number of parameters for the algorithm.
`0x5`	Get parameter info	Get	`algo_slot` `algo_param_index`	N/A	`param_value1`: Default value. `param_value2`: Minimum value. `param_value3`: Maximum value. `prm`: 0 (RO), 1 (RW), or 2 (RO + clearable). `text`: Parameter name and description.
`0x6`	Get parameter value	Get	`algo_slot` `algo_param_index`	N/A	`value`: Parameter value.
`0x7`	Get and clear parameter	Get	`algo_slot` `algo_param_index`	N/A	`value`: Parameter value (then cleared).

Command (`cmd_type`)	Description	Method	Index	Input (in `--set`)	Output
`0x8`	Set parameter value	Set	`algo_slot` `algo_param_index`	`value` : Parameter value. Silently ignored if outside min..max range or if param is not RW.	N/A
`0xA`	Bulk get parameters	Get	`algo_slot`	N/A	`text_length` : `param_num` × 4 bytes. `text[0]...text[n]` : Parameter values.
`0xB`	Bulk set parameters	Set	`algo_slot`	`text_length` : `param_num` × 4. `text[0]...text[n]` : Parameter values.	N/A
`0xC`	Bulk get counters	Get	`algo_slot`	Optional: • `cfg_counter_group_en=1` • `counter_group_id` – specifies which counter group to query. If `cfg_counter_group_en=0` , uses group from algo slot header.	`text_length` : `counter_num` × 4 bytes. `text[0]...text[n]` : Counter values.
`0xD`	Bulk get and clear counters	Get	`algo_slot`	Optional: `cfg_counter_group_en=1` `counter_group_id` (same as 0xC).	`text_length` : `counter_num` × 4 bytes. `text[0]...text[n]` : Counter values (then cleared).
`0xE`	Get number of counters	Get	`algo_slot`	N/A	`value` : Number of counters. `param_value1` : Maximum counter group ID (0-indexed, max = `num_groups` -1).
`0xF`	Get counter info	Get	`algo_slot` `algo_param_index`	N/A	`param_value3` : Max counter value (wraps to 0). `prm` : 0 (RO), 1 (RW), or 2 (RO + clearable). `text` : Counter name and description.

Command (`cmd_type`)	Description	Method	Index	Input (in `--set`)	Output
`0x10`	Get algorithm info array	Get	N/A	N/A	`text_length` : `num_slots` × 4 bytes. `text[0]...text[n]` : `algo_id[31:16]` , `major_ver[15:8]` , `minor_ver[7:0]` (`0` = empty slot).
`0x12`	Get histogram IDs for algo slot	Get	`algo_slot` (algorithm slot number)	N/A	`value` : Number of histogram IDs. `text[0]...text[n]` : 32-bit histogram ID values.
`0x13`	Histogram get description	Get	`algo_slot` = histogram ID	N/A	`text` : Null-terminated ASCII description (up to 220 bytes).
`0x14`	Histogram get	Get	`algo_slot` = histogram ID	N/A	`algo_param_index` : Mode (0=LINEAR, 1=EXPONENTIAL). `value` : `num_edges` (2–33, i.e., 1–32 bins). `param_value1` : Enable status (0=disabled, 1=enabled). `counter_group_id` : Assigned counter group (0–14). `text[0]...text[n]` : Edge values array (32-bit each).
`0x15`	Histogram set	Set	`algo_slot` = histogram ID `algo_param_index` = mode (0=LINEAR, 1=EXP)	`value` : • 0 = disable histogram • 1 = enable (no param change) • 2–33 = `num_edges` (set & enable). `counter_group_id` : Assigns counter group. `text[0]...text[n]` : Edge values (when value ≥2).	N/A

6.15.9.2 Internal Default Algorithm

The internal default algorithm is used when enhanced connection establishment (ECE) negotiation fails. It is mainly used for backward compatibility and can be disabled using "force mode". Otherwise, users may change **doca_pcc_dev_user_algo()** in the device app to run a specific algorithm without considering the algorithm negotiation.

The force mode command is per port:

```
sudo mlxreg -d /dev/mst/mt41692_pciconf0 -y --get --op "cmd_type=2" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=15,algo_param_index=0,target_app=0"
sudo mlxreg -d /dev/mst/mt41692_pciconf0.1 -y --get --op "cmd_type=2" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=15,algo_param_index=0,target_app=0"
```

6.15.9.3 Counters

Each port supports multiple counter groups. The total number of available counter groups is returned by executing **cmd_type=0xE**, which populates the **param_value1** field with the maximum counter group ID (0-indexed).

6.15.9.3.1 Enabling Counters

Counters are enabled per algorithm slot using **cmd_type=0x1** (enable algorithm) and setting **counter_en=1**. Optionally, you can assign a specific counter group to the algorithm slot by setting **cfg_counter_group_en=1** and providing a **counter_group_id**.

If **cfg_counter_group_en=0** (or is omitted), the algorithm code sets the counter group ID, otherwise the default group (Group 0) is used.

To enable the algorithm with counters on the default counter group (Group 0), execute:

```
sudo mlxreg -d /dev/mst/mt41692_pciconf0 -y --set "cmd_type=1,counter_en=1" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=0,algo_param_index=0,target_app=0"
```

To enable the algorithm with counters on a specific counter group (e.g., Group 2), execute:

```
sudo mlxreg -d /dev/mst/mt41692_pciconf0 -y --set
"cmd_type=1,counter_en=1,cfg_counter_group_en=1,counter_group_id=2" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=0,algo_param_index=0,target_app=0"
```

6.15.9.3.2 Querying Counters

After counters are enabled on an algorithm slot, they can be queried using **cmd_type=0xC** (bulk get counters) or **cmd_type=0xD** (bulk get and clear counters).

To execute a bulk get of counters on the default group:

```
sudo mlxreg -d /dev/mst/mt41692_pciconf0 -y --get --op "cmd_type=0xC" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=0,algo_param_index=0,target_app=0"
```

To query a specific counter group, append **cfg_counter_group_en=1** and provide the desired **counter_group_id**. For example, to query Group 2:

```
sudo mlxreg -d /dev/mst/mt41692_pciconf0 -y --get --op "cmd_type=0xC,cfg_counter_group_en=1,counter_group_id=2" --
reg_name PPCC --indexes "local_port=1,pnat=0,lp_msb=0,algo_slot=0,algo_param_index=0,target_app=0"
```

To execute a bulk get and clear the counters, use **cmd_type=0xD**:

```
sudo mlxreg -d /dev/mst/mt41692_pciconf0 -y --get --op "cmd_type=0xD" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=0,algo_param_index=0,target_app=0"
```

6.15.9.3.3 Getting Counter Information

Use **cmd_type=0xE** to query the total number of counters and the maximum available counter group ID:

```
sudo mlxreg -d /dev/mst/mt41692_pciconf0 -y --get --op "cmd_type=0xE" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=0,algo_param_index=0,target_app=0"
```

- **value**: The number of counters the algorithm possesses.
- **param_value1**: The maximum counter group ID (0-indexed).

Use **cmd_type=0xF** to retrieve information about an individual counter:

```
sudo mlxreg -d /dev/mst/mt41692_pciconf0 -y --get --op "cmd_type=0xF" --reg_name PPCC --indexes
"local_port=1,pnat=0,lp_msb=0,algo_slot=0,algo_param_index=0,target_app=0"
```

- **param_value3**: The maximum counter value (wraps to 0 when exceeded).
- **prm**: The permissions flag (0 = read-only, 1 = read-write, 2 = read-only + clearable).
- **text**: The ASCII counter name and description.

6.15.10 References

- **/opt/mellanox/doca/applications/pcc/**

6.16 DOCA PSP Gateway Application Guide

This document describes the usage of the NVIDIA DOCA PSP Gateway sample application on top of an NVIDIA® BlueField® networking platform or NVIDIA® ConnectX® SmartNIC.

6.16.1 Introduction

> ⚠ DOCA PSP Gateway is supported only on BlueField-3 or ConnectX-7 and later.

The DOCA PSP Gateway application leverages the BlueField or ConnectX hardware capability for fully offloaded secure network communication using the PSP security protocol. The application demonstrates how to exchange keys between application instances and insert rules controlling PSP encryption and decryption using the DOCA Flow library.

The application exchanges keys using an unencrypted gRPC channel. If your environment requires the protection of encryption keys, you must modify the application to create the gRPC channel using the applicable certificates.

The PSP Gateway application supports only the PSP tunnel protocol. The PSP transport protocol is not supported by the application in this release, although it is supported by the underlying DOCA Flow library.

The PSP Gateway application supports only IPv4 inner and IPv6 outer headers. Other combinations are not supported by the application in the current release, although they are supported by the underlying DOCA Flow library.

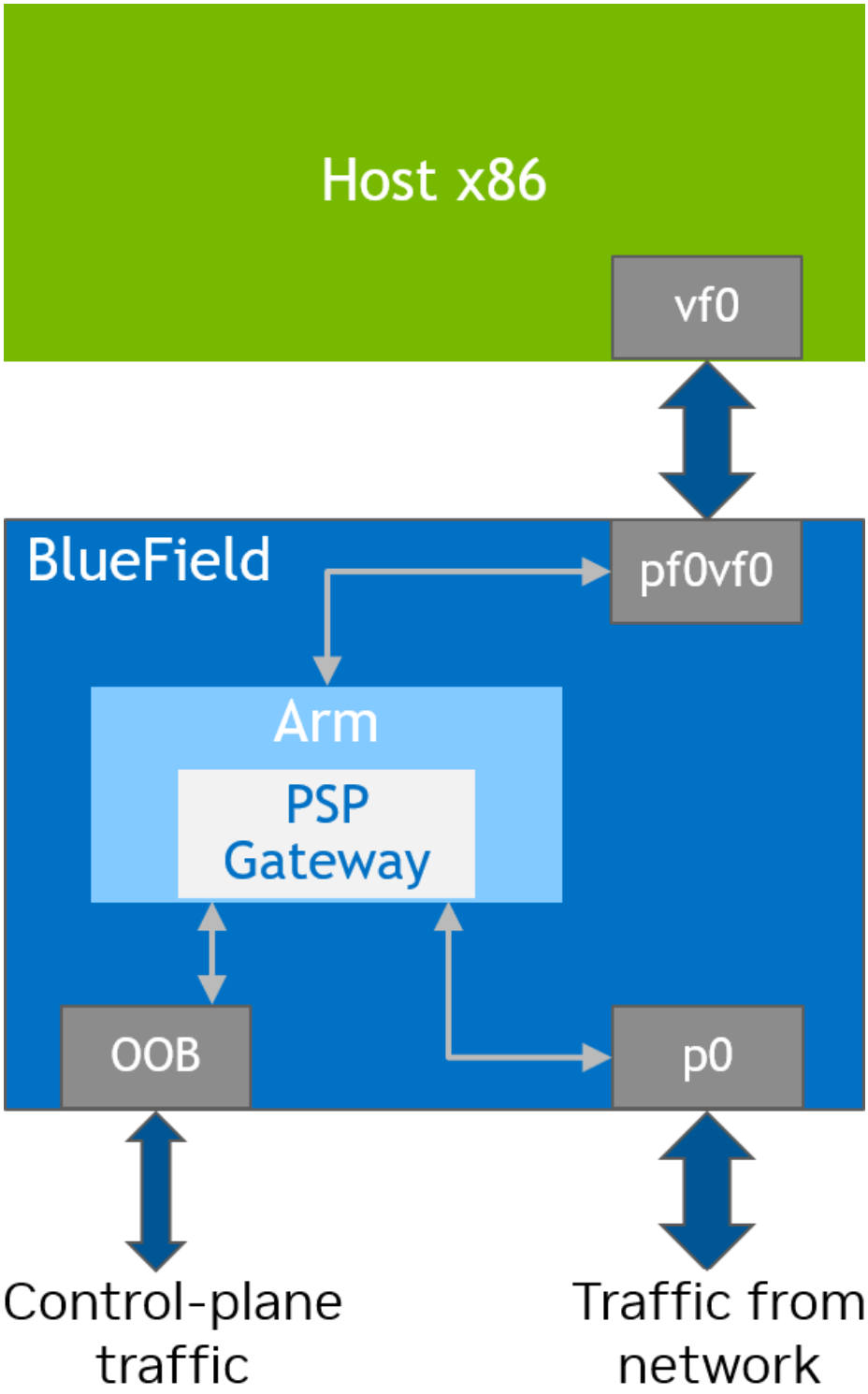
The application can be configured to establish out-bound PSP tunnel connections via individual command-line arguments, or via a text file configured via a command-line argument. The connections are established on-demand by default, but can also be configured to connect at startup.

6.16.2 System Design

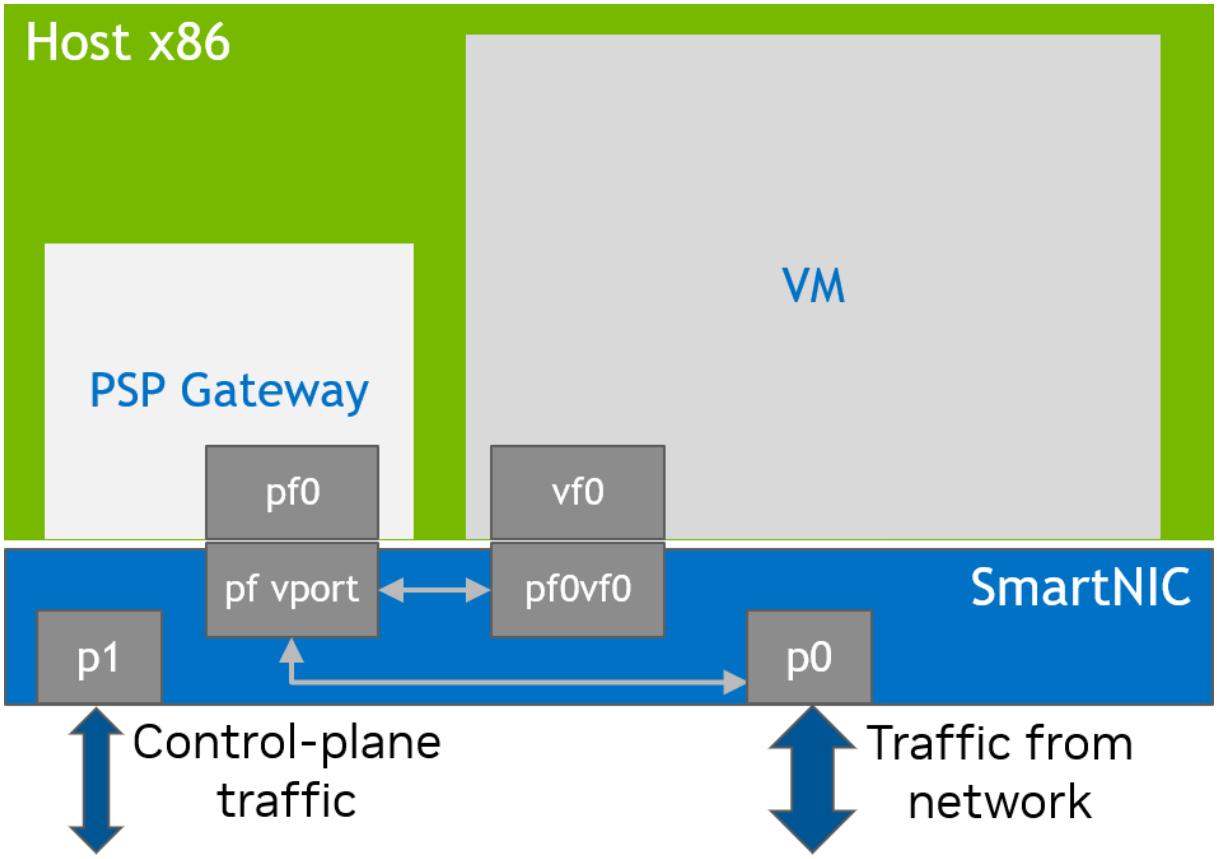
The DOCA PSP Gateway is designed to run with three ports:

- A secure (encrypted) uplink netdev (i.e., `p0`)
- An unsecured (plaintext) netdev representor (VF or SF)
- An out-of-bound (OOB) management port, used to communicate with peer instances using standard sockets

Whether the DOCA PSP Gateway is deployed to a BlueField or a ConnectX device, the functionality is the same. The Out of Bounds (OOB) network device carries PSP parameters between peers, the Uplink port carries secure (encrypted) traffic, and the VF carries the unencrypted traffic.



When the application is deployed to a DPU, the operation of the PSP encryption protocol is entirely transparent to the Host. All the resources required to manage the PSP connections are physically located on the DPU.



When the application is deployed to the host, the operation of the PSP encryption protocol is the responsibility of the host, and resources are allocated from the host. However, the operation of the PSP encryption protocol is entirely transparent to any virtual machines and containers attached to the VF network devices.

6.16.3 Application Architecture

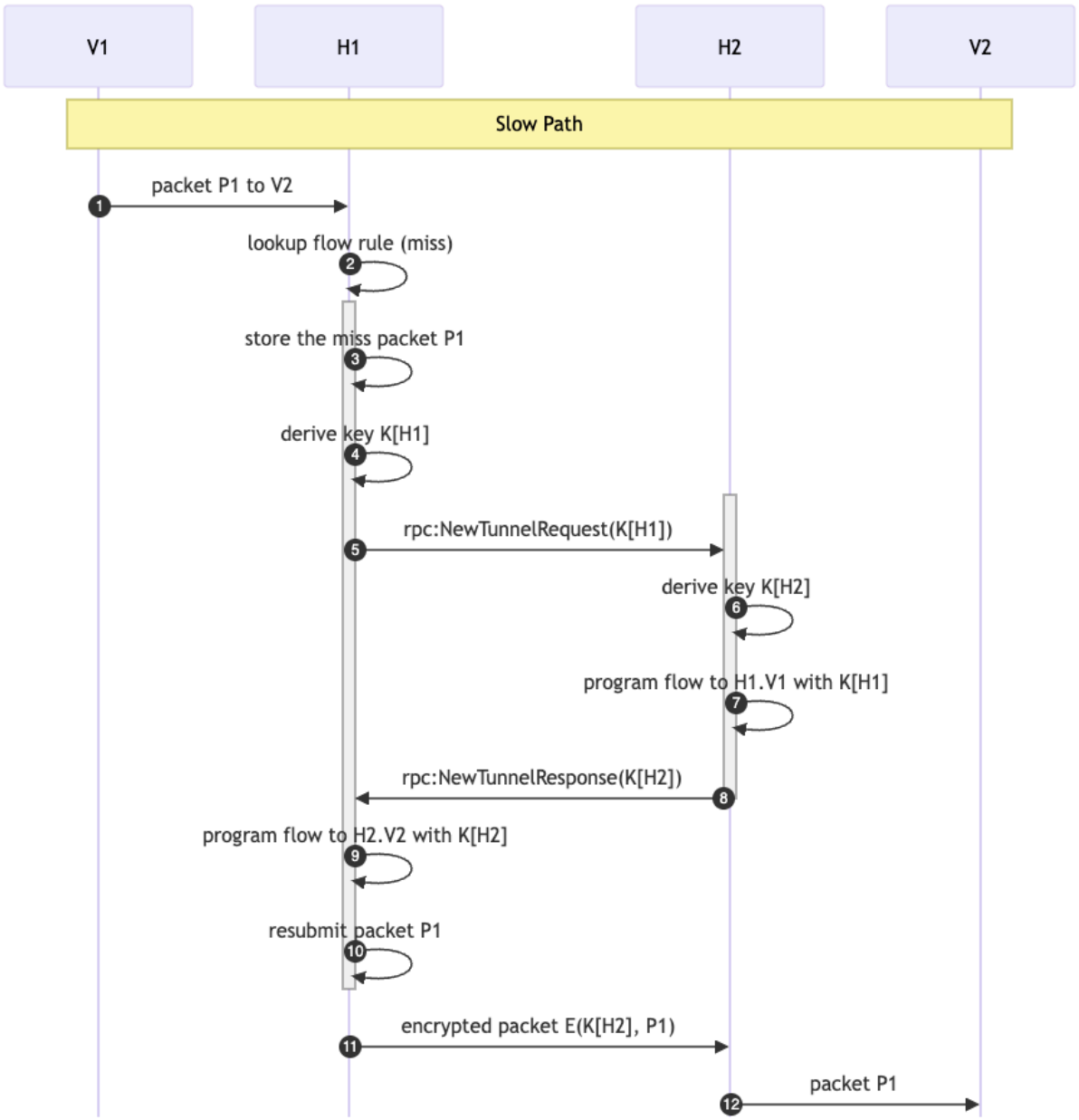
The creation of PSP tunnel connections requires two-way communication between peers. Each "sender" must request a unique security parameters index (SPI) and encryption key from the intended "receiver". The receiver derives sequential SPIs and encryption keys using the hardware resources inside the BlueField or ConnectX device, which manages a secret pair of master keys to produce the SPIs and encryption keys.

One key architectural benefit of PSP over similar protocols (e.g., IPsec) is that the receiver does not incur any additional resource utilization whenever it creates a new SPI and encryption key. This is because the decryption key associated with the SPI is computed on the fly, based on the SPI and master key, for each received packet. This lack of requirement for additional context memory for each additional decryption rule is partly responsible for the ability of the PSP protocol to scale to many thousands of peers.

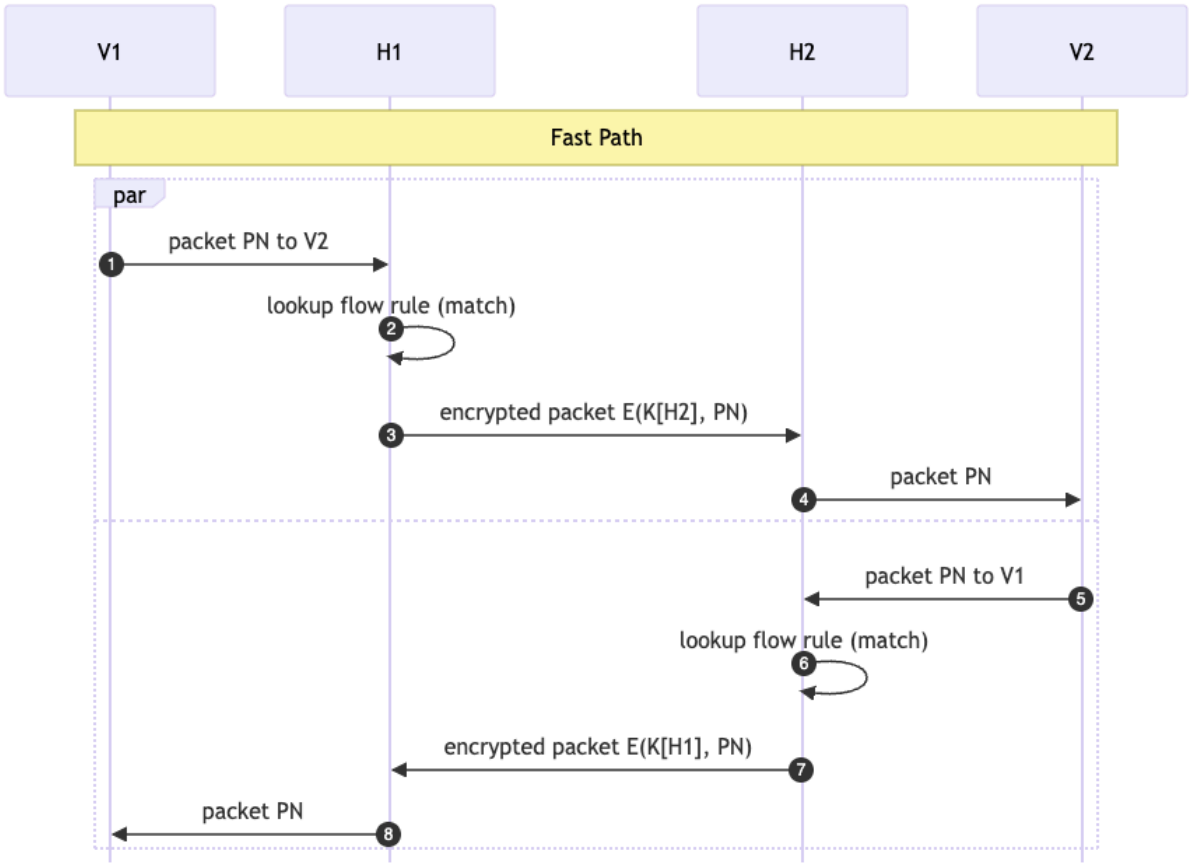
6.16.3.1 Startup vs. On-Demand Tunnel Creation

The default mode of operation is on-demand tunnel creation. That is, when a packet is received from the unsecured port for which the flow pipeline does not have an encryption rule, the packet misses to RSS, where the CPU must decide how to handle the packet. If the destination IP address in the packet belongs to a known peer's virtual network, the CPU uses gRPC on the OOB network connection to attempt a key exchange with the peer. If the key exchange is successful and a new encryption flow is created successfully, then the packet is then resubmitted to the pipeline, where it is encrypted and sent just as any of the following packets having the same destination IP address.

The following diagram illustrates this sequence (the "Slow Path"), for Virtual Machine V1 which intends to send a packet to Virtual Machine V2. In this case, V1 is hosted on physical host H1 and V2 on physical host H2. The first packet sent (1) results in a miss (2), so the packet is retained (3) while the keys are exchanged in both directions (4-8). Then the pipeline is updated (9) and the original packet is resubmitted (10). From there, the packet follows the same logic as the fast path, below.



Once the tunnel is established, and packets received from the VF (1) match a rule (2) and are encrypted and sent (3-4) without any intervention from the CPU ("Fast Path").



In the case of on-startup tunnel creation, the application's main thread repeatedly attempts to perform the key exchange for each of the peers specified on the command line until the list is completed. Each peer is connected only once and, if a connection to one peer fails, the loop continues onto the next peer and retries the failed connection after all the others have been attempted.

6.16.3.2 Sampling

The PSP gateway application supports the sample-at-receiver (S) bit in the PSP header. If sampling is enabled, then packets marked with the S bit are mirrored to the RSS queues and logged to the console. In addition, on transmit, the S bit of a random subset of packets (1 out of 2^N for command-line parameter N) is set to 1, and those packets are mirrored to RSS.

To avoid out-of-sequence indication in ROCE traffic, the **`--maintain-order`** flag can be used. When used, the application creates two mirror resources for egress sampling:

- One for when the sample bit is on – mirror is to RSS
- One for when the sample bit is off – mirror with fwd type drop

> ⚠ Sampling packets on transmit is currently supported only following encryption. Sampling of egress packets before encryption will be supported in a future release.

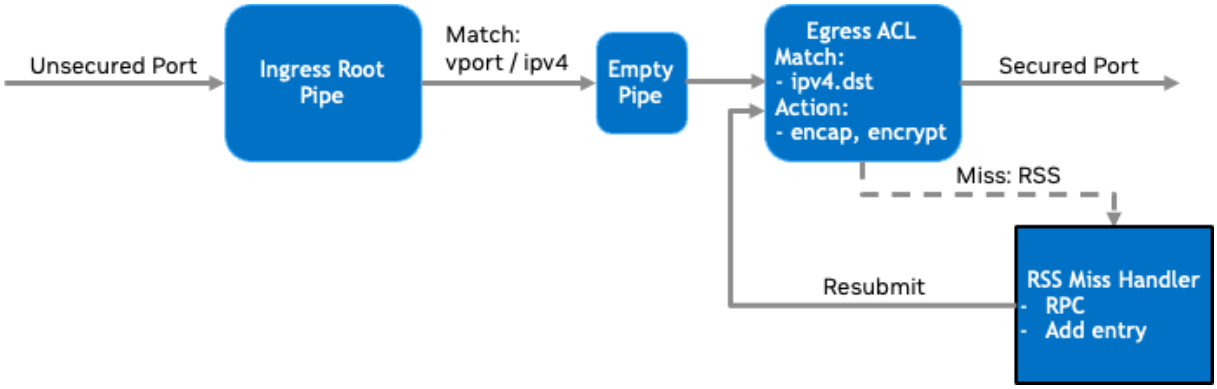
6.16.3.3 Pipelines

6.16.3.3.1 Host-to-Network Flows

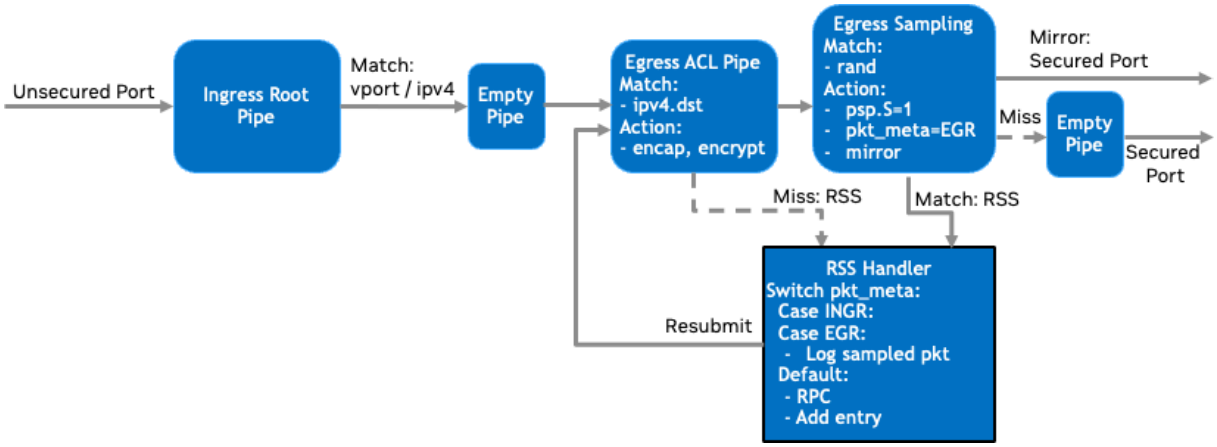
Traffic sent from the local, unsecured port (host-to-net) without sampling enabled travels through the pipeline as shown in the diagrams that follow. Note that the Ingress Root Pipe is the first destination for packets arriving from either the VF or the secured uplink port. However, the Egress ACL pipe is the first destination for packets sent via **tx_burst** on the PF (in the switch model's expert mode).

The Empty Pipe is a vestigial transition from the Default Domain, in which the Ingress Root Pipe is created, to the Secure Egress Domain, where the Egress ACL pipe performs encryption.

> ⚠ This pipe may be removed in a future release.



If sampling is enabled, the host-to-net pipeline is modified as shown in the following:



Here, an Egress Sampling Pipe is added between the Egress ACL Pipe and the Secured Port. It performs a match of the **random** metadata, masked according to command-line parameters, and then:

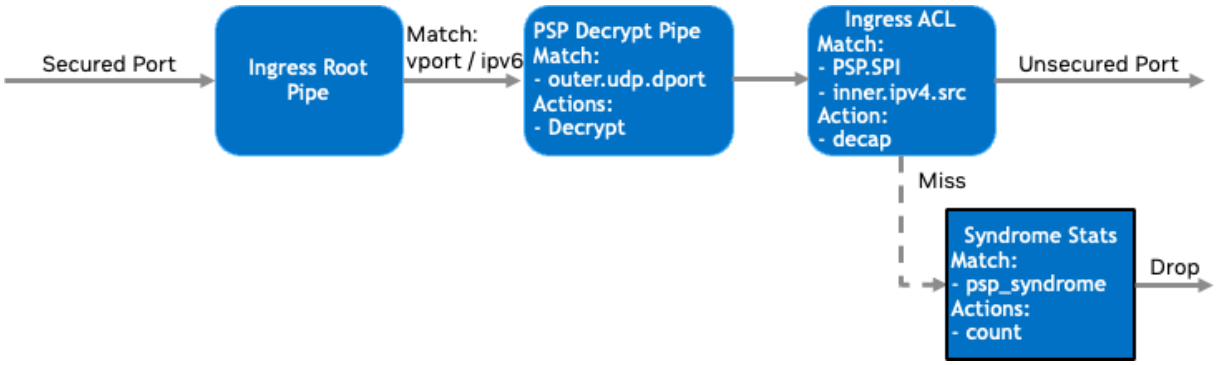
- On match, the following actions occur:
  a. Packet modifications:
    i. The S bit in the PSP header is set to **true**.
    ii. The **pkt_meta** field is set to a sentinel value to indicate to CPU software why the packet was sent to RSS.
  b. The original packet is forwarded to RSS.
  c. The mirror action forwards the packet to the secured port.
- On miss, the following actions occur:
  a. No packet modifications are made.
  b. The packet is forwarded to a vestigial pipe which can then forward the packet to the wire.

> A **fwd_miss** cannot target a port.

> This pipe may be removed in a future release.

6.16.3.3.2 Network-to-Host Flows

When a packet arrives from the secured port, the following flows are executed.

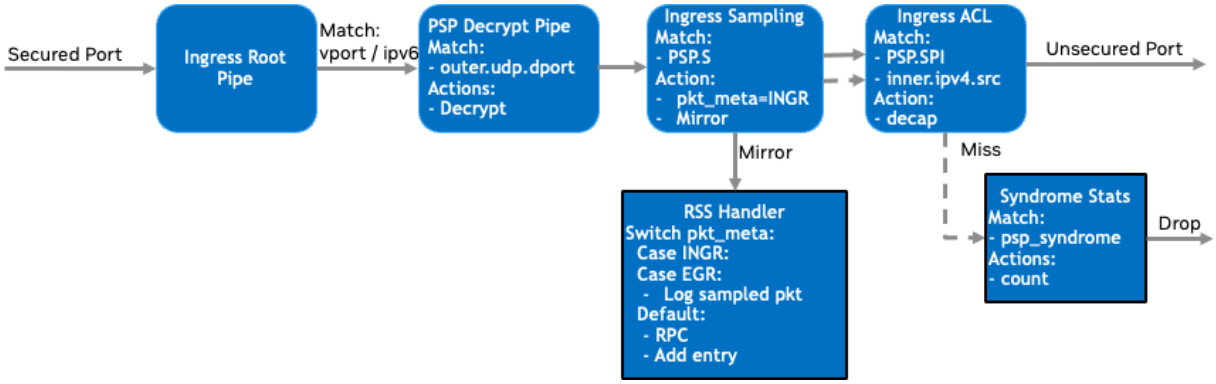


As before, the Ingress Root Pipe is the first destination and, here, the secured port ID as well as IPv6 outer L3 type are matched for. Matching packets flow to the decryption pipe, which matches the outer UDP port number against 1000, the constant specified in the PSP specification. On match, the packet is decrypted, but not yet de-capped. Then the Ingress ACL pipe checks the following:

- PSP_Syndrome – did the packet decrypt correctly and pass its ICV check?
- PSP SPI and inner IP src address – was this packet encrypted with the key associated with the given source?

If the packet passes the syndrome and ACL check, it is forwarded to the VF. Otherwise, the Syndrome Stats pipe counts the occurrences of the different bits in the PSP Syndrome word.

When sampling is enabled, the Ingress Sampling Pipe is inserted before the ACL. Unlike the Egress Sampling Pipe, no randomness is involved; the match criteria is the sample-on-receive flag in the PSP header. On a match, the incoming packet are mirrored to RSS with **pkt_meta** indicating the reason for forwarding the packet to RSS. On match or miss, the next pipe is the Ingress ACL Pipe.



6.16.3.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Flow

Refer to their respective programming guide for more information.

6.16.4 Compiling the Application

> Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

> For more information about the applications as well as development and compilation tips, refer to the DOCA Applications page.

The source code for the application is located in the following directory:

```
/opt/mellanox/doca/applications/psp_gateway/
```

6.16.4.1 Prerequisites

- The application relies on the **json-c** open source, requiring the following to be installed:

- Ubuntu/Debian:

```
$ sudo apt install libjson-c-dev
```

- CentOS/RHEL:

```
$ sudo yum install json-c-devel
```

- Installing the **gRPC** open source.

> **gRPC** can be installed either from upstream packages or by building it directly from source. Due to build changes on **gRPC** 's side, if building new versions directly from source, the following DOCA applications build flag will also be required:
>
> ```
> $ meson -Dupstream_grpc=false /tmp/build
> ```

6.16.4.2 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

> **doca_psp_gateway** is created under **/tmp/build/psp_gateway/** .

6.16.4.3 Compiling Only the Current Application

To directly build only the PSP Gateway application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_psp_gateway=true
ninja -C /tmp/build
```

> **doca_psp_gateway** is created under **/tmp/build/psp_gateway/** .

Alternatively, users can set the desired flags in the **meson_options.txt** file instead of providing them in the compilation command line:

1. Edit the following flags in **/opt/mellanox/doca/applications/meson_options.txt** :
   - Set **enable_all_applications** to **false**
   - Set **enable_psp_gateway** to **true**
2. Run the following compilation commands:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_psp_gateway is created under **/tmp/build/psp_gateway/**.

6.16.5 Running the Application

6.16.5.1 Prerequisites

The PSP gateway application is based on DOCA Flow. Therefore, the user is required to allocate huge pages:

```
$ echo '2048' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
$ sudo mkdir /mnt/huge
$ sudo mount -t hugetlbfs -o pagesize=2M nodev /mnt/huge
```

6.16.5.2 Application Execution

The PSP Gateway application is provided in source form. Therefore, a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_psp_gateway [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -s, --device                      Device identifier (required)
  -r, --repr                        Device representor (required)
  -m, --core-mask                   EAL Core Mask
  --decap-dmac                      mac_dst addr of the decapped packets
  -d, --vf-name                     Name of the virtual function device / unsecured port
  -n, --nexthop-dmac                next-hop mac_dst addr of the encapped packets
  --cookie                          Enable use of PSP virtualization cookies
  -a, --disable-ingress-acl         Allows any ingress packet that successfully decrypts
  --sample-rate                     Sets the log2 sample rate: 0: disabled, 1: 50%, ... 16: 1.5e-3%
  -x, --max-tunnels                 Specify the max number of PSP tunnels
  -o, --crypt-offset                Specify the PSP crypt offset
  --psp-version                     Specify the PSP version for outgoing connections (0 or 1)
  -z, --static-tunnels              Create tunnels at startup
  -k, --debug-keys                  Enable debug keys
  --stat-print                      Enable printing statistics
  --perf-print                      Enable printing performance metrics (key-gen, insertion, all)
  --show-rss-rx-packets             Show RSS rx packets
  --outer-ip-type                   outer IP type
  --inner-ip-type                   inner IP type
  -c, --config                      Path to the JSON file with application configuration
  --mode                            tunnel or transport mode
```

2. This usage printout can be printed to the command line using the **-h** (or **--help**) options:

```
./doca_psp_gateway -h
```

For additional information, refer to section "Command Line Flags".

3. CLI example for running the application on the BlueField or host:

```
./doca_psp_gateway -s pci/03:00.0 -r pci/03:00.0,pf0vf0 -d ens2f0v0 -c /tmp/psp_gw_config.json
```

- The device identifier (`-s` flag) should match the addresses of the desired PCIe device
- The `-d` flag indicates the vf that is used to extract MAC address and local IP. It should match the MAC address of the virtual function specified by the `-r` argument.
- The `-c` flag is the path to json configuration file that includes all info about tunnels.

For additional information regarding the command line syntax of the device and device representor identifiers, please referr to the matching section in the DOCA Arg Parser Guide.

6.16.5.3 Command Line Flags

6.16.5.3.1 General Flags

Short Flag	Long Flag	Description
`-h`	`--help`	Prints a help synopsis and exits
`-v`	`--version`	Prints program version information and exits
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.16.5.3.2 Program Flags

Short Flag	Long Flag	Description
`s`	`device`	Device identifier of the desired device
`r`	`repr`	Device representor identifier of the desired representor
`m`	`core-mask`	EAL core mask
N/A	`decap-dmac`	`mac_dst` address of the decapped packets (cannot be used with `vf-name`)
N/A	`vf-name`	Name of the VF device/unsecured port (Host only. Automatically detects MAC/VIP. Requires IP address bound to VF.)
`n`	`nexthop-dmac`	Next-hop `mac_dst` address of the encapped packets
N/A	`cookie`	Enable use of PSP virtualization cookies
`a`	`disable-ingress-acl`	Allows any ingress packet that successfully decrypts
N/A	`sample-rate`	Sets the log2 sample rate: • 0 – disabled, • 1 – 50%, ... • 16 – 1.5e-3%
`x`	`max-tunnels`	Specify the max number of PSP tunnels
`o`	`crypt-offset`	Specify the PSP crypt offset
N/A	`psp-version`	Specify the PSP version for outgoing connections (`0` or `1`)
`z`	`static-tunnels`	Create tunnels at startup using the given local IP address
`k`	`debug-keys`	Enable debug keys
N/A	`stat-print`	Print statistics regarding active sessions
N/A	`perf-print`	Print performance metrics of key generation of key-SPI pairs (`key-gen`) or insertion rate (`insertion`) or both (`all`)
N/A	`show-rss-rx-packets`	Print RX packets that got to RSS queues (i.e., to software)
N/A	`outer-ip-type`	Outer IP tunnel type (`ipv4` or `ipv6`). When not specified, default is `ipv6`.
N/A	`inner-ip-type`	Inner IP tunnel type (`ipv4` or `ipv6`). When not specified, the default is `ipv4`.
`c`	`config`	Path to JSON file that includes tunnel information See section "Config File JSON Format" for specifics on the JSON file format.

Short Flag	Long Flag	Description
N/A	`maintain-order`	Force maintaining original packet ordering

6.16.5.4 Config File JSON Format

The JSON configuration file maps an OOB network address (gRPC server) to a list of virtual IP addresses behind that physical address. Each "peer" in the list contains information about a remote gRPC server to which the pairs of `local-vip` and `remote-vip` are mapped to.

The following is a template of the JSON file structure:

Json config

```
{
    "config": {
        "local-grpc-address": <local gRPC server ip address to listen for connections on, possible with port>
    },
    "peers": [ /* list of peers (differentiated by remote gRPC server) */
        {
            "remote-grpc-address": <remote gRPC server address to connect to for connections of given sessions> ,
            "sessions": [ /* list of sessions allowed with peer (differentiated by local vip) */
                {
                    "local-vip": <local virtual ip>, /* if not present, take the value from vf interface's IP ("--
vf-name" flag) */
                    "remote-vips": [
                        <ip address, possible with prefix mask>,
                        /* [<more ip addresses>...] */
                    ]
                },
                /* [<more sessions>... ] */
            ]
        },
        /* [<more peers>...] */
    ]
}
```

When a packet from the VF does not match any existing flows, this table defines the physical peer which should provide the tunnel to the given (virtual) src-ip, dst-ip 2-tuple.

Example JSON:

Json config

```
{
    "config": {
        "local-grpc-address": "10.237.113.143:3000"
    },
    "peers": [
        {
            "remote-grpc-address": "10.237.113.144",
            "sessions": [
                {
                    "local-vip": "6.6.6.5",
                    "remote-vips": [
                        "8.8.8.0/24",
                        "7.7.7.7"
                    ]
                },
                {
                    "local-vip": "6.6.6.4",
                    "remote-vips": [
                        "9.9.9.9"
                    ]
                }
            ]
        },
        {
            "remote-grpc-address": "10.237.1.1",
            "sessions": [
                {
```

```
                    // "local vip" not present - default value from vf interface's IP (Must run application with
"--vf-name" param)
                    "remote-vips": [
                        "11.11.11.11/24"
                    ]
                }
            ]
        }
    ]
}
```

6.16.5.5 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.16.6 Application Code Flow

1. Main loop code flow
    a. Initialize the logger facility.
        i. The standard logger and the SDK logger are created, and the SDK logger default log level is selected.

```
doca_log_backend_create_standard();
doca_log_backend_create_with_file_sdk(stdout, &sdk_log);
doca_log_backend_set_sdk_level(sdk_log, DOCA_LOG_LEVEL_WARNING);
```

        ii. The signal handler is connected to enable a clean shutdown.

```
signal(SIGINT, signal_handler);
signal(SIGTERM, signal_handler);
```

    b. Parse application arguments. The main function invokes **psp_gw_argp_exec()**, which initializes the arg parser resources and registers DOCA general parameters, and then registers the PSP application-specific parameters. Then the parser is invoked.

```
doca_argp_init();
psp_gw_register_params();
doca_argp_start();
```

    c. Parse input configuration JSON file. This will include all information about peers and their sessions.

```
psp_gw_parse_config_file();
```

    d. DPDK initialization. Call **rte_eal_init()** to initialize EAL resources with the provided EAL flags for not probing the ports (**-a00:0.0**).

```
rte_eal_init(n_eal_args, (char **)eal_args);
```

    e. Initialize devices and ports.
        i. Open DOCA devices with input PCIe addresses/interface names.
        ii. Probe DPDK port from each opened device.

```
open_doca_device_with_pci(...); // not part of doca_flow; see doca/samples/common.c
doca_dpdk_port_probe(...);
```

        iii. The MAC and IP addresses of the PF are queried and logged.

```
rte_eth_macaddr_get(...);
doca_devinfo_get_ipv6_addr(...);
DOCA_LOG_INFO("Port %d: Detected PF mac addr: %s, IPv6 addr: %s, total ports: %d", ...);
```

f. Initialize and start DPDK ports. Initialize DPDK ports, including mempool allocation. No hairpin queues are created.

```
dpdk_queues_and_ports_init(); // not part of doca_flow; see doca/applications/common/dpdk_utils.c
```

g. Initialize DOCA Flow objects used by the PSP Gateway application. The DOCA Flow library is initialized with the string **"switch,hws,isolated,expert"**, because it is desirable for the application to act as an intermediary between the uplink physical port and some number of VF representors (switch mode), and **hws** (hardware steering mode) and **isolated** mode are mandatory for switch mode. The optional **expert** flag prevents DOCA Flow from automating certain packet operations and gives more control to the application, as described in the DOCA Flow page.

```
PSP_GatewayFlows psp_flows(&pf_dev, vf_port_id, &app_config);
psp_flows.init();
```

   i. Initialize DOCA Flow library.
   ii. Start the ports.
   iii. Allocate shared resources (PSP crypto objects and Mirror actions).
   iv. Create the ingress and egress pipes.

h. Create the gRPC service.

```
PSP_GatewayImpl psp_svc(&app_config, &psp_flows);
```

i. Launch the L-Core threads to handle RSS packets.

```
rte_eal_remote_launch(lcore_pkt_proc_func, &lcore_params, lcore_id);
```

j. Launch the gRPC service.
   i. This implementation uses **InsecureServerCredentials**. Update as needed.

```
grpc::ServerBuilder builder;
builder.AddListeningPort(server_address, grpc::InsecureServerCredentials());
builder.RegisterService(&psp_svc);
auto server_instance = builder.BuildAndStart();
```

k. Wait for traffic. If configured to connect at startup, process the list of remaining connections. Then display the flow pipe counters.

```
while (!force_quit) {
    psp_svc.try_connect(remotes_to_connect, local_vf_addr);
...
    psp_flows.show_static_flow_counts();
    psp_svc.show_flow_counts();
}
```

   - Wait in a loop until the user terminates the program.

l. PSP Gateway cleanup:
   i. Destroy DPDK ports and queues.

```
dpdk_queues_and_ports_fini();
```

ii. DPDK finish.

```
dpdk_fini();
```

Calls **rte_eal_destroy()** to destroy initialized EAL resources.

iii. Arg parser destroy.

```
doca_argp_destroy()
```

2. Miss-packet code flow.
   a. The L-Core launch routine from the main loop pointed to the **lcore_pkt_proc_func** routine.
   b. The **force_quit** flag is polled to respond to the signal handler.

```
while (!*params->force_quit) { ... }
```

   c. The **rte_eth_rx_burst** function polls the PF queue for received packets.

```
nb_rx_packets = rte_eth_rx_burst(port_id, queue_id, rx_packets, MAX_RX_BURST_SIZE);
```

   d. Inside **handle_packet()**, the packet metadata is inspected to detect whether this packet is sampled on ingress, sampled on egress, or a miss packet.

```
uint32_t pkt_meta = rte_flow_dynf_metadata_get(packet);
```

      i. Sampled packets are simply logged using the **rte_pktmbuf_dump** function.

   e. Miss packets are passed to the **handle_miss_packet** method of the gRPC service. This method handles cases where an application attached to the VF wishes to send a packet to another virtual address, but a PSP tunnel must first be established by exchanging SPI and key information between hosts.
   f. The service acts as a gRPC client, and the appropriate server is looked up from the **config->net_config.hosts** vector, which is comprised of hosts passed via the **-t** tunnels arguments or the **-f** tunnels file argument.
   g. Once the client connection exists, the **request_tunnel_to_host** method takes care of invoking the the **RequestTunnelParams** operation defined in the schema.
      - Optionally, this function generates a corresponding set of tunnel parameters appropriate for the server host to send traffic back via **generate_tunnel_params()**.

```
doca_flow_crypto_psp_spi_key_bulk_generate(bulk_key_gen);
doca_flow_crypto_psp_spi_key_bulk_get(bulk_key_gen, 0, &spi, key);
doca_flow_crypto_psp_spi_key_wipe(bulk_key_gen, 0);
```

   h. The RPC operation is invoked, and if successful, **create_tunnel_flow** is called to create the egress flow:

```
status = stub->RequestTunnelParams(&context, request, &response);
```

i. The **create_tunnel_flow** method translates the resulting Protobuf objects to application-specific data structures and passes them to the **add_encrypt_entry** method of the flows object. Here, the PSP SPI and key are programmed into an available **crypto_id** index as follows.

> ⚠ SPI and **crypto_id** are two independent concepts:
>
> - The SPI value in the PSP packet header indicates to the receiver which key was used by the sender to encrypt the data. Each receiver computes an SPI and key to provide to a sender. Since each receiver is responsible for tracking its next SPI, multiple receivers may provide the same SPI to a sender, so one sender may send the same SPI to multiple different peers. This is allowed, as each of the receiving peers has its own decryption key to handle that SPI.
> - The **crypto_id** acts as an index into the bucket of PSP keys allocated by DOCA Flow. The **doca_flow_shared_resource_cfg()** function writes a given PSP encryption key to a given slot in the bucket of keys in NIC memory. These slots can be overwritten as needed by the application.
> - There is no explicit association between **crypto_id** and SPI. The **doca_flow_shared_resource_cfg()** function writes a key at the slot provided by the **crypto_id** argument, then the flow pipe entry **actions.crypto.crypto_id** references this key, and **actions.crypto_encap.encap_data** includes a PSP header with the desired SPI.

```
struct doca_flow_shared_resource_cfg res_cfg = {};
res_cfg.domain = DOCA_FLOW_PIPE_DOMAIN_SECURE_EGRESS;
res_cfg.psp_cfg.key_cfg.key_type = DOCA_FLOW_CRYPTO_KEY_256;
res_cfg.psp_cfg.key_cfg.key = (uint32_t *)encrypt_key;
doca_flow_shared_resource_cfg(DOCA_FLOW_SHARED_RESOURCE_PSP, session->crypto_id, &res_cfg);
```

j. A flow pipe entry which references the newly programmed PSP encryption key (via its index **crypto.crypto_id**) must be inserted. Additionally, this pipe entry must specify all the outer Ethernet, IP, UDP, and PSP header fields to insert.

```
format_encap_data(session, actions.crypto_encap.encap_data);
actions.crypto.action_type = DOCA_FLOW_CRYPTO_ACTION_ENCRYPT;
actions.crypto.resource_type = DOCA_FLOW_CRYPTO_RESOURCE_PSP;
actions.crypto.crypto_id = session->crypto_id;
...

doca_flow_pipe_add_entry(pipe_queue, pipe, match, actions, mon, fwd, flags, &status, entry);
...

doca_flow_entries_process(port, 0, DEFAULT_TIMEOUT_US, num_of_entries);
```

k. The original packet received via **rte_ethdev_rx_burst** is sent back through the newly updated pipelines via **rte_ethdev_tx_burst**. Since the **port_id** argument is that of the PF, and since DOCA Flow has been initialized in **expert** mode, the packet is transferred to the root of the egress domain (the "empty pipe" before **egress_acl_pipe**).

```
nsent = rte_eth_tx_burst(port_id, queue_id, &packet, 1);
```

3. Tunnel parameter request handling
    a. The gRPC service provided by the PSP Gateway implements the **RequestTunnelParams** operation referenced above. A client uses this operation to request an SPI and key to encrypt traffic to send to the server's NIC device. The request indicates the virtual remote address for which the tunnel will be created.
    b. This operation begins by generating a new SPI and key inside **generate_tunnel_params()** as described previously.
    c. The operation creates an ACL entry permitting the new SPI and the remote virtual address using the **add_ingress_acl_entry** method of the Flows object.

```
doca_flow_match match = {};
match.parser_meta.psp_syndrome = 0;
match.tun.type = DOCA_FLOW_TUN_PSP;
match.tun.psp.spi = RTE_BE32(session->spi_ingress);
match.inner.l3_type = DOCA_FLOW_L3_TYPE_IP4;
match.inner.ip4.src_ip = session->src_vip;
...

doca_flow_pipe_add_entry(pipe_queue, pipe, match, actions, mon, fwd, flags, &status, entry);
...

doca_flow_entries_process(port, 0, DEFAULT_TIMEOUT_US, num_of_entries);
```

    d. If the request included parameters for traffic in the reverse direction (traffic to encrypt and send to the client), these parameters are translated and passed to the Flows object by calling **create_tunnel_flow** described above.

6.16.6.1 References

- PSP Security Protocol Specification
- Google's Open-Source PSP tools
- Google Remote Procedure Calls library

6.17 DOCA Secure Channel Application Guide

This guide provides a secure channel implementation on top of NVIDIA® BlueField® DPU.

6.17.1 Introduction

The DOCA Secure Channel reference application leverages the DOCA Comch API which creates a secure, network independent communication channel between the host and the NVIDIA BlueField DPU.

Comch allows the host to control services on the DPU, activate certain offloads, or exchange messages using client-server/producer-consumer framework.

The client (host) side can communicate only with one server at a time while the server side is able to communicate with multiple clients.

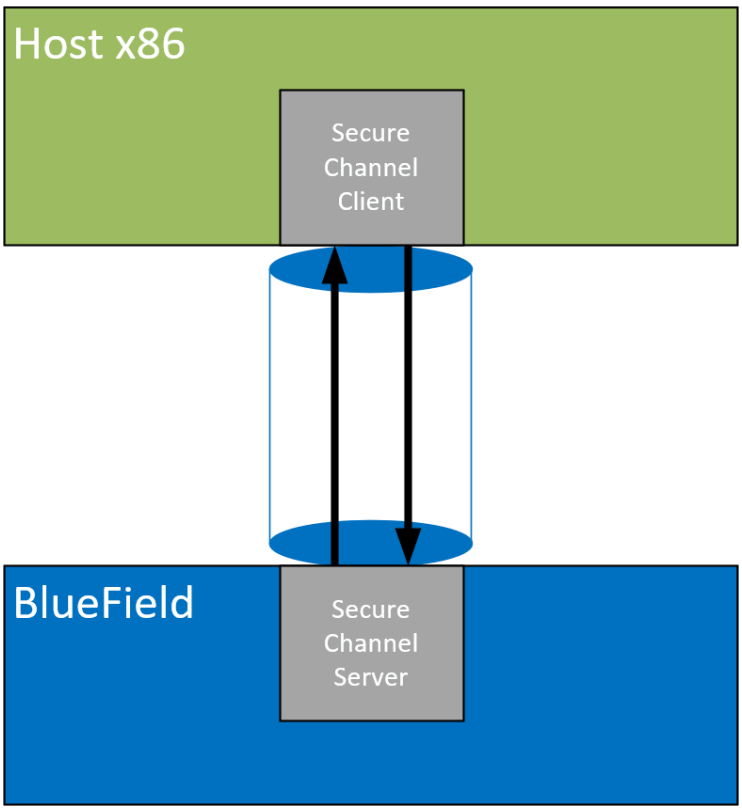
The API allows communication between any PF/VF/SF on the host to the server located on the DPU.

Once a client-server connection is established, multiple producer-consumer instances can be spawned to transfer large amount of data in a first-in-first-out (FIFO) style queue.

Secure channel allows the user to select the message size and amount to be exchanged between the client and the server to simulate heavy load on the channel.

6.17.2 System Design

A secure channel application runs on client mode (host) and server mode (DPU). Once a channel is open, messages can flow from both sides.



6.17.3 Application Architecture

The secure channel application runs on top of the DOCA Comch API.

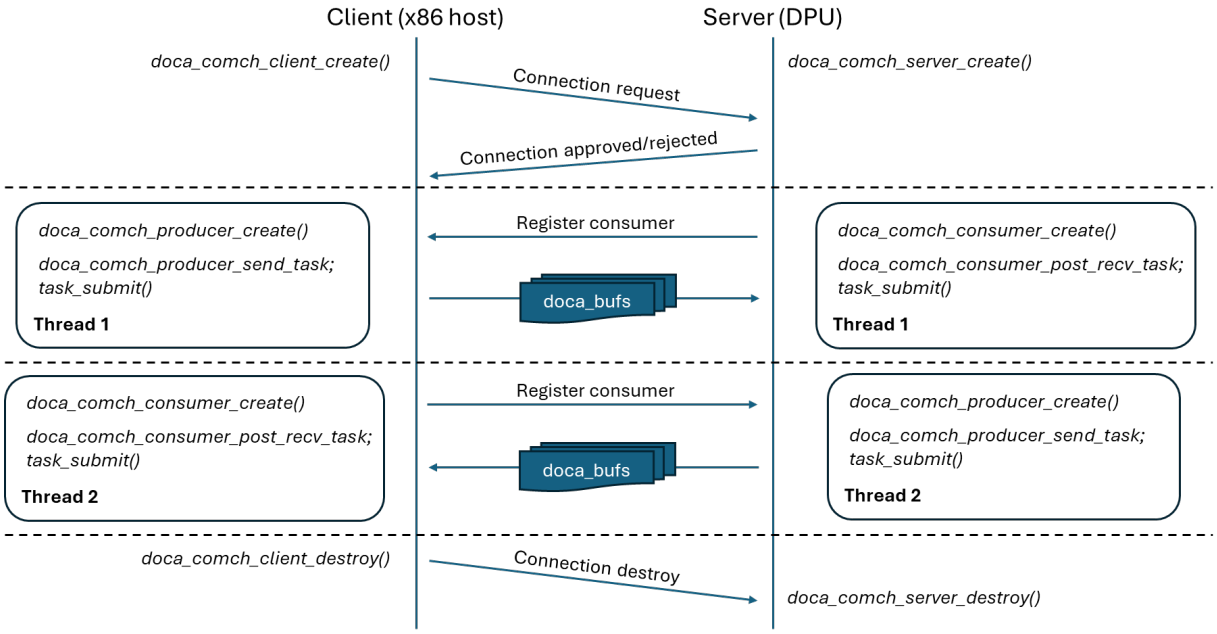
The application begins by establishing a client-server connection between host (client) and DPU (server). Once this connection is established, the application on both sides spawns two new threads:

- Consumer – The consumer thread registers a number of **post_recv** buffers with the opposite end of the connection
- Producer – The producer thread, after it detects the consumer, populates these **post_recv** buffers with data that can then be read by the consumer

Once the consumer has processed a received buffer, the buffer is reposted (via task submit) to again be populated by the opposite producer.

When the producer has sent the requested number of messages, and the consumer has received the same amount, both threads exit and the main client-server connection is destroyed.

The flow is described in the following diagram:



6.17.4 DOCA Libraries

This application leverages the following DOCA library:

- DOCA Comch

Refer to its respective programming guide for more information.

6.17.5 Compiling the Application

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for the application is located in the following directory:

```
/opt/mellanox/doca/applications/secure_channel/
```

6.17.5.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_secure_channel is created under **/tmp/build/secure_channel/** .

6.17.5.2 Compiling Only the Current Application

To directly build only the secure channel application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_secure_channel=true
ninja -C /tmp/build
```

doca_secure_channel is created under **/tmp/build/secure_channel/** .

Alternatively, users can set the desired flags in the **meson_options.txt** file instead of providing them in the compilation command line:

1. Edit the following flags in **/opt/mellanox/doca/applications/meson_options.txt** :
   - Set **enable_all_applications** to **false**
   - Set **enable_secure_channel** to **true**
2. Run the following compilation commands:

   ```
   cd /opt/mellanox/doca/applications/
   meson /tmp/build
   ninja -C /tmp/build
   ```

   doca_secure_channel is created under **/tmp/build/secure_channel/** .

6.17.6 Running the Application

6.17.6.1 Application Execution

The secure channel application is provided in source form. Therefore, a compilation is required before the application can be executed.

- Application usage instructions:

  ```
  Usage: doca_secure_channel [DOCA Flags] [Program Flags]
  ```

```
DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -s, --msg-size                    Message size to be sent
  -n, --num-msgs                    Number of messages to be sent
  -p, --pci-addr                    DOCA Comm Channel device PCI address
  -r, --rep-pci                     DOCA Comm Channel device representor PCI address (needed only on DPU)
```

This usage printout can be printed to the command line using the **-h** (or **--help**) options:

```
./doca_secure_channel -h
```

For additional information, refer to section "Command Line Flags".

- CLI example for running the application on the BlueField:

```
./doca_secure_channel -s 256 -n 10 -p 03:00.0 -r 3b:00.0
```

Both the DOCA Comch device PCIe address (**03:00.0**) and the DOCA Comch device representor PCIe address (**3b:00.0**) should match the addresses of the desired PCIe devices. Users may use the DOCA Capabilities Print Tool to find out their device and device representor PCIe addresses.

- CLI example for running the application on the host:

```
./doca_secure_channel -s 256 -n 10 -p 3b:00.0
```

The DOCA Comch device PCIe address (**3b:00.0**) should match the address of the desired PCIe device.

6.17.6.2 Command Line Flags

6.17.6.2.1 General Flags

Short Flag	Long Flag	Description
-h	**--help**	Prints a help synopsis and exits
-v	**--version**	Prints program version information and exits

Short Flag	Long Flag	Description
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.17.6.2.2 Program Flags

Short Flag	Long Flag	Description
`s`	`msg-size`	Message size in bytes ⚠ This is a mandatory flag.
`n`	`num-msgs`	Number of messages to send on both sides ⚠ This is a mandatory flag.
`p`	`pci-addr`	DOCA Comch device PCIe address ⚠ This is a mandatory flag.
`r`	`rep-pci`	DOCA Comch device representor PCIe address ⚠ This is a mandatory flag only on the DPU.

6.17.6.3 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.17.7 Application Code Flow

1. Parse application argument.
    a. Initialize the arg parser resources and register DOCA general parameters.

    ```
    doca_argp_init();
    ```

    b. Register secure channel application parameters.

    ```
    register_secure_channel_params();
    ```

    c. Parse application parameters:

    ```
    doca_argp_start();
    ```

    d. Establish the client-server control path connection:

    ```
    comch_utils_fast_path_init();
    ```

        i. Create a **doca_comch_client** or **doca_comch_server** depending on where the application is run (x86 host or DPU Arm cores).
        ii. Register a callback which is triggered when new consumers are created on the opposite end.
        iii. Progress the connection until it becomes established.
2. Run main logic.

    ```
    sc_start();
    ```

    a. Send a message on the control channel telling the opposite end how many fastpath messages it intends to send and their length.
    b. Wait to receive a similar message from the opposite side.
    c. Start a producer thread:
        i. Create and starts a **doca_comch_producer**.
        ii. Wait until a consumer has been registered from the opposite end of the control channel.
        iii. Create a **doca_buf** of input message length to send repeatedly.
        iv. Submit the max number of **doca_comch_producer_task_send** tasks each containing the **doca_buf**.
        v. Resubmit each task from its completion callback until the requested number of tasks are sent.
        vi. Destroy producer and ends thread.
    d. Start a consumer thread
        i. Create and starts a **doca_comch_consumer**.
        ii. Progress until consumer has been fully registered with the control channel connection.
        iii. Create **doca_buf**s of negotiated size to receive data on.
        iv. Submit the max number of **doca_comch_consumer_task_post_recv** tasks with each of the allocated **doca_buf**s.

v. Resubmit each `post_recv` buffer from its callback when it has been populated by a producer.
vi. Destroy consumer and thread when it has received the communicated number of fastpath messages.

e. Sends (DPU) or waits (host) on a message to indicate that all fastpath messages have completed and the Comch control connection can be destroyed.

6.17.8 References

- `/opt/mellanox/doca/applications/secure_channel/`

6.18 DOCA Simple Forward VNF Application Guide

This guide provides a Simple Forward implementation on top of NVIDIA® BlueField® DPU.

6.18.1 Introduction

Simple forward is a forwarding application that leverages the DOCA Flow API to take either VXLAN, GRE, or GTP traffic from a single RX port and transmits it on a single TX port.

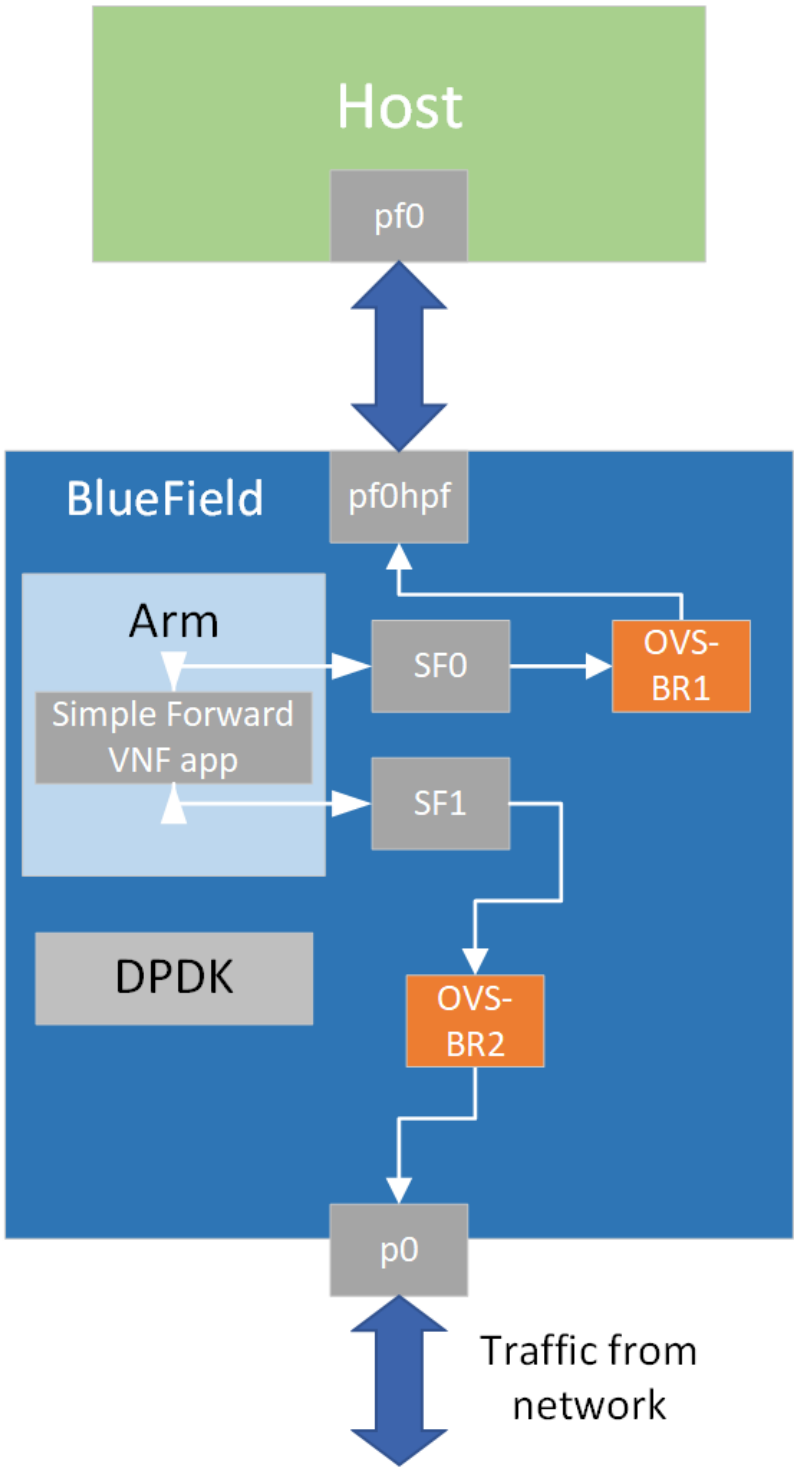
For every packet received on an RX queue on a given port, DOCA Simple Forward checks the packet's key, which consists of the inner 5-tuple. If it finds that the packet matches an existing flow, then it does not create a new one. Otherwise, a new flow is created with a FORWARDING component. Finally, the packet is forwarded to the TX queue of the egress port if the "rx-only" mode is not set.

The FORWARDING component type depends on the flags delivered when running the application. For example, if the `hairpinq` flag is provided, then the FORWARDING component would be hairpin. Otherwise, it would be RSS'd to software, and hence every VXLAN, GTP, or GRE packet would be received on RX queues.

Simple forward should be run with dual ports. By using a traffic generator, the RX port receives the VXLAN, GRE, or GTP packets and forwarding forwards them back to the traffic generator.

6.18.2 System Design

The following diagram illustrates simple forward's packet flows. It receives traffic coming from the wire and passes it to the other port.



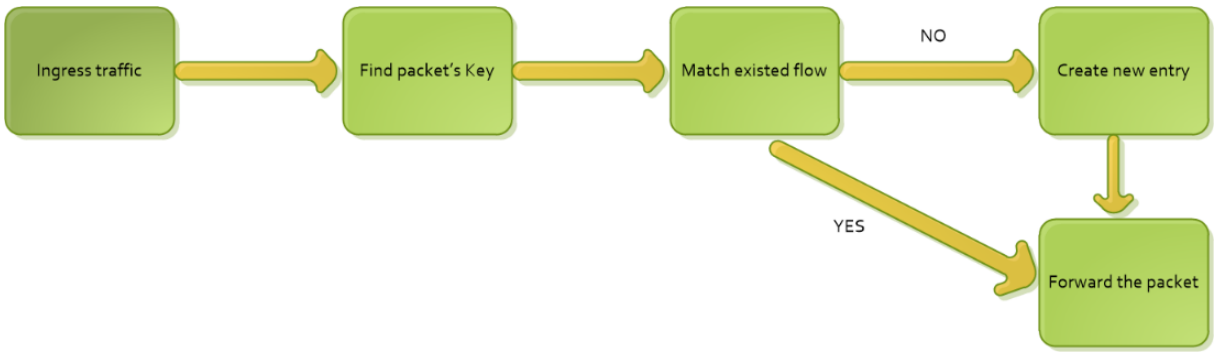
6.18.3 Application Architecture

Simple forward first initializes DPDK, after which the application handles the incoming packets.

The following diagram illustrates the initialization process.

1. **Init_DPDK** – EAL init, parse argument from command line and register signal.
2. Start port – **mbuf_create** , **dev_configure** , rx/tx/hairpin queue setup and start the port.
3. **Simple_fwd INIT** – create flow tables, build default forward pipes.

The following diagram illustrates how to process the packet.



1. Based on the packet's info, find the key values based on the inner headers (e.g. src/dst IP, src/dst port, etc).
2. Traverse the inner flow tables, check if the keys exist or not.
   - If yes, update inner counter
   - If no, a new flow table is added to the DPU
3. Forward the packet to the other port.

6.18.4 DOCA Libraries

This application leverages the following DOCA library:

- DOCA Flow

Refer to its respective programming guide for more information.

6.18.5 Compiling the Application

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for the application is located in the following directory:

```
/opt/mellanox/doca/applications/simple_fwd_vnf/
```

6.18.5.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_simple_fwd_vnf is created under **/tmp/build/simple_fwd_vnf/**.

6.18.5.2 Compiling Simple Forward Application Only

To directly build only the simple forward application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_simple_fwd_vnf=true
ninja -C /tmp/build
```

doca_simple_fwd_vnf is created under **/tmp/build/simple_fwd_vnf/**.

Alternatively, users can set the desired flags in the **meson_options.txt** file instead of providing them in the compilation command line:

1. Edit the following flags in **/opt/mellanox/doca/applications/meson_options.txt**:
   - Set **enable_all_applications** to **false**
   - Set **enable_simple_fwd_vnf** to **true**
2. Run the following compilation commands:

   ```
   cd /opt/mellanox/doca/applications/
   meson /tmp/build
   ninja -C /tmp/build
   ```

   doca_simple_fwd_vnf is created under **/tmp/build/simple_fwd_vnf/**.

6.18.6 Running the Application

6.18.6.1 Prerequisites

1. A FLEX profile number should be manually set to 3 on the system for the application to build the GRE, standard VXLAN and GRE pipes.
    a. Set FLEX profile number to 3 from the DPU.

    ```
    sudo mlxconfig -d <pcie_address> s FLEX_PARSER_PROFILE_ENABLE=3
    ```

    b. Perform a BlueField system reboot for the **mlxconfig** settings to take effect.

    > Resetting the firmware can be done from the BlueField as well. For more information, refer to step 3.b of the "Upgrading Firmware" section of the DOCA Installation Guide for Linux.

2. The Simple Forward application is based on DOCA Flow. Therefore, the user is required to allocate huge pages.

    ```
    $ echo '2048' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
    $ sudo mkdir /mnt/huge
    $ sudo mount -t hugetlbfs -o pagesize=2M nodev /mnt/huge
    ```

6.18.6.2 Application Execution

The simple forward application is provided in source form. Therefore, a compilation is required before the application can be executed.

- Application usage instructions:

```
Usage: doca_simple_fwd_vnf [DPDK Flags] -- [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -a, --device                      device
  -r, --rep                         device representor
  -t, --stats-timer <time>          Set interval to dump stats information
  -q, --nr-queues <num>             Set queues number
  -rx, --rx-only                    Set rx only
  -o, --hw-offload                  Set PCI address of the RXP engine to use
  -hq, --hairpinq                   Set forwarding to hairpin queue
  -g, --age-thread                  Start thread to do aging
```

> This usage printout can be printed to the command line using the **-h** (or **--help**) options:
>
> ```
> ./doca_simple_fwd_vnf -- -h
> ```

For additional information, refer to section "Command Line Flags".

- CLI example for running the application on the BlueField:

```
./doca_simple_fwd_vnf -- -a aux/4 -a aux/5 -l 60
```

SFs must be enabled according to the BlueField Scalable Functions.

Before creating SFs on a specific physical port, it is important to verify the encap mode on the respective PF FDB. The default mode is **basic**. To check the encap mode, run:

```
cat /sys/class/net/p0/compat/devlink/encap
```

In this case, disable encap on the PF FDB before creating the SFs by running:

```
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode legacy
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode legacy
echo none > /sys/class/net/p0/compat/devlink/encap
echo none > /sys/class/net/p1/compat/devlink/encap
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode switchdev
/opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode switchdev
```

If the encap mode is set to **basic** then the application fails upon initialization.

The flag **-a aux/4 -a aux/5** is mandatory for proper usage of the application.

a. Modifying this flag will result in an unexpected behavior as only 2 ports are supported.

b. The SF number is arbitrary and configurable.

The SF numbers must match the desired SF devices.

- CLI example for running the application on the host:

```
./doca_simple_fwd_vnf -- -a pci/04:00.3 -a pci/04:00.4 -l 60
```

The device identifiers must match the desired network devices.

For additional information regarding the command line syntax of the device and device representor identifiers, please referr to the matching section in the DOCA Arg Parser Guide.

For more information, refer to section "Running DOCA Application on Host" in DOCA Virtual Functions.

6.18.6.3 Command Line Flags

6.18.6.3.1 General Flags

Short Flag	Long Flag	Description
`-h`	`--help`	Prints a help synopsis and exits
`-v`	`--version`	Prints program version information and exits
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.18.6.3.2 Host Program Flags

Short Flag	Long Flag	Description
`a`	`device`	Device identifier
`r`	`rep`	Device representor identifiers
`t`	`stats-timer`	Set interval to dump stats information.
`q`	`nr-queues`	Set queues number.
`rx`	`rx-only`	Set RX only. When set, the packets will not be sent to the TX queues.
`o`	`hw-offload`	Set HW offload of the RXP engine to use.
`hq`	`hairpinq`	Set forwarding to hairpin queue.
`g`	`age-thread`	Start a dedicated thread that handles the aged flows.

6.18.6.4 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.18.7 Application Code Flow

1. Parse application argument.
    a. Initialize arg parser resources and register DOCA general parameters.

    ```
    doca_argp_init();
    ```

    b. Register application parameters.

    ```
    register_simple_fwd_params();
    ```

    c. Parse the arguments.

    ```
    doca_argp_start();
    ```

        i. Parse DPDK flags and invoke handler for calling the **rte_eal_init()** function.
        ii. Parse app parameters.
2. DPDK initialization.

    ```
    dpdk_init();
    ```

    Calls **rte_eal_init()** to initialize EAL resources with the provided EAL flags.

3. DPDK port initialization and start.

    ```
    dpdk_queues_and_ports_init();
    ```

    a. Initialize DPDK ports.
    b. Create mbuf pool using **rte_pktmbuf_pool_create**.
    c. Driver initialization – use **rte_eth_dev_configure** to configure the number of queues.
    d. Rx/Tx queue initialization – use **rte_eth_rx_queue_setup** and **rte_eth_tx_queue_setup** to initialize the queues.
    e. Rx hairpin queue initialization – use **rte_eth_rx_hairpin_queue_setup** to initialize the queues.
    f. Start the port using **rte_eth_dev_start**.
4. Simple forward initialization.

    ```
    simple_fwd_init();
    ```

    a. **simple_fwd_create_ins** – create flow tables using **simple_fwd_ft_create**.

b. **simple_fwd_init_ports_and_pipes** – initialize DOCA port using **simple_fwd_init_doca_port** and build default pipes for each port.

5. Main loop.

```
simple_fwd_process_pkts();
```

a. Receive packets using **rte_eth_rx_burst** in a loop.
b. Process packets using **simple_fwd_process_offload**.
c. Transmit the packets on the other port by calling **rte_eth_tx_burst**. Or free the packet mbuf if **rx_only** is set to **true**.

6. Process packets.

```
simple_fwd_process_offload();
```

a. Parse the packet's **rte_mbuf** using **simple_fwd_pkt_info**.
b. Handle the packet using **simple_fwd_handle_packet**. If the packet's key does not match the existed the flow entry, create a new flow entry and PIPE using **simple_fwd_handle_new_flow**. Otherwise, increase the total packet's counter.

7. Simple forward destroy.

```
simple_fwd_destroy();
```

Simple forward close port and clean the flow resources.

8. DPDK ports and queues destruction.

```
dpdk_queues_and_ports_fini();
```

9. DPDK finish.

```
dpdk_fini();
```

Calls **rte_eal_destroy()** to destroy initialized EAL resources.

10. Arg parser destroy.

```
doca_argp_destroy();
```

- Free DPDK resources by call **rte_eal_cleanup()** function.

6.18.8 References

- **/opt/mellanox/doca/applications/simple_fwd_vnf/**

6.19 DOCA SNAP Virtio-fs Application Guide

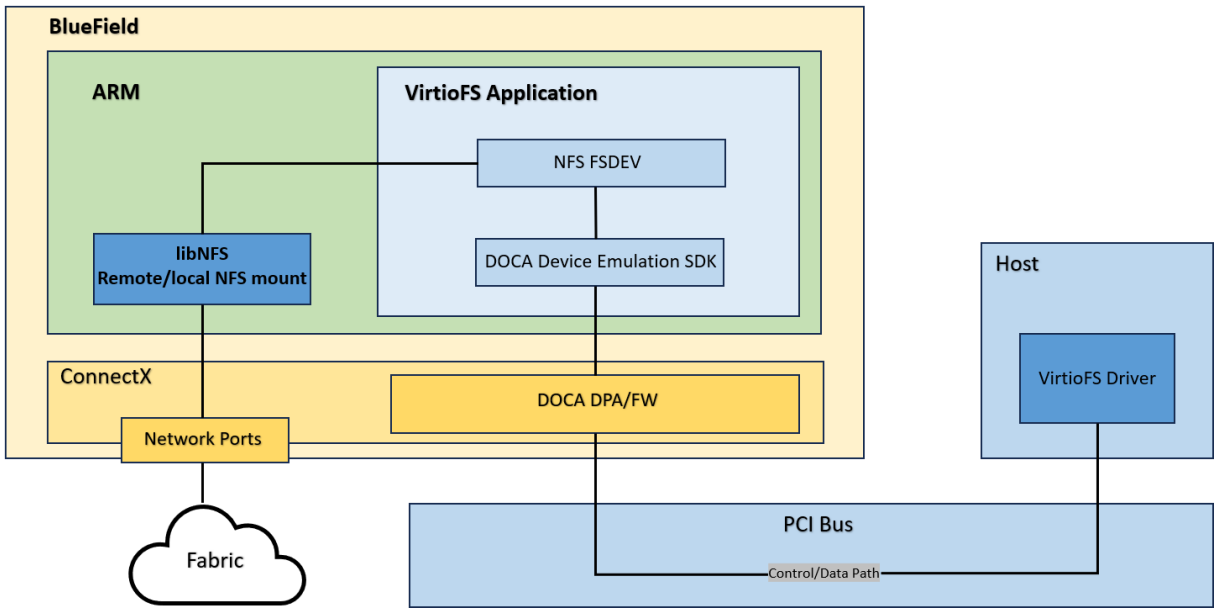
This guide describes the DOCA SNAP Virtio-fs Application which leverages DOCA DevEmu APIs and NFS backends via libnfs to provide high-performance, hardware-accelerated file system emulation.

> ⚠ This feature is currently supported at alpha level.

6.19.1 Introduction

The DOCA SNAP Virtio-fs Application demonstrates how to use the DOCA DevEmu PCI Generic API to emulate a virtio-fs PCIe function, enabling hardware-accelerated file system emulation on the BlueField DPU. This solution offloads virtio-fs handling from the host, facilitating seamless remote file access via NFS.

6.19.2 System Design



6.19.2.1 Host Side (Virtio-fs Client Driver)

- Runs a standard virtio-fs client driver in the host operating system.
- Host applications issue file system operations (e.g., open, read, write, mkdir).
- The virtio-fs driver translates these into virtio-fs protocol I/O requests.

6.19.2.2 BlueField DPU (DOCA SNAP Virtio-fs Application)

- Runs the DOCA SNAP Virtio-fs Application.
- Retrieves virtio-fs I/O requests from the host via Virtio queues.

- Decodes the requests into file operations (lookup, getattr, read, write, mkdir, etc.).
- Translates file operations into NFS requests using libnfs APIs (based on libnfs4).

6.19.2.3 Network Communication (libnfs)

- Communicates with a local or remote NFS server using libnfs over TCP.
- Executes file operations remotely on the NFS server.
- Receives responses from the NFS server and returns them to the host.

6.19.2.4 NFS Server

- Receives NFS requests (read, write, getattr, mkdir, etc.) sent by libnfs.
- Performs I/O operations on local storage.
- Sends responses back to the DPU, which forwards them to the host through virtio-fs.

6.19.2.5 End-to-End Flow

1. Host application initiates a file system operation.
2. Virtio-fs client driver sends a request to the DPU.
3. The DPU processes the request and forwards it to the NFS server via libnfs.
4. The NFS server executes the request and returns the result.
5. The DPU sends the result back to the host through virtio-fs.

This architecture offloads file system emulation from the host to the DPU, bridging virtio-fs requests to remote storage using the NFS protocol.

6.19.3 Application Architecture

Virtio-fs is a virtualization file system designed for efficient and secure file sharing between a host and guest VMs. It uses the Virtio framework to establish a high-performance communication channel, bypassing traditional network-based file system protocols to improve throughput and reduce latency.

Virtio-fs maps files and directories from a host or remote storage system into the guest's address space, achieving near-native file system performance over TCP protocol only.

Virtio-fs emulation allows this functionality to be simulated in virtualized or development environments without the need for dedicated hardware devices.

6.19.3.1 Virtio-fs Protocol Components

6.19.3.1.1 FUSE (Filesystem in Userspace)

Virtio-fs uses the FUSE protocol to forward file system operations from the guest VM to the host. Key FUSE operations include:

- LOOKUP – Resolves file names and retrieves attributes.
- GETATTR – Fetches file metadata.
- READ/WRITE – Performs I/O operations.
- MKDIR/MKNOD/UNLINK – Manages file and directory creation and deletion.

6.19.3.1.2 Virtio Queues

Virtio-fs employs request and response queues to communicate between guest and host:

- Request queue – The guest places file system operation requests.
- Response queue – The host returns operation results or error statuses.

6.19.3.2 Initialization Process

The virtio-fs initialization process involves:

1. Setting up virtio-fs request and response queues.
2. Establishing shared memory regions for efficient data transfer.
3. Negotiating feature capabilities between guest and host.
4. Mounting the shared filesystem into the guest environment.

6.19.3.3 libnfs Integration

While virtio-fs typically interfaces with local host storage, integrating libnfs enables seamless file sharing with remote NFS targets. libnfs provides a user-space NFS client implementation, allowing the DOCA SNAP Virtio-fs Application to translate virtio-fs requests into NFS protocol operations.

6.19.3.4 File Operations Workflow

1. Guest VM submits a file system request (e.g., open, read).
2. Virtio-fs forwards the request to the host via the Virtio request queue.
3. The DPU processes the request and uses libnfs to communicate with the NFS server.
4. The NFS server executes the file operation and returns the result to the DPU.
5. The DPU forwards the response to the guest VM via the Virtio response queue.

6.19.3.5 Reset and Shutdown

Proper reset and shutdown procedures ensure file system integrity and resource cleanup:

- Reset – Reinitializes Virtio queues and shared memory regions without affecting persistent data.
- Shutdown – Gracefully terminates file operations, ensuring all pending transactions are flushed and committed.

6.19.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Comch

For additional information about the used DOCA libraries, please refer to the respective programming guides.

6.19.5 Dependencies

The following software components are required to build and run the DOCA SNAP Virtio-fs Application:

- BlueField-3 DPU
- libnfs 4.0.0-1 for the FSDEV backend: https://packages.debian.org/bullseye/libnfs-dev. Install using the following command:

```
sudo apt update && sudo apt install -y libnfs-dev
```

In some setups, the following dependencies may be missing and must be installed before compiling the application:

- **libjson-c-dev**
- **libjsoncpp-dev**

6.19.5.1 Compiling the Application

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

For more information on DOCA applications, development workflows, and build guidance, see the DOCA Reference Applications documentation page.

The source code for the application is located in the following directory:

```
/opt/mellanox/doca/applications/virtiofs/
```

6.19.5.2 Compiling All Applications

All DOCA reference applications are part of a unified meson project. By default, building the project compiles all available applications.

To build all applications:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_virtiofs is created under **/tmp/build/virtiofs/**

6.19.5.3 Compiling Only the Current Application

To build only the virtio-fs application, use the following command to disable all other applications:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_virtiofs=true
ninja -C /tmp/build
```

> **doca_virtiofs** is created under **/tmp/build/virtiofs/**.

Alternatively, you can configure build flags directly in the **meson_options.txt** file rather than passing them as command-line arguments:

1. Edit the following file **/opt/mellanox/doca/applications/meson_options.txt**.
2. Modify the flags:
   - Set **enable_all_applications** to **false**
   - Set **enable_virtiofs** to **true**
3. Run the standard build commands:

   ```
   cd /opt/mellanox/doca/applications/
   meson /tmp/build
   ninja -C /tmp/build
   ```

   > **doca_virtiofs** is created under **/tmp/build/virtiofs/**

6.19.6 Running the Application

6.19.6.1 Prerequisites

- Enable the firmware configuration with virtio-fs emulation PF. A cold reboot is required after updating the firmware configuration:

  ```
  [dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s VIRTIO_FS_EMULATION_ENABLE=1 VIRTIO_FS_EMULATION_NUM_PF=1
  ```

- This application utilizes DOCA Flow and requires huge page allocation. You must ensure huge pages are properly configured on your system by allocating 2,048 pages, each 2MB in size. Execute the following commands to configure and reload the huge pages:

  ```
  [dpu] doca-hugepages config --app doca_virtiofs --size 2048 --num 2048
  [dpu] doca-hugepages reload
  ```

- To skip reads and writes to the backend file system, set the environment variable **DOCA_VIRTIOFS_USE_NULL_NFS_FSDEV=1**.
- The application supports a single static PF with hardcoded values for **num_request_queues** (32), **queue_size** (256), and **tag** (**docavirtiofs**). These values can be modified in the **virtiofs_device_create_static()** function before compiling the application.
- Configure a local or remote mounted NFS target over TCP.

- Remote NFS server example:

```
sudo apt install nfs-kernel-server
echo "/export/data 10.245.16.0/24(rw,sync,no_subtree_check,insecure)" >> /etc/exports
sudo systemctl start nfs-kernel-server.service
```

- Local NFS server example:

```
apt-get install nfs-kernel-server
mkdir /VIRTUAL
echo "/VIRTUAL 127.0.0.1(rw,no_root_squash,insecure,no_subtree_check,sync)" >> /etc/exports
service nfs-kernel-server restart
```

6.19.6.2 Application Execution

This application is provided in source form; therefore, compilation is required before execution. Refer to the "DOCA SNAP Virtio-fs Application Guide#Compiling the Application" section for build instructions.

6.19.6.2.1 Application Usage

```
Usage: doca_virtiofs [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -m, --core-mask <core_mask>       Set core mask.
  -s, --nfs-server <nfs_server>     Set NFS server.
  -e, --nfs-export <nfs_export>     Set NFS export.
  -q, --num_request_queues <num_request_queues> Set number of request queues to be used. (Range [1,254], Default:
32)
  --emu-manager <emu_manager>   Set emulation manager ibdev name. (Default: mlx5_0)
```

Use the **-h** or **--help** flag to print this usage information at runtime:

```
./doca_virtiofs -h
```

For additional information, refer to the "DOCA SNAP Virtio-fs Application Guide#Command Line Flags" section.

6.19.6.2.2 Example Command Line Execution

- To run the application on BlueField with the default values presented in this table:

```
./doca_virtiofs
```

Long Flag	Description	Default content
num-cores	Application core mask	"0x1"

Long Flag	Description	Default content
nfs-server	Local or remote NFS server IP.	"localhost"
nfs-export	Local or remote NFS export folder.	"/VIRTUAL"

- To run the application on BlueField with a specified core mask, NFS server, and export path:

```
./doca_virtiofs -m 0xff -s 1.1.1.11 -e /export/data
```

⚠ There is a known issue with hugepage memory management on Ubuntu 24.04 when using 512 MB hugepages. In this configuration, the application supports up to 5 cores only.

6.19.6.2.3 JSON-Based Deployment Mode

The application also supports JSON-based configuration for deployment. All parameters can be provided via a JSON file:

```
./doca_virtiofs --json [json_file]
```

For example:

```
./doca_virtiofs --json ./virtiofs_params.json
```

⚠ Ensure the JSON file includes the correct configuration parameters, particularly the NFS parameters required for deployment.

6.19.6.2.4 Mounting the Virtio-fs Device (Host/VM Side)

To mount the virtio-fs device from the host or guest VM:

1. Create a mount point and load the **virtiofs** module (if not already loaded):

```
mkdir /tmp/test
modprobe -v virtiofs
mount -t virtiofs docavirtiofs /tmp/test
```

2. To unmount and clean up:

```
umount /tmp/test
modprobe -rv virtiofs
```

6.19.6.3 Command Line Flags

6.19.6.3.1 General Flags

Short Flag	Long Flag	Description
`-h`	`--help`	Prints a help synopsis and exits
`-v`	`--version`	Prints program version information and exits
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.19.6.3.2 Host Program Flags

Short Flag	Long Flag/JSON Key	Description	JSON Content
`m`	`num-cores`	Application core mask	`"num-cores": "0x1",`
`s`	`nfs-server`	Local or remote NFS server IP.	`"nfs-server": "localhost",`
`e`	`nfs-export`	Local or remote NFS export folder.	`"nfs-export": "/VIRTUAL",`

6.19.6.4 Troubleshooting

Refer to the *NVIDIA BlueField Platform Software Troubleshooting Guide* for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.19.7 Application Code Flow

This section provides a comprehensive guide to implementing a virtio-fs device emulation application using the DOCA library. The DOCA Virtio-fs Reference Application demonstrates how to leverage hardware acceleration on a Data Processing Unit (DPU) to emulate a fully functional file system storage device, appearing to the host as a standard virtio-fs PCIe device.

The implementation provides hardware-accelerated filesystem emulation, utilizing DPU resources for high-performance operation processing. The multi-threaded architecture, with explicit CPU core affinity, ensures optimal resource utilization and minimizes contention between worker threads. Asynchronous processing throughout the pipeline maintains high throughput by avoiding blocking operations that could degrade system performance.

The design supports basic filesystem functionality, including file and directory operations, metadata management, and data transfers. While the reference implementation uses default configuration values and supports a single device, the underlying architecture is designed for scalability and can be extended to support multiple devices with custom configurations in production environments.

This modular approach allows developers to understand each phase of the application flow independently while appreciating how they integrate into a complete virtio-fs emulation solution using DOCA libraries and hardware acceleration.

6.19.7.1 Framework Initialization

This section outlines the essential APIs and data structures required to initialize the DOCA virtio-fs emulation framework before creating and managing virtio-fs devices.

6.19.7.1.1 Initialization Flow

1. Parse application arguments
    - Initialize the argument parser framework using **doca_argp_init()**.
    - Register application-specific parameters via a custom function (e.g., **register_application_params()**).
    - Store parsed values in a user-defined **struct application_config**.
2. Start argument parsing
    - Parse command-line arguments using **doca_argp_start()**.
    - Validate arguments and populate the configuration structure with parsed values.
3. Create logging backends
    - Configure application log backend using **doca_log_backend_create_standard()**.
    - Set up SDK internal message routing with **doca_log_backend_create_with_file_sdk()**.
    - Adjust SDK log verbosity using **doca_log_backend_set_sdk_level()**.
    - Logging configurations are maintained using **struct doca_log_backend**.
4. Create VFS configuration
    - Create a virtio-fs configuration object using **doca_devemu_vfs_cfg_create()**.
    - The returned **struct doca_devemu_vfs_cfg** will hold framework parameters and the device list.
5. Discover available devices

- Enumerate DOCA-capable hardware devices using **doca_devinfo_create_list()**.
- The function returns an array of **struct doca_devinfo** containing device properties and capabilities.

6. Filter supported devices
   - For each discovered device, verify virtio-fs emulation capability using **doca_devemu_vfs_is_default_vfs_type_supported()**.
7. Open supported devices
   - For devices supporting virtio-fs emulation, open device handles using **doca_dev_open()**.
   - This returns **struct doca_dev** handles necessary for subsequent operations.
8. Add devices to configuration
   - Register opened device handles into the virtio-fs configuration object using **doca_devemu_vfs_cfg_add_dev()**.
9. Set framework parameters
   - Configure per-request context size and other framework-specific parameters using **doca_devemu_vfs_cfg_set_vfs_fuse_req_user_data_size()**.
10. Initialize the virtio-fs framework
    - Finalize the global framework setup with **doca_devemu_vfs_init()**.
    - After this call, the framework assumes ownership of all registered devices.
11. Clean up temporary configuration objects
    - Release the temporary configuration object using **doca_devemu_vfs_cfg_destroy()**.
    - Free device information resources with **doca_devinfo_destroy_list()**, as the framework now owns the devices.

6.19.7.1.2 Key Points

- Resource Ownership – After initialization, the virtio-fs framework takes ownership of all registered device handles.
- Global Initialization – Framework initialization must be performed once per application instance.
- Thread Safety – All initialization procedures should be executed from the main thread before spawning worker threads.

6.19.7.2 Resource Management

This section describes the setup and initialization of core resources required for virtio-fs operation after the framework initialization is complete. These resources include device managers, worker threads, progress engines, memory pools, DMA contexts, and thread-local configurations necessary for high-performance virtio-fs emulation.

6.19.7.2.1 Resource Management Flow

1. Create device managers
   - Retrieve the default virtio-fs emulation type using **doca_devemu_vfs_type_create()**, which returns a **struct doca_devemu_vfs_type**.

- For each opened device, create a device manager using `doca_devemu_vfs_manager_create()`. Each manager is represented by a `struct doca_devemu_vfs_manager`.

2. Initialize manager list
   - Organize all device managers into a list structure for efficient management.
   - Use application-defined linked list operations (e.g., `SLIST_INIT`, `SLIST_INSERT_HEAD`) to build and maintain the manager list.
   - The list is defined using an application-specific `struct manager_list`.
3. Determine thread configuration
   - Calculate the number of worker threads based on the user-specified CPU core mask.
   - Use CPU enumeration functions to identify available cores and select which cores to allocate for virtio-fs worker threads.
4. Create worker threads
   - For each designated CPU core, spawn a worker thread using `pthread_create()`.
   - Bind each thread to a specific CPU core using `pthread_setaffinity_np()` to ensure core affinity and minimize resource contention.
   - Thread handles are stored in `pthread_t`, with CPU affinity settings managed via `cpu_set_t`.
5. Initialize progress engines
   - For each worker thread, create a progress engine using `doca_pe_create()`.
   - Connect application contexts to the progress engines with `doca_pe_connect_ctx()` to enable asynchronous operation handling.
   - Progress engine handles are stored in `struct doca_pe`.
6. Set up memory pools
   - Create memory pools for efficient buffer allocation and management using `doca_mpool_create()`.
   - For DMA operations, create buffer inventories using `doca_buf_inventory_create()`.
   - Memory pools and inventories are represented by `struct doca_mpool` and `struct doca_buf_inventory`, respectively.
7. Create DMA contexts
   - Initialize DMA contexts for data transfer operations using `doca_dma_create()`.
   - Configure DMA task parameters, such as memcpy settings, using `doca_dma_task_memcpy_set_conf()`.
   - DMA context handles are maintained using `struct doca_dma`.
8. Configure thread contexts
   - Set up per-thread context structures using application-defined initialization routines.
   - Assign memory pools and DMA contexts to each thread.
   - Configure thread-local storage to maintain isolation and minimize cross-thread contention.
   - Thread-specific data is managed using an application-defined `struct thread_context`.
9. Start worker threads
   - Launch all worker threads using pthread-based startup mechanisms.
   - Each thread enters its main processing loop, where it continuously polls for incoming requests and performs I/O operations.
10. Verify resource readiness

- Validate that all required resources have been properly initialized:
  - All device managers are created and registered.
  - Worker threads are running and bound to designated CPU cores.
  - Memory pools and DMA contexts are allocated and ready.
  - Progress engines are connected to their corresponding contexts.

6.19.7.2.2 Key Points

- Thread Affinity – Each worker thread must be bound to a dedicated CPU core to ensure optimal performance and minimize scheduling overhead.
- Admin Thread Role – The first worker thread (admin thread) is responsible for device lifecycle operations, such as polling control events, while other threads focus on I/O processing.
- Memory Pool Sizing – Memory pool sizes should be configured based on anticipated workload, queue depths, and expected concurrency levels to prevent allocation bottlenecks.
- Progress Engines – Each worker thread requires its own progress engine to independently handle asynchronous operations and maintain high throughput.
- Resource Isolation – Ensure that threads have isolated resources (e.g., memory pools, DMA contexts) to eliminate cross-thread contention and performance degradation.
- Error Recovery – If resource creation fails at any stage, the application must perform cleanup of already-initialized resources to avoid leaks and maintain system stability.
- Scalability – Resource allocation logic should scale with the number of configured CPU cores, allowing the application to adapt to various deployment environments.

6.19.7.3 Device Configuration

This section describes the process of creating and configuring virtio-fs device instances after the resource management infrastructure is in place. The reference application demonstrates how to emulate a virtio-fs PCIe function using hardware acceleration to fully emulate a file system storage device.

For simplicity, this implementation supports a single virtio-fs device created with default configuration values (number of queues, queue size, filesystem tag). However, the DOCA DevEmu PCI Generic API supports scalable deployments with multiple devices and fully customizable parameters.

6.19.7.3.1 Device Configuration Flow

- Create NFS backend filesystem
  - Initialize the NFS filesystem backend using an application-defined **`nfs_fsdev_create()`** function, which takes the NFS server address and export path (provided via command-line arguments).
  - The backend must be ready before creating the virtio-fs device, as it will be linked during device instantiation.

    > Backend creation details are provided in the "Backend Integration" section.
- Search for available virtio-fs functions

    - Discover available virtio-fs PCIe functions using **doca_devinfo_rep_create_list()** with the **DOCA_DEVINFO_REP_FILTER_EMULATED** filter.
    - Iterate through the list to find the first available virtio-fs function for device creation.
    - This reference implementation selects the first match for simplicity.
- Create device representors
    - Open the selected representor device using **doca_dev_rep_open()** to establish a communication channel between the DPU and the host.
    - Representor handles are managed via **struct doca_dev_rep**, while representor information is stored in **struct doca_devinfo_rep**.
- Match VUIDs to representors
    - Retrieve the Virtual Unique Identifier (VUID) for each representor using **doca_devinfo_rep_get_vuid()**.
    - Match the VUID against configured values to associate representors with specific virtio-fs virtual functions visible to the host.
- Create virtio-fs devices
    - Instantiate the virtio-fs device using **doca_devemu_vfs_dev_create()**, providing the selected representor, the admin thread's progress engine, and the initialized NFS backend.
    - The resulting device handle is maintained via **struct doca_devemu_vfs_dev**.
- Configure device attributes
    - Set device-specific parameters:
        - Filesystem tag using **doca_devemu_vfs_dev_set_tag()** (default tag in reference implementation).
        - Number of request queues using **doca_devemu_vfs_dev_set_num_request_queues()** (default value).
        - Queue size using **doca_devemu_virtio_dev_set_queue_size()** (default size).
    - In production, these parameters can be customized to meet performance and scaling requirements.
    - Configuration details are maintained in an application-defined **struct device_config**.
- Set up device callbacks
    - Register event handlers for device lifecycle management:
        - Device reset events via **doca_devemu_virtio_dev_event_reset_register()**.
        - PCIe Function Level Reset (FLR) events via **doca_devemu_pci_dev_event_flr_register()**.
        - State change notifications using **doca_ctx_set_state_changed_cb()**.
    - Application-defined callback functions handle these events appropriately.
- Configure virtio properties
    - Specify the number of I/O contexts using **doca_devemu_virtio_dev_set_num_required_running_virtio_io_ctxs()** (should match the number of worker threads).
    - Set the total number of virtqueues using **doca_devemu_virtio_dev_set_num_queues()**, including request queues and a high-priority queue for metadata operations.
- Initialize device contexts
    - Allocate per-device context structures using application-defined initialization functions.

        - Link the virtio-fs device with its assigned device manager and NFS backend.
        - These contexts maintain the association between virtio-fs emulation and underlying storage.
- Validate device configuration
    - Ensure all device parameters are correctly configured:
        - Queue sizes within hardware limits.
        - Filesystem tags properly formatted.
        - VUIDs correspond to valid virtual functions.
        - All required event callbacks are registered.
    - Validation prevents runtime misconfiguration and operational failures.
- Add devices to global list
    - Register the configured device into the application's global device list using linked list operations (**SLIST_INSERT_HEAD**).
    - Maintain counters for tracking active devices.
    - Although this reference supports a single device, the infrastructure allows scaling to multiple devices.
- Prepare for device startup
    - Set the initial device state to "stopped."
    - Store startup callback information for later use when the host virtio-fs driver initiates device activation.
    - Devices remain in the configuration state until explicitly started by host-side interactions.

6.19.7.3.2 Key Points

- Backend-first approach – The NFS backend must be created before virtio-fs device creation, as the device requires a functional storage backend during initialization.
- Function discovery – This reference uses the first available virtio-fs function. Production deployments can implement advanced selection logic based on performance or topology requirements.
- Reference implementation scope – Demonstrates single-device virtio-fs emulation with default configurations. DOCA APIs support large-scale, multi-device deployments with customizable parameters.
- Hardware acceleration – The virtio-fs PCIe function leverages hardware acceleration on the DPU for high-performance file system emulation.
- VUID association – Each virtio-fs device must be linked to a unique VUID that maps to a specific PCIe virtual function visible to the host OS.
- Admin thread execution – All device creation and configuration operations should be performed by the admin thread to ensure synchronization and thread safety.
- Queue configuration – The total number of queues includes request processing queues and a dedicated high-priority queue for handling filesystem metadata operations.
- Event handling – Correct registration of reset, FLR, and state change callbacks is essential for managing device lifecycle events triggered by the host.
- Resource validation – Comprehensive validation of device parameters ensures system stability and avoids runtime failures during activation.
- State management – Devices are initialized in a stopped state and require explicit commands to transition to an operational state when activated by the host.

- Scalability potential – Though this reference focuses on a single device, the architecture supports scalable deployments with multiple virtio-fs devices, each configurable for different use cases.

6.19.7.4 Backend Integration

This section details the setup and integration of the NFS filesystem backend, which provides the actual storage functionality behind the virtio-fs emulation. The backend acts as a bridge between FUSE operations received from the host and the underlying NFS storage system.

6.19.7.4.1 Backend Integration Flow

1. Initialize NFS client library
    - Set up the NFS client infrastructure using libnfs initialization routines.
    - This includes initializing the RPC layer and NFS protocol handlers that facilitate communication with the remote NFS server.
    - NFS client state is managed through libnfs context structures.
2. Create NFS filesystem device
    - Instantiate the NFS backend using an application-defined **`nfs_fsdev_create()`** function, providing the configured NFS server address and export path.
    - This creates the main NFS filesystem device object (**`struct nfs_fsdev`**), which encapsulates the connection to the NFS server and provides the interface for file operations.
3. Configure NFS connection parameters
    - Set connection parameters such as:
        - NFS protocol version (typically NFSv3) using **`nfs_set_version()`** .
        - RPC timeout values using **`nfs_set_timeout()`** .
        - Connection-specific settings such as buffer sizes and retry counts.
    - These parameters should be tuned based on network conditions and server performance characteristics.
4. Establish NFS server connection
    - Connect to the NFS server using **`nfs_mount()`** to mount the specified export path.
    - This step involves mount protocol negotiations and, if required, authentication procedures.
    - The connection must be fully established before initiating any file operations.
5. Set up per-thread NFS contexts
    - For each worker thread, create a dedicated NFS client context using **`nfs_fsdev_get()`** .
    - Thread-local NFS contexts enable concurrent operations without requiring synchronization across threads.
    - Per-thread state is managed through **`struct nfs_fsdev_thread_ctx`** .
6. Initialize operation mapping
    - Define a translation layer that maps FUSE operations (e.g., LOOKUP, READ, WRITE, GETATTR) to corresponding NFS RPC calls (e.g., LOOKUP3, READ3, WRITE3, GETATTR3).

- Application-defined handler functions are used to perform this mapping for each supported FUSE operation.

7. Configure asynchronous operation support
    - Implement asynchronous NFS operations using callback mechanisms.
    - NFS requests are submitted asynchronously, with completion handled via registered callbacks to ensure non-blocking execution.
    - Application-defined callback structures are used for tracking in-flight operations and handling their completion.
8. Initialize request context pools
    - Allocate memory pools for NFS request contexts to efficiently manage allocation and reuse.
    - Each NFS operation requires a context structure to track its state and manage associated resources until completion.
9. Set up error handling and recovery
    - Implement error handling mechanisms to:
        - Translate NFS error codes to their corresponding FUSE error codes.
        - Retry transient failures where applicable.
        - Recover from NFS server disconnections by attempting reconnection or triggering failover logic.
10. Register progress polling
    - Integrate progress polling functions that are periodically invoked within each worker thread's main loop.
    - These functions check for completed NFS operations and trigger their associated completion callbacks.
    - Efficient polling ensures timely processing of operations without idle wait cycles.
11. Validate backend readiness
    - Perform backend readiness checks, such as:
        - Validating connectivity by performing a stat operation on the root directory of the NFS export.
        - Ensuring that all per-thread NFS contexts are properly initialized.
        - Confirming that the FUSE-to-NFS operation mapping is complete and functional.
12. Integrate with virtio-fs device
    - Link the prepared NFS backend to the virtio-fs device instance.
    - This integration ensures that FUSE requests received by the virtio-fs emulation layer are routed to the appropriate NFS backend handlers for processing.

6.19.7.4.2 Key Points

- NFS Protocol Support – The backend typically uses NFSv3 for broad compatibility, though NFSv4 can be employed for advanced features.
- Thread-Safe Operations – Each worker thread must operate with its own NFS client context to avoid synchronization overhead and maintain parallelism.
- Asynchronous Processing – All NFS operations are performed asynchronously, preventing blocking behavior and ensuring high throughput.
- Error Translation – Correct mapping between NFS and FUSE error codes is essential for consistent behavior on the host side.

- Connection Resilience – The backend must gracefully handle NFS server disconnections, implementing automatic reconnection or recovery procedures.
- Performance Tuning – NFS client parameters (e.g., timeouts, buffer sizes) should be tuned based on workload characteristics and network conditions.
- Memory Management – Efficient allocation and reuse of request context structures are critical for performance at scale.
- Progress Polling – Regular polling for completed NFS operations ensures timely request completion and system responsiveness.
- Operation Mapping – A comprehensive FUSE-to-NFS operation mapping ensures full filesystem functionality for the host.
- Backend Abstraction – The architecture supports backend abstraction, enabling future support for alternative storage backends (e.g., local POSIX FS, object storage) through the same interface.

6.19.7.5 Request Handler Registration

This section describes the process of registering FUSE operation handlers to process filesystem requests from the host, setting up DMA contexts for efficient data transfers, and configuring asynchronous callbacks for request completion and error handling.

6.19.7.5.1 Request Handler Registration Flow

1. Create VFS I/O contexts
    - Initialize I/O contexts for each worker thread using **`doca_devemu_vfs_io_create()`**.
    - Each worker thread receives its own **`struct doca_devemu_vfs_io`** handle to process FUSE requests independently, avoiding cross-thread contention.
2. Register FUSE operation handlers
    - For every supported FUSE operation, register corresponding handler functions:
        - Lookup: **`doca_devemu_vfs_io_event_vfs_fuse_lookup_req_handler_register()`**
        - Read: **`doca_devemu_vfs_io_event_vfs_fuse_read_req_handler_register()`**
        - Write: **`doca_devemu_vfs_io_event_vfs_fuse_write_req_handler_register()`**
        - Getattr, Setattr, Create, Mkdir, etc.
    - Each handler translates the incoming FUSE request into corresponding backend operations (e.g., NFS RPC calls) and manages the request lifecycle.
3. Set up DMA contexts for data transfers
    - Create DMA contexts for each I/O context using **`doca_dma_create()`**.
    - Configure DMA memcpy task parameters using **`doca_dma_task_memcpy_set_conf()`**, including completion and error callbacks.
    - DMA contexts are essential for enabling efficient zero-copy data transfers between host memory and DPU memory.
4. Configure DMA completion callbacks
    - Register completion and error handlers for DMA operations using **`doca_dma_task_memcpy_set_conf()`**.
    - Application-defined callback functions will handle the completion of data transfers and resume processing of the originating FUSE requests.
5. Connect I/O contexts to progress engines

- Associate each I/O context with its corresponding thread's progress engine using **doca_pe_connect_ctx()**.
- This ensures that I/O operations are executed within the correct thread context, maintaining thread affinity and balanced load distribution.

6. Initialize request context management
   - Set up request context pools using application-defined allocation mechanisms.
   - Each incoming FUSE request is assigned a context structure that tracks its state, associated DMA operations, and completion handlers.
7. Configure asynchronous operation handling
   - FUSE request handlers should submit backend operations (e.g., NFS RPC calls) asynchronously.
   - Completion callbacks are registered to be invoked when backend operations complete, enabling concurrent request processing without blocking worker threads.
8. Set up request queuing and scheduling
   - Implement queuing mechanisms using application-defined structures to manage pending requests during high-load conditions.
   - Queuing helps maintain fairness and prevents resource exhaustion by controlling the number of active in-flight operations.
9. Register error handling callbacks
   - Configure error handling routines to:
     - Translate backend error codes to corresponding FUSE error responses.
     - Handle DMA operation failures gracefully.
     - Ensure proper cleanup of request contexts in failure scenarios.
10. Initialize request statistics and monitoring
    - Set up performance monitoring structures to track:
      - Request counts.
      - Latency metrics.
      - Error rates.
      - Other relevant performance indicators.
    - This data aids in debugging, optimization, and capacity planning.
11. Configure request validation
    - Implement input validation routines to verify:
      - Request parameters.
      - Validity of file handles.
      - Access permissions.
      - Format and correctness of incoming requests.
    - Early validation helps prevent crashes, resource leaks, and security vulnerabilities.
12. Start I/O context processing
    - Activate each I/O context using **doca_ctx_start()**.
    - Once started, I/O contexts will begin accepting and processing FUSE requests from the host, dispatching them to the appropriate registered handlers.

6.19.7.5.2 Key Points

- Complete Operation Coverage – All relevant FUSE operations must have registered handlers to ensure full filesystem functionality on the host.

- Thread-Specific Contexts – Each worker thread must have its own VFS I/O context to avoid contention and maintain scalability.
- DMA Integration – Proper setup of DMA contexts and completion callbacks is essential for efficient data movement during read/write operations.
- Asynchronous Processing – Handlers should submit backend operations asynchronously to maximize throughput and avoid blocking threads.
- Robust Error Handling – Comprehensive error translation, recovery, and cleanup routines ensure stable and predictable application behavior.
- Request Lifecycle Management – Every request must be tracked from reception to completion, ensuring proper resource allocation and cleanup.
- Efficient Resource Management – Request contexts and DMA buffers should be managed via efficient allocation and reuse strategies to support high workloads.
- Performance Monitoring – Request statistics provide visibility into system performance, aiding in bottleneck identification and optimization efforts.
- Input Validation – Validating incoming requests early prevents invalid operations and mitigates security risks.
- Progress Engine Integration – All I/O contexts must be connected to progress engines to ensure correct and efficient operation processing.
- Callback Management – Completion and error callbacks for all operations must be correctly registered and invoked to maintain request flow and ensure system integrity.

6.19.7.6 Device Lifecycle Management

This section describes the procedures for starting device emulation, handling device state transitions, managing reset events, and implementing error recovery mechanisms to ensure robust and continuous virtio-fs device operation throughout its lifecycle.

6.19.7.6.1 Device Lifecycle Management Flow

1. Start VFS device context
    - Begin device emulation by starting the main virtio-fs device context using **`doca_ctx_start()`**.
    - This transitions the device from IDLE to STARTING state and initiates the device initialization sequence.
    - The operation is asynchronous, with completion signaled through state change callbacks.
2. Start I/O contexts
    - Activate all worker thread I/O contexts using **`doca_ctx_start()`** for each **`struct doca_devemu_vfs_io`** created during request handler registration.
    - All I/O contexts must be started to begin processing FUSE requests from the host.
    - Contexts are started in parallel across all worker threads.
3. Monitor device state transitions
    - Use registered state change callbacks (**`doca_ctx_set_state_changed_cb()`**) to track device state transitions:
        - IDLE → STARTING → RUNNING
    - These callbacks update application-level state tracking and trigger actions appropriate to each transition.

4. Handle device startup completion
    - When the device reaches RUNNING state:
        - Update device status.
        - Log successful startup.
        - Invoke any registered startup completion callbacks.
    - At this point, the device is fully operational and visible to the host system.
5. Manage device visibility to host
    - Upon entering RUNNING state, the virtio-fs device is exposed to the host as a PCIe device.
    - The host OS can now mount the filesystem and start sending FUSE requests for processing.
6. Handle device reset events
    - Process device reset requests from the host using the registered reset event callback (**doca_devemu_virtio_dev_event_reset_register()**).
    - Reset handling includes:
        - Stopping the device.
        - Clearing internal state.
        - Re-initializing and restarting the device to resume normal operation.
7. Handle function-level reset (FLR)
    - Process PCIe Function Level Reset (FLR) events using the registered FLR callback (**doca_devemu_pci_dev_event_flr_register()**).
    - FLR handling requires stopping all device activity and reinitializing the device to a clean state to maintain PCIe protocol compliance.
8. Implement graceful device stop
    - Use **doca_ctx_stop()** to gracefully shut down device operation.
    - The shutdown sequence involves:
        - Stopping all I/O contexts first.
        - Then stopping the main device context.
        - Ensuring all pending operations are either completed or safely aborted.
9. Manage device state during stop
    - Track device shutdown transitions using state change callbacks:
        - RUNNING → STOPPING → IDLE
    - Each transition requires specific cleanup actions and resource deallocation to ensure a clean shutdown.
10. Handle error recovery
    - Implement error recovery mechanisms to detect device failures and perform recovery actions:
        - Attempt soft recovery where possible.
        - Escalate to a full device restart if necessary.
    - These procedures ensure device availability during transient failures.

    > ⚠ The application checks files under **/tmp/nfs_map/** to determine whether recovery is needed. If recovery is not required, the user should delete the **/tmp/nfs_map/** directory.

11. Coordinate multi-context lifecycle

- Synchronize lifecycle operations across the main device context and all associated I/O contexts.
- Application-defined coordination mechanisms should be used to ensure proper sequencing of start/stop operations and avoid race conditions.

12. Cleanup on device removal
    - On device removal or application shutdown:
        - Ensure all contexts are properly stopped.
        - Free all allocated resources.
        - Close backend connections.
        - Cleanly remove the device from the host system to prevent resource leaks or dangling devices.

6.19.7.6.2 Key Points

- Asynchronous Operations – Device start and stop procedures are asynchronous; completion is monitored via state change callbacks.
- State Synchronization – Synchronizing the state between the main device context and multiple I/O contexts is essential for correct and reliable operation.
- Host Visibility – The device becomes visible to the host only after successfully transitioning to **RUNNING** state.
- Reset Handling – Proper handling of both VirtIO resets and PCIe FLR events is required to maintain compatibility with standard host drivers.
- Graceful Shutdown – A structured stop sequence ensures all in-flight operations are safely completed or canceled, preventing data loss or corruption.
- Error Resilience – Robust error detection and recovery procedures help maintain device availability in the event of transient or recoverable errors.
- Resource Cleanup – All resources (contexts, memory pools, DMA buffers) must be freed upon shutdown to prevent memory leaks and resource exhaustion.
- Thread Coordination – Lifecycle operations (start/stop) must be carefully coordinated across multiple worker threads to ensure ordered and race-free execution.
- Callback Management – State change, reset, and FLR callbacks are critical for tracking device status and triggering necessary actions throughout the lifecycle.
- Host Driver Compatibility – Proper lifecycle management is essential to maintaining compatibility with standard virtio-fs drivers on the host side.
- Startup Sequencing – Device and I/O contexts must be started in the correct order to ensure proper initialization and readiness.
- Recovery Procedures – Failed startup or runtime errors should trigger cleanup routines and proper error reporting to maintain system stability.

6.19.7.7 Runtime Operations

This section describes the core operational phase where the application processes incoming FUSE requests from the host, executes corresponding filesystem operations via the NFS backend, and manages data transfers between the host and storage with hardware-accelerated efficiency.

6.19.7.7.1 Runtime Operations Flow

1. Receive FUSE requests from host

    - Incoming filesystem requests from the host are received through the virtio-fs device.
    - The appropriate FUSE operation handler is invoked based on the request's operation code (e.g., LOOKUP, READ, WRITE, GETATTR).
    - Requests are dispatched to their corresponding handler functions for further processing.
2. Parse and validate FUSE requests
    - Extract request parameters using:
        - `struct fuse_in_header` for common fields.
        - Operation-specific structures (e.g., `struct fuse_read_in`, `struct fuse_write_in`) for request-specific parameters.
    - Validate fields such as file handles, offsets, sizes, and permissions to ensure requests are well-formed before processing.
3. Allocate request context
    - Allocate a request context structure from a pre-allocated pool to manage the request's lifecycle.
    - The context tracks request state, intermediate results, associated DMA operations, and completion callbacks.
4. Translate FUSE operations to NFS calls
    - Map FUSE operations to corresponding NFS RPC calls:
        - FUSE_LOOKUP → NFS LOOKUP3
        - FUSE_READ → NFS READ3
        - FUSE_WRITE → NFS WRITE3
        - Parameter translation ensures compatibility between FUSE and NFS semantics.
5. Submit operations to NFS backend
    - Submit NFS operations asynchronously using thread-specific NFS contexts.
    - Asynchronous submission allows multiple operations to be in-flight concurrently without blocking worker threads.
6. Handle data transfers for read/write operations
    - For read operations:
        - DMA is used to transfer data from NFS buffers to host memory.
    - For write operations:
        - DMA transfers data from host memory to NFS buffers before sending it to the NFS server.
    - DMA enables zero-copy data movement, reducing CPU overhead and improving performance.
7. Process NFS operation completion
    - Upon NFS operation completion, registered callbacks are invoked.
    - Callbacks process NFS responses, extract result data, and prepare corresponding FUSE responses for the host.
8. Manage concurrent operations
    - Per-thread request queues and asynchronous processing allow worker threads to handle multiple requests concurrently.
    - Non-blocking operations ensure high throughput even under heavy load.
9. Handle operation errors
    - Failed operations are processed by translating NFS error codes to appropriate FUSE error responses.

- Common errors include:
  - ENOENT (file not found)
  - EACCES (permission denied)
  - EIO (I/O errors)
- Errors are propagated back to the host to ensure correct application behavior.

10. Complete FUSE responses
    - Responses are sent back to the host using **doca_virtiofs_req_complete()** or similar APIs.
    - Operation results are packaged into FUSE response structures and delivered through the virtio-fs interface to the host driver.
11. Update request statistics
    - Application-defined statistics collection tracks:
      - Request counts.
      - Latency metrics.
      - Error rates.
      - Throughput measurements.
    - Continuous monitoring helps identify bottlenecks and optimize system performance.
12. Manage memory and resource cleanup
    - After request completion:
      - Release DMA buffers.
      - Free or recycle request contexts.
      - Ensure all resources are properly cleaned up to prevent memory leaks during prolonged operation.

6.19.7.7.2 Key Points

- High Concurrency – Multiple requests are processed simultaneously across worker threads, ensuring maximum throughput and low latency.
- Zero-Copy Transfers – DMA operations enable efficient data transfers between host and DPU memory without involving the CPU.
- Asynchronous Processing – All filesystem and data operations are handled asynchronously to prevent thread blocking and maintain system responsiveness.
- Error Propagation – Accurate translation of NFS errors to FUSE responses is essential for correct host-side application behavior.
- Resource Efficiency – Request contexts and DMA buffers are pooled and reused to minimize allocation overhead and support high workloads.
- Thread Isolation – Each worker thread processes requests independently, avoiding shared resource contention and synchronization bottlenecks.
- Performance Monitoring – Real-time statistics collection provides visibility into system health, enabling proactive performance tuning.
- Memory Management – Proper allocation and cleanup of resources are critical to prevent memory leaks during long-running operations.
- Protocol Compliance – All FUSE responses must conform to the virtio-fs specification to ensure compatibility with standard host drivers.
- Load Balancing – Request distribution across worker threads balances workload and optimizes hardware utilization.

- Data Consistency – Proper ordering and synchronization of requests ensure data consistency between the host and the NFS backend.
- Scalability – The architecture supports scaling to handle high request rates and large numbers of concurrent operations across multiple worker threads.

6.19.7.8 Cleanup and Shutdown

This section describes the procedures for gracefully shutting down the application, freeing allocated resources, closing active connections and contexts, and ensuring a clean termination without leaving resource leaks or dangling system components.

6.19.7.8.1 Cleanup and Shutdown Flow

1. Initiate graceful shutdown –
    - On receiving termination signals (SIGINT, SIGTERM), signal handlers set shutdown flags to begin the termination process.
    - The shutdown sequence ensures that all in-flight operations are allowed to complete before services are stopped.
2. Stop accepting new requests
    - Disable the acceptance of new FUSE requests.
    - Request handlers will reject incoming operations with appropriate error responses while allowing ongoing operations to complete.
3. Drain pending operations
    - Wait for all in-flight operations to complete:
        - NFS operations.
        - DMA transfers.
        - Outstanding FUSE requests.
    - Ensure all request contexts are returned to their respective pools before proceeding.
4. Stop VFS I/O contexts
    - Invoke **doca_ctx_stop()** on each VFS I/O context associated with worker threads.
    - These operations are asynchronous; completion is monitored through state change callbacks.
    - Stopping I/O contexts halts request processing and triggers per-thread resource cleanup.
5. Stop main VFS device context
    - Stop the primary device context using **doca_ctx_stop()**.
    - The device transitions from RUNNING → STOPPING → IDLE, becoming invisible to the host.
    - Device-level operations are halted.
6. Cleanup DMA resources
    - Free DMA contexts using **doca_dma_destroy()** for each allocated DMA context.
    - Ensure any pending DMA operations are properly finalized and associated memory allocations are released.
7. Destroy VFS I/O contexts
    - Free I/O contexts for each worker thread using **doca_devemu_vfs_io_destroy()**.
    - This step ensures cleanup of per-thread processing resources.
8. Close NFS backend connections

- Unmount NFS exports and destroy NFS client contexts:
  - Use `nfs_umount()` to unmount NFS paths.
  - Use `nfs_destroy_context()` to release NFS client-side resources.
- Clean up per-thread NFS contexts and connection pools to ensure backend disconnection.

9. Stop worker threads
   - Terminate worker threads gracefully:
     - Use `pthread_join()` to wait for threads to exit cleanly.
     - Use `pthread_cancel()` for unresponsive threads if necessary.
   - Ensure no threads remain active before continuing with shared resource deallocation.
10. Destroy progress engines
    - Free progress engines using `doca_pe_destroy()` for each progress engine created during initialization.
    - This ensures proper cleanup of asynchronous operation handling infrastructure.
11. Free memory pools and inventories
    - Destroy memory pools using `doca_mpool_destroy()` and buffer inventories using `doca_buf_inventory_destroy()`.
    - This step returns all allocated buffers and memory resources to the system.
12. Close devices and cleanup framework
    - Close all opened devices and representors:
      - Use `doca_dev_close()` for device handles.
      - Use `doca_dev_rep_close()` for representor handles.
    - Finalize VFS framework cleanup to ensure hardware resources are fully released.
13. Destroy device and manager resources
    - Free application-specific resources:
      - Device lists.
      - Manager structures.
      - Configuration objects.
    - Clean up linked lists, hash tables, and other dynamic data structures used in the application.
14. Cleanup argument parser and logging infrastructure
    - Destroy the argument parser with `doca_argp_destroy()`.
    - Tear down logging backends to clean up logging resources and ensure a clean exit.
15. Verify resource cleanup
    - Perform a final validation step to confirm:
      - All memory allocations are freed.
      - All contexts are destroyed.
      - No background threads or pending operations remain active.
    - Check for potential memory leaks or orphaned resources.

6.19.7.8.2 Key Points

- Graceful Shutdown – Ensure in-flight operations are completed before shutting down to avoid data corruption or loss.
- Ordered Cleanup – Resources must be released in the reverse order of their creation, respecting dependencies between components.

- Signal Handling – Proper handling of external termination signals ensures orderly shutdown procedures are triggered.
- Thread Synchronization – Worker threads must be fully stopped and joined before deallocating shared resources.
- Resource Tracking – A systematic cleanup approach is critical to prevent resource leaks and ensure system stability after termination.
- Error Handling During Cleanup – Cleanup operations should proceed even if individual cleanup steps encounter errors, ensuring best-effort resource release.
- State Verification – Confirm that all contexts and resources have been freed and no residual state remains.
- Backend Cleanup – NFS backend connections and contexts must be properly closed to prevent lingering open connections.
- Device Visibility – Devices should be made invisible to the host before initiating cleanup to avoid host-side errors.
- Timeout Handling – Ensure that cleanup operations have timeouts to prevent the application from hanging during shutdown.
- Dependency Management – Cleanup operations must respect dependencies, ensuring no premature deallocation of dependent resources.
- Memory Leak Prevention – All dynamically allocated memory (DMA buffers, request contexts, etc.) must be freed.
- Hardware Resource Release – All device handles, representors, and associated contexts must be properly closed to return hardware resources to the system.
- Framework Cleanup – DOCA framework resources must be de-initialized to restore the system to a clean state.
- Logging and Monitoring – Shutdown progress and any errors during cleanup should be logged to aid in debugging and post-mortem analysis.

6.19.8 FUSE Operations Support Summary

The following table summarizes the FUSE operations supported by the virtio-fs reference application, along with their implementation status and brief descriptions:

FUSE Operation	Status	Description
LOOKUP	✅ Supported	Look up a directory entry by name and return file attributes.
FORGET	✅ Supported	Forget an inode (decrease reference count).
GETATTR	✅ Supported	Get file attributes (similar to `stat` syscall).
SETATTR	✅ Supported	Set file attributes (`chmod` , `chown` , `truncate` , etc.).
READLINK	❌ Not Implemented	Read the target of a symbolic link.
SYMLINK	❌ Not Implemented	Create a symbolic link.
MKNOD	✅ Supported	Create a file node (regular file, device file, etc.).
MKDIR	✅ Supported	Create a directory.
UNLINK	✅ Supported	Remove a file.
RMDIR	❌ Not Implemented	Remove a directory.

FUSE Operation	Status	Description
RENAME	❌ Not Implemented	Rename or move a file or directory.
LINK	❌ Not Implemented	Create a hard link to a file.
OPEN	✅ Supported	Open a file and return a file handle.
READ	✅ Supported	Read data from a file.
WRITE	✅ Supported	Write data to a file.
STATFS	❌ Not Implemented	Retrieve filesystem statistics (disk space, etc.).
RELEASE	✅ Supported	Close a file handle.
FSYNC	❌ Not Implemented	Synchronize file contents to storage.
SETXATTR	❌ Not Implemented	Set extended file attributes.
GETXATTR	❌ Not Implemented	Get extended file attributes.
LISTXATTR	❌ Not Implemented	List extended file attributes.
REMOVEXATTR	❌ Not Implemented	Remove extended file attributes.
FLUSH	❌ Not Implemented	Flush cached data (called on file close).
INIT	✅ Supported	Initialize filesystem connection.
OPENDIR	✅ Supported	Open a directory for reading.
READDIR	✅ Supported	Read directory entries.
RELEASEDIR	✅ Supported	Close a directory handle.
FSYNCDIR	❌ Not Implemented	Synchronize directory contents.
GETLK	❌ Not Implemented	Test for file locks.
SETLK	❌ Not Implemented	Set non-blocking file locks.
SETLKW	❌ Not Implemented	Set blocking file locks.
ACCESS	❌ Not Implemented	Check file access permissions.
CREATE	❌ Not Implemented	Atomically create and open a file.
INTERRUPT	❌ Not Implemented	Interrupt a pending operation.
BMAP	❌ Not Implemented	Map file block to device block.
IOCTL	❌ Not Implemented	Device-specific input/output control.
POLL	❌ Not Implemented	Poll for I/O readiness.
FALLOCATE	❌ Not Implemented	Preallocate space for a file.
DESTROY	❌ Not Implemented	Cleanup filesystem connection.
NOTIFY_REPLY	❌ Not Implemented	Reply to kernel notification.
BATCH_FORGET	❌ Not Implemented	Batch version of FORGET.
READDIRPLUS	✅ Supported	Read directory entries with attributes.
RENAME2	❌ Not Implemented	Extended rename with flags.
COPY_FILE_RANGE	❌ Not Implemented	Copy data between files efficiently.
SETUPMAPPING	❌ Not Implemented	Set up DAX (Direct Access) memory mapping.
REMOVEMAPPING	❌ Not Implemented	Remove DAX memory mapping.

FUSE Operation	Status	Description
SYNCFS	❌ Not Implemented	Synchronize entire filesystem.
LSEEK	❌ Not Implemented	Seek to a position within a file.
TMPFILE	❌ Not Implemented	Create an unnamed temporary file.
STATX	❌ Not Implemented	Extended file status query.

6.20 DOCA Storage Applications

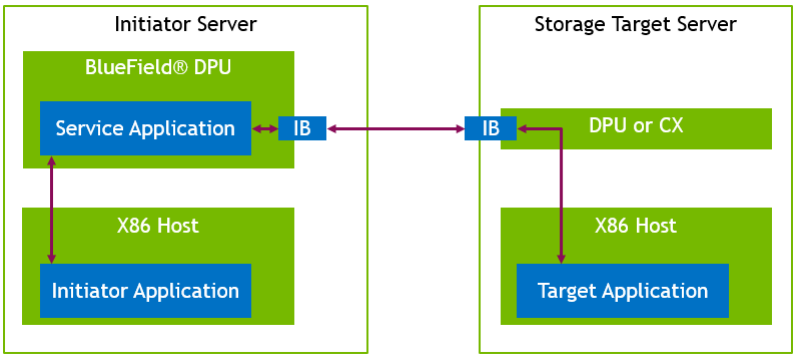
This page outlines NVIDIA DOCA storage applications that demonstrate how to develop data storage implementations on the NVIDIA® BlueField® platform.

6.20.1 Introduction

The DOCA storage applications serve as a reference to demonstrate how a block storage solution can be developed using the DOCA framework on the NVIDIA BlueField platform.

These applications are simplified to make the flow and overall process easy to understand. A real-world storage solution involves significantly more complexity, and solutions must be tailored to specific use cases, which is not attempted in this reference design.

The general architecture involves an initiator host application sending storage requests to a local DPU application. The DPU then interfaces with a remote storage target using RDMA, orchestrating the storage and retrieval of data directly to or from the initiator application's memory.



6.20.2 Applications

The storage reference consists of several applications that can be orchestrated in different combinations to support varying storage approaches and implementations. Each application is described at a high level below, followed by detailed use case examples highlighting how these applications can be combined.

6.20.2.1 doca_storage_initiator_comch

The doca_storage_initiator_comch application serves two purposes. First, it acts as the user/client of the storage reference application that controls the execution flow. Second, it includes benchmarking capabilities to demonstrate the achievable performance of the storage solution. The application supports the following tests:

- Read throughput performance test
- Write throughput performance test
- Read-only data validation test
- Write-then-read data validation test

6.20.2.2 doca_storage_target_rdma

The doca_storage_target_rdma application provides a simple storage backend by utilizing a block of memory instead of interfacing with physical non-volatile storage. It includes the following features:

- Configurable block count
- Configurable block size
- Import of initial data from a text file
- Import of initial data from a binary format (.sbc)

6.20.2.3 doca_storage_comch_to_rdma_zero_copy

The doca_storage_comch_to_rdma_zero_copy application serves as a bridge between the initiator and a single storage target. It facilitates the exchange of the control path messages and data path messages between the initiator and the target. The data being exchanged between the initiator and the target is never visible in this application as is exchanged directly between the initiator and the target via RDMA which is it is referred to as zero copy. It offers the following features:

- Facilitates the zero copy transfer of data from initiator to target

6.20.2.4 doca_storage_comch_to_rdma_gga_offload

The doca_storage_comch_to_rdma_gga_offload application serves as a bridge between the initiator and three storage targets. It is active in the transfer of data and uses **doca_libs** to accelerate some data operations transparently to the initiator and targets. It offers the following features:

- Transparent data redundancy and error correction
- Inline compression and decompression of stored data
- Inline creation of redundancy data
- Inline recovery of lost data blocks
- Configuration of how frequently a simulated data loss occurs

6.20.2.5 doca_storage_gga_offload_sbc_generator

The doca_storage_gga_offload_sbc_generator application is used to generate Storage Binary Content (.sbc) files. These files are consumed by the **doca_storage_target_rdma** application when it is used with the **doca_storage_comch_to_rdma_gga_offload** application.

Using a user-supplied data file, the generator:

1. Splits the input data into chunks.
2. Compresses each chunk.
3. Creates error correction blocks for each compressed block.

The output is stored in three files (named by the user), which can be used to initialize the storage with valid data upon startup:

- Data1 file (**data_1**)
- Data2 file (**data_2**)
- Data-parity file (**data_p**)

6.20.3 Use Case 1: Zero Copy

This use case demonstrates a simple, hardware-accelerated storage solution that avoids unnecessary data copying by using the DOCA Comch, DOCA Core, and DOCA RDMA libraries. Data is transferred directly between the initiator's memory and the storage target's memory via DOCA RDMA. The initiator has no awareness of the target; it only communicates with the service.

6.20.3.1 Application Components

This use case involves three distinct applications:

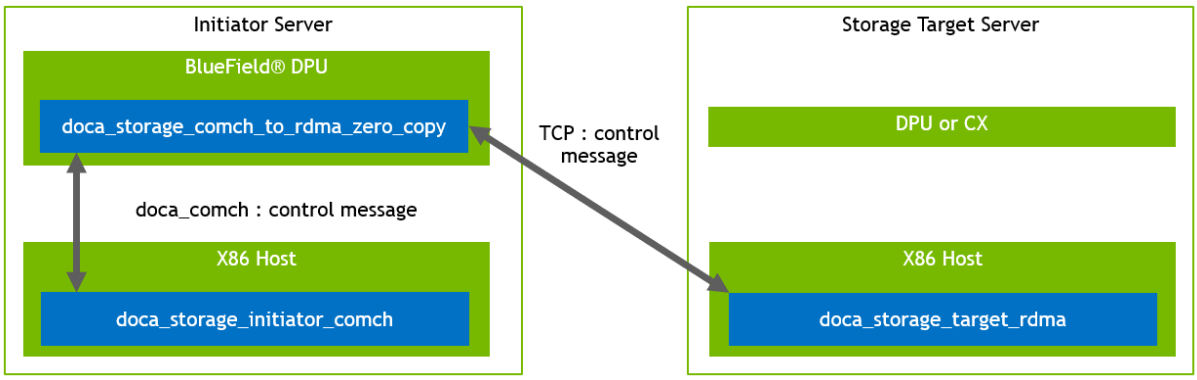
- Initiator (**doca_storage_initiator_comch**): A client/benchmark application that sends read and write requests to the storage backend.
- Service (**doca_storage_comch_to_rdma_zero_copy**): A bridge that abstracts the target's details from the initiator. It relays IO messages between the two.
- Target (**doca_storage_target_rdma**): A memory-based storage application that executes RDMA operations to fulfill the initiator's requests.

6.20.3.2 Operational Flow

The use case operates in two main phases.

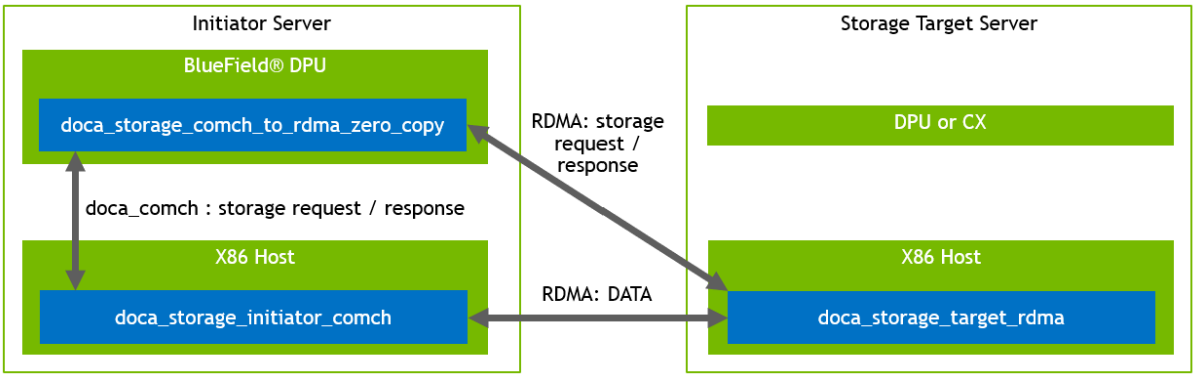
6.20.3.2.1 Setup Phase

The applications first use DOCA Comch and a TCP socket to register with each other, share configuration, and prepare the data path operations.



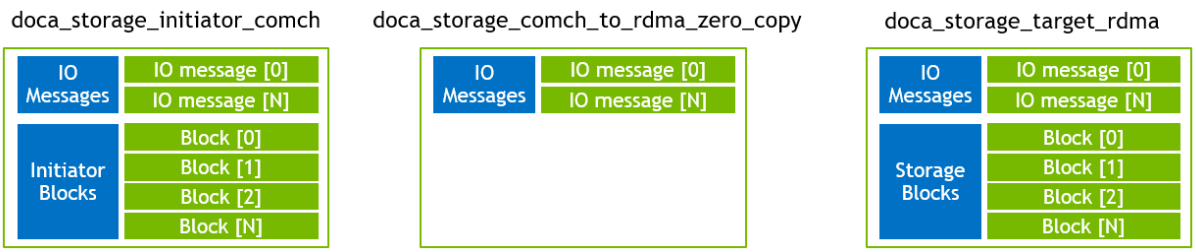
6.20.3.2.2 Data Path Phase

Once configured, the data path contexts (**doca_comch_producer** , **doca_comch_consumer** , and **doca_rdma**) are used to perform the actual storage requests and send completion responses.



6.20.3.3 Memory Layout

The memory layout is simple: both the initiator and the target allocate memory regions partitioned into blocks. All RDMA read/write operations transfer data directly between these memory regions.



The number of blocks on the initiator and target does not need to match, but their block size must be identical.

6.20.3.4 Data Flow

The data flow for a storage operation is as follows:

1. The Initiator sends a read/write IO request to the Service (via **doca_comch_producer**).
2. The Service receives the request (via **doca_comch_consumer**).
3. The Service forwards the request to the Target (via **doca_rdma**).
4. The Target performs the RDMA read or write operation, transferring data directly between its memory and the Initiator's memory.
5. The Target sends an IO response (completion) to the Service (via **doca_rdma**).
6. The Service receives the IO response.
7. The Service forwards the response to the Initiator (via **doca_comch_producer**).
8. The Initiator receives the final response (via **doca_comch_consumer**).

6.20.3.5 Running the Zero Copy Use Case

Assumptions:

- The service and storage applications can communicate via TCP
- The service is deployed on a BlueField-3 DPU
- Full line-rate RDMA data transfer is possible between service and storage

The examples below use CPU 0. Users should ensure that this CPU is suitable, considering NUMA configuration, pinned cores, kernel interrupts, etc.

Run the applications in the following order:

1. **doca_storage_target_rdma**
2. **doca_storage_comch_to_rdma_zero_copy**
3. **doca_storage_initiator_comch**

Replace the following placeholders with actual values:

- **{{INITIATOR_DEV_PCI_ADDR}}**
- **{{STORAGE_DEV_PCI_ADDR}}**
- **{{STORAGE_IP_ADDR}}**
- **{{STORAGE_TCP_PORT}}**

6.20.3.5.1 Run Write Data Validity Test

1. Run the storage target:

```
doca_storage_target_rdma \
    -d {{STORAGE_DEV_PCI_ADDR}} \
    --listen-port {{STORAGE_TCP_PORT}} \
    --block-size 4096 \
    --block-count 1024 \
    --cpu 0
```

2. Run the zero copy service:

```
doca_storage_comch_to_rdma_zero_copy \
    -d 03:00.0 \
    -r {{INITIATOR_DEV_PCI_ADDR}} \
    --storage-server {{STORAGE_IP_ADDR}}:{{STORAGE_TCP_PORT}} \
    --cpu 0
```

3. Run the initiator:

```
doca_storage_initiator_comch \
    -d {{INITIATOR_DEV_PCI_ADDR}} \
    --execution-strategy read_write_data_validity_test \
    --run-limit-operation-count 10000 \
    --cpu 0
```

6.20.3.5.2 Run Write Throughput Test

1. Repeat step 1 from the previous section.
2. Repeat step 2 from the previous section.
3. Run the initiator:

```
doca_storage_initiator_comch \
    -d {{INITIATOR_DEV_PCI_ADDR}} \
    --execution-strategy write_throughput_test \
    --run-limit-operation-count 1000000 \
    --cpu 0
```

6.20.3.5.3 Run Read Data Validity Test

The file provided to the storage and initiator should have the same content, and a size that matches **the storage block-size * the storage block-count**.

1. Run the storage target:

```
doca_storage_target_rdma \
    -d {{STORAGE_DEV_PCI_ADDR}} \
    --listen-port {{STORAGE_TCP_PORT}} \
    --block-size 4096 \
    --block-count 1024 \
    --binary-content storage_content.txt \
    --cpu 0
```

2. Run the zero copy service:

```
doca_storage_comch_to_rdma_zero_copy \
    -d 03:00.0 \
    -r {{INITIATOR_DEV_PCI_ADDR}} \
    --storage-server {{STORAGE_IP_ADDR}}:{{STORAGE_TCP_PORT}} \
    --cpu 0
```

3. Run the initiator:

```
doca_storage_initiator_comch \
    -d {{INITIATOR_DEV_PCI_ADDR}} \
    --execution-strategy read_only_data_validity_test \
    --storage-plain-content storage_content.txt \
    --run-limit-operation-count 10000 \
    --cpu 0
```

6.20.3.5.4 Run Read Throughput Test

1. Repeat step 1 from the previous section.
2. Repeat step 2 from the previous section.
3. Run the initiator:

```
doca_storage_initiator_comch \
    -d {{INITIATOR_DEV_PCI_ADDR}} \
    --execution-strategy read_throughput_test \
    --run-limit-operation-count 1000000 \
    --cpu 0
```

6.20.4 Use Case 2: GGA Offload

This use case demonstrates how to leverage hardware acceleration for a storage solution that provides transparent compression, recovery, and data mirroring. It uses a combination of the DOCA Comch, DOCA Core, DOCA RDMA, DOCA Erasure Coding and DOCA Compress libraries.

6.20.4.1 Architecture and Components

The GGA offload use case involves three applications running as five distinct instances:

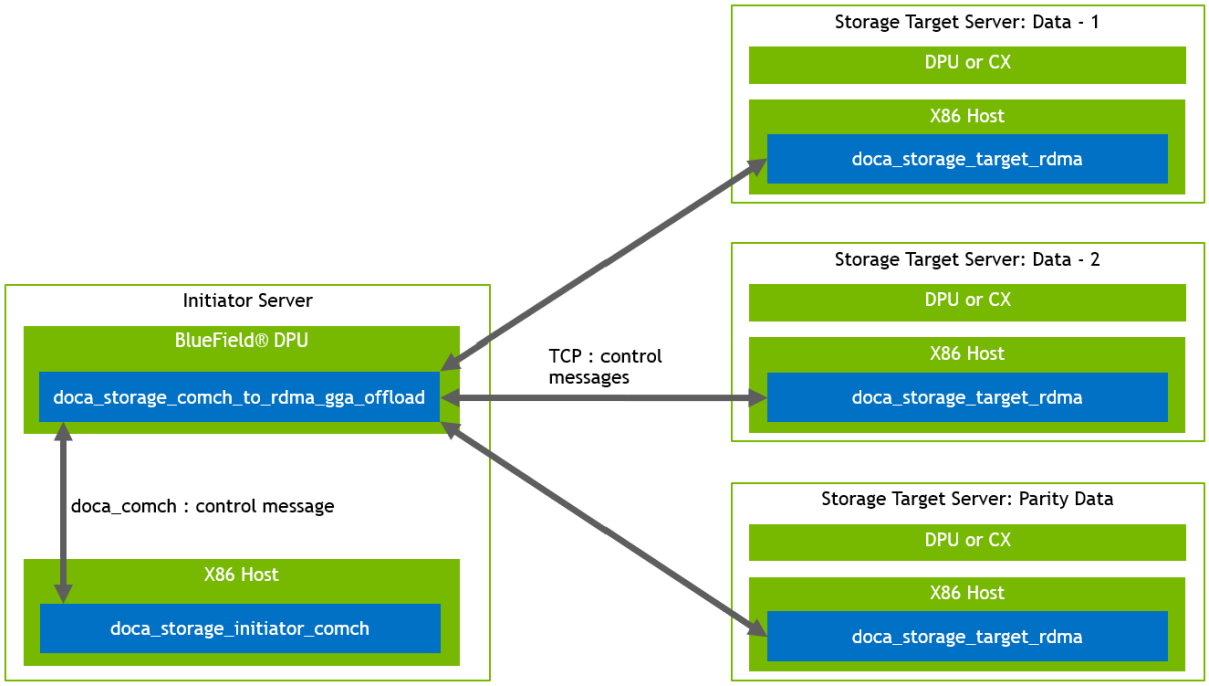
- 1x Initiator (**doca_storage_initiator_comch**): A client/benchmark application that sends read and write requests.
- 1x Service (**doca_storage_comch_to_rdma_gga_offload**): A bridge that abstracts the storage details and orchestrates the offload operations (compression, erasure coding).
- 3x Targets (**doca_storage_target_rdma**): Three instances of a memory-based storage application, one for each data "shard":
    - Target 1 (**data1**): Stores 50% of the logical data.
    - Target 2 (**data2**): Stores the other 50% of the logical data.
    - Target 3 (**data_p**): Stores parity information for redundancy, allowing recovery if **data1** or **data2** is lost.

6.20.4.2 Operational Flow

The use case operates in two main phases.

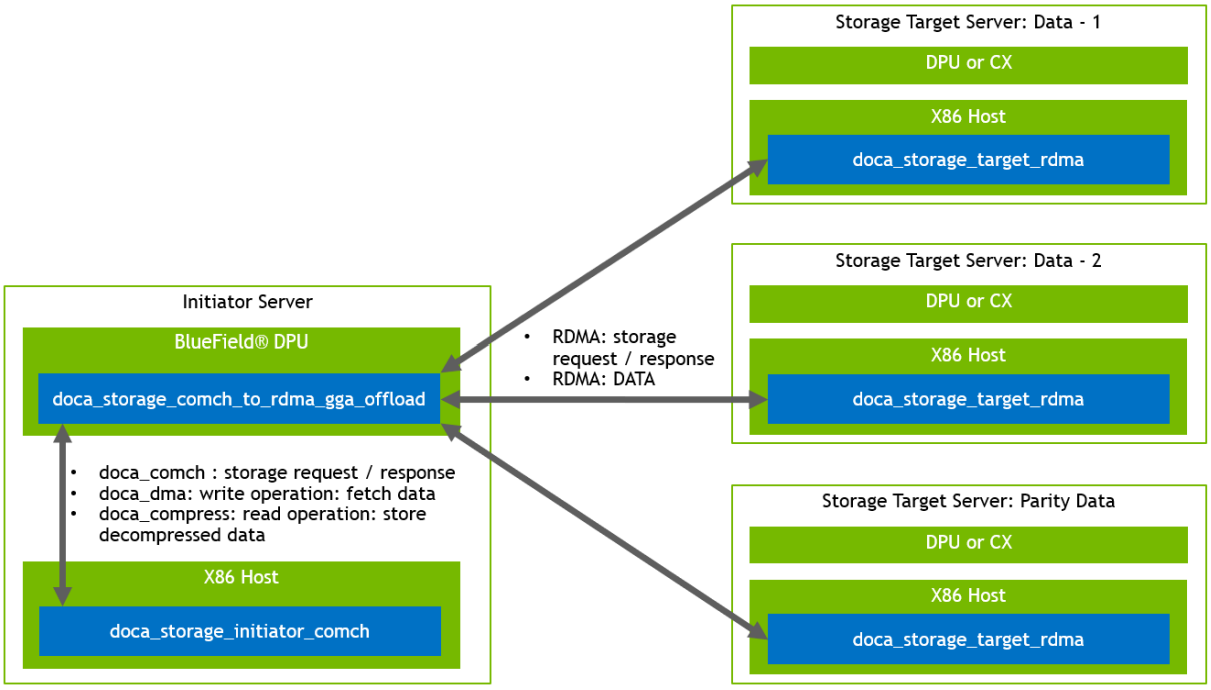
6.20.4.2.1 Setup Phase

The applications use DOCA Comch and three TCP sockets (one for each target) to register with each other, share configuration, and prepare the data path contexts.



6.20.4.2.2 Data Path Phase

Once configured, the contexts are used to perform storage requests and signal completions using **doca_comch_producer**, **doca_comch_consumer**, **doca_rdma**, **doca_dma**, **doca_ec**, and **doca_compress**.



6.20.4.3 Memory Layout

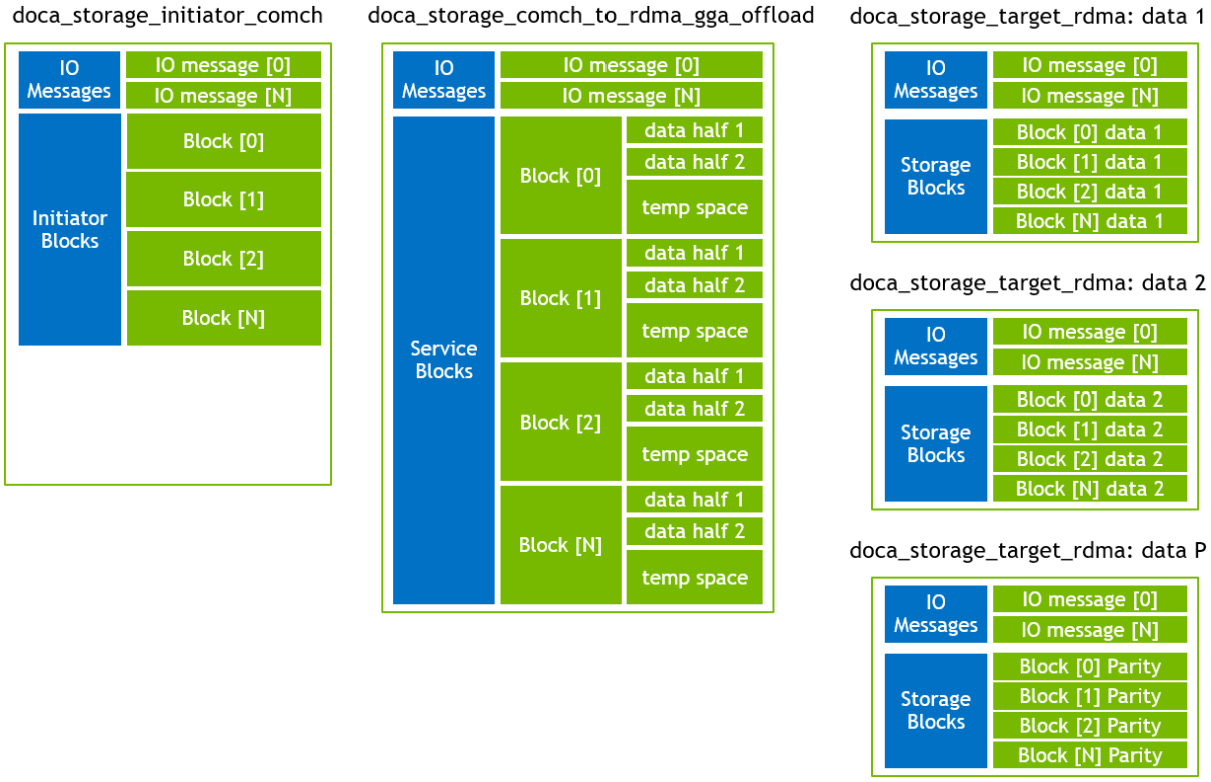
The memory layout for this use case is more complex:

- Initiator: Allocates a memory region partitioned into data blocks.
- Service: Allocates a larger region where each block is twice the size of the initiator's blocks. This provides space to perform compression, decompression, and parity calculations.
- Storage Targets: Each of the three targets allocates blocks that are 50% of the *initiator's* block size. All targets must use the same block size and count.

Data is transferred between the targets and the service via RDMA. Cross-PCIe memory access via DOCA mmap (mmap from Export) is used for data transfers between the service (on the DPU) and the initiator (on the host).

The flow of data through this layout is as follows:

- During Write Operations: DOCA DMA is used to transfer initiator blocks into the service's larger storage blocks (using the cross-PCIe **DOCA mmap**). The block's content is then compressed, parity data is calculated, and the resulting data (**data_1**, **data_2**, **data_p**) is transferred to the storage targets via RDMA.
- During Normal Read Operations: Data is transferred from the targets to the service blocks via RDMA. The DOCA mmap is then used to decompress the data from the service blocks directly into the initiator's blocks.
- During Recovery Read Operations: The service uses RDMA to fetch the valid data half and the parity data into its temporary block space. It combines them to reconstruct the missing data, and the resulting restored data is then decompressed into the initiator's block as normal.



6.20.4.4 Normal Read Flow

The standard read flow, when all targets are available, is as follows:

1. The Initiator sends a read request to the Service (via **doca_comch_producer**).
2. The Service receives the request (via **doca_comch_consumer**).
3. The Service sends two IO requests (via **doca_rdma**) to the **data_1** and **data_2** Targets.
4. Both Targets perform RDMA transfers, sending their storage blocks to the Service's blocks.
5. Both Targets send an IO response to the Service (via **doca_rdma**).
6. The Service, upon receiving both responses, examines the block metadata to determine the compressed data size.
7. The Service decompresses the data (via **doca_compress** using LZ4), using the cross-PCIe **DOCA mmap** feature to write the decompressed data *directly* into the Initiator's memory.
8. The Service forwards an IO response to the Initiator (via **doca_comch_producer**).
9. The Initiator receives the final response (via **doca_comch_consumer**).

6.20.4.5 Recovery Read Flow

If a data target is unavailable, the service performs a recovery read:

1. The Initiator sends a read request to the Service (via **doca_comch_producer**).
2. The Service receives the request (via **doca_comch_consumer**).

3. The Service sends two IO requests (via **doca_rdma**) to the *available* data target and the *parity* target.
    - To recover **data_1**: Send requests to **data_2** and **data_p**.
    - To recover **data_2**: Send requests to **data_1** and **data_p**.
4. Both Targets perform RDMA transfers to the Service's blocks.
5. Both Targets send an IO response to the Service (via **doca_rdma**).
6. The Service uses **doca_ec** to reconstruct the missing data block from the available data and parity blocks.
7. The Service examines the *recovered* block's metadata to determine the compressed data size.
8. The Service decompresses the recovered data (via **doca_compress**), using the cross-PCIe **DOCA mmap** to write the data directly into the Initiator's memory.
9. The Service forwards an IO response to the Initiator (via **doca_comch_producer**).
10. The Initiator receives the final response (via **doca_comch_consumer**).

6.20.4.6 Write Flow

The write path is only available if the **doca_storage_comch_to_rdma_gga_offload** service is compiled with the **liblz4** dev libraries.

1. The Initiator sends a write request to the Service (via **doca_comch_producer**).
2. The Service receives the request (via **doca_comch_consumer**).
3. The Service fetches the data block from the Initiator using **doca_dma** via the cross-PCIe **DOCA mmap**.
4. The Service compresses the data and packages it into a block with metadata.
    a. If the data is not compressible enough, an error is returned to the Initiator (via **doca_comch_producer**).
    b. The Initiator receives the error response (via **doca_comch_consumer**).
    c. The flow ends.
5. The Service sends three IO requests (via **doca_rdma**) to the Targets to write **data_1**, **data_2**, and **data_p**.
6. The Service waits for all three IO responses.
7. The Service forwards a final IO response to the Initiator (via **doca_comch_producer**).
8. The Initiator receives the response (via **doca_comch_consumer**).

6.20.4.7 Running the GGA Offload Use Case

It is assumed that:

- The service and storage applications can communicate via TCP
- The service (**doca_storage_comch_to_rdma_gga_offload**) is deployed on a BlueField-3 DPU

- Full line-rate RDMA data transfer is possible between the service and the storage targets

⚠ All examples use CPU 0. Users should verify if this is optimal by considering NUMA affinity, core pinning, and kernel interrupt distribution.

6.20.4.7.1 Execution Order

Running this use case involves running the applications in this specific order:

1. **doca_storage_target_rdma** (for **data_1**)
2. **doca_storage_target_rdma** (for **data_2**)
3. **doca_storage_target_rdma** (for **data_p**)
4. **doca_storage_comch_to_rdma_gga_offload** (the service)
5. **doca_storage_initiator_comch** (the initiator)

6.20.4.7.2 Placeholders

Replace the following placeholders in the commands with actual values:

- **{{INITIATOR_DEV_PCI_ADDR}}**
- **{{STORAGE_DEV_PCI_ADDR}}**
- **{{STORAGE_IP_ADDR_DATA_1}}**
- **{{STORAGE_IP_ADDR_DATA_2}}**
- **{{STORAGE_IP_ADDR_DATA_P}}**
- **{{STORAGE_TCP_PORT_DATA_1}}**
- **{{STORAGE_TCP_PORT_DATA_2}}**
- **{{STORAGE_TCP_PORT_DATA_P}}**

6.20.4.7.3 Generate SBC File

⚠ **doca_storage_gga_offload_sbc_generator** is not compiled if the **liblz4** dev libraries are not present.

```
doca_storage_gga_offload_sbc_generator \
    -d 3b:00.0 \
    --original-input-data storage_content.txt \
    --block-size 4096 \
    --data-1 data_1.sbc \
    --data-2 data_2.sbc \
    --data-p data_p.sbc
```

6.20.4.7.4 Run Read Data Validity Test

1. Run three storage targets:

```
doca_storage_target_rdma -d {{STORAGE_DEV_PCI_ADDR}} --listen-port {{STORAGE_TCP_PORT_DATA_1}} --binary-
content data_1.sbc --cpu 0
```

```
doca_storage_target_rdma -d {{STORAGE_DEV_PCI_ADDR}} --listen-port {{STORAGE_TCP_PORT_DATA_2}} --binary-
content data_2.sbc --cpu 1
doca_storage_target_rdma -d {{STORAGE_DEV_PCI_ADDR}} --listen-port {{STORAGE_TCP_PORT_DATA_P}} --binary-
content data_p.sbc --cpu 2
```

2. Run the offload service:

```
doca_storage_comch_to_rdma_gga_offload \
    -d 03:00.0 \
    -r {{INITIATOR_DEV_PCI_ADDR}} \
    --data-1-storage {{STORAGE_IP_ADDR_DATA_1}}:{{STORAGE_TCP_PORT_DATA_1}} \
    --data-2-storage {{STORAGE_IP_ADDR_DATA_2}}:{{STORAGE_TCP_PORT_DATA_2}} \
    --data-p-storage {{STORAGE_IP_ADDR_DATA_P}}:{{STORAGE_TCP_PORT_DATA_P}} \
    --trigger-recovery-read-every-n 0 \
    --cpu 0
```

3. Run the initiator:

```
doca_storage_initiator_comch \
    -d {{INITIATOR_DEV_PCI_ADDR}} \
    --execution-strategy read_only_data_validity_test \
    --storage-plain-content storage_content.txt \
    --run-limit-operation-count 20000 \
    --cpu 0
```

6.20.4.7.5 Run Read Throughput Test

1. Repeat step 1 from the "Read Data Validity Test".
2. Repeat step 2 from the "Read Data Validity Test".
3. Run the initiator:

```
doca_storage_initiator_comch \
    -d {{INITIATOR_DEV_PCI_ADDR}} \
    --execution-strategy read_throughput_test\
    --storage-plain-content storage_content.txt \
    --run-limit-operation-count 1000000 \
    --cpu 0
```

6.20.4.7.6 Run Write Data Validity Test

1. Run three storage targets:

```
doca_storage_target_rdma -d {{STORAGE_DEV_PCI_ADDR}} --listen-port {{STORAGE_TCP_PORT_DATA_1}} --binary-
content data_1.sbc --cpu 0
doca_storage_target_rdma -d {{STORAGE_DEV_PCI_ADDR}} --listen-port {{STORAGE_TCP_PORT_DATA_2}} --binary-
content data_2.sbc --cpu 1
doca_storage_target_rdma -d {{STORAGE_DEV_PCI_ADDR}} --listen-port {{STORAGE_TCP_PORT_DATA_P}} --binary-
content data_p.sbc --cpu 2
```

2. Run the offload service:

```
doca_storage_comch_to_rdma_gga_offload \
    -d 03:00.0 \
    -r {{INITIATOR_DEV_PCI_ADDR}} \
    --data-1-storage {{STORAGE_IP_ADDR_DATA_1}}:{{STORAGE_TCP_PORT_DATA_1}} \
    --data-2-storage {{STORAGE_IP_ADDR_DATA_2}}:{{STORAGE_TCP_PORT_DATA_2}} \
    --data-p-storage {{STORAGE_IP_ADDR_DATA_P}}:{{STORAGE_TCP_PORT_DATA_P}} \
    --cpu 0
```

3. Run the initiator:

```
doca_storage_initiator_comch \
    -d {{INITIATOR_DEV_PCI_ADDR}} \
    --execution-strategy read_write_data_validity_test \
    --storage-plain-content storage_content.txt \
    --run-limit-operation-count 20000 \
    --cpu 0
```

6.20.4.7.7 Run Write Throughput Test

1. Repeat step 1 from the "Run Write Data Validity Test".
2. Repeat step 2 from the "Run Write Data Validity Test".
3. Run the initiator:

```
doca_storage_initiator_comch \
    -d {{INITIATOR_DEV_PCI_ADDR}} \
    --execution-strategy write_throughput_test \
    --run-limit-operation-count  50000 \
    --cpu 0
```

6.20.5 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Core
- DOCA Comch
- DOCA Compress
- DOCA DMA
- DOCA Erasure Coding
- DOCA RDMA

Refer to their respective programming guide for more information.

6.20.6 Compiling the Application

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for the application is located in the following directory:

```
/opt/mellanox/doca/applications/storage/
```

doca_storage_gga_offload_sbc_generator is not compiled when **liblz4** dev libraries are not present.

The write path is only available if the **doca_storage_comch_to_rdma_gga_offload** service is compiled with the **liblz4** dev libraries.

6.20.6.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

The storage zero copy applications are:

- **doca_storage_comch_to_rdma_gga_offload**
- **doca_storage_comch_to_rdma_zero_copy**
- **doca_storage_gga_offload_sbc_generator**
- **doca_storage_initiator_comch**
- **doca_storage_target_rdma**

The applications are built under **/tmp/build/storage/**.

6.20.6.2 Compiling Storage Applications Only

To directly build only the storage applications:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_storage=true
ninja -C /tmp/build
```

The storage applications are built under **/tmp/build/storage/**.

Alternatively, you can set the desired flags in the **meson_options.txt** file instead of providing them in the compilation command line:

1. Edit the following flags in **/opt/mellanox/doca/applications/meson_options.txt**:
   - Set **enable_all_applications** to **false**
   - Set **enable_storage** to **true**
2. Run the following compilation commands:

   ```
   cd /opt/mellanox/doca/applications/
   meson /tmp/build
   ninja -C /tmp/build
   ```

   The storage applications are created under **/tmp/build/storage/**.

6.20.6.3 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.20.7 Running the Applications

Refer to each use case for details on how to run the applications:

- Use case 1 examples
- Use case 2 examples

6.20.8 DOCA Storage Comch to RDMA GGA Offload Application Guide

6.20.8.1 Introduction

The **doca_storage_comch_to_rdma_gga_offload** application acts as a bridge between the initiator and three storage targets. It actively participates in data transfer and leverages **doca_libs** to accelerate certain data operations transparently to both the initiator and the targets.

6.20.8.2 System Design

The **doca_storage_comch_to_rdma_gga_offload** application performs the following key functions:

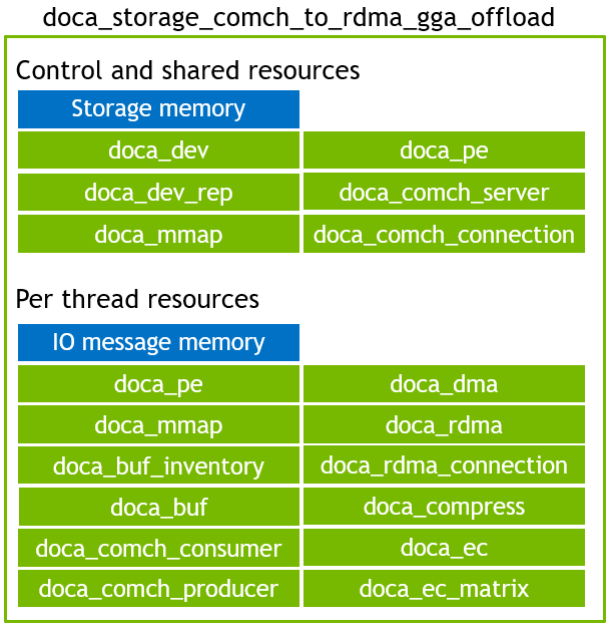
- Orchestrates communication between the initiator and the three storage instances
- Periodically triggers data recovery flows to simulate data loss
- Performs data recovery using erasure coding
- Performs inline data (de)compression
- Performs creation of erasure coding redundancy data

To accomplish these tasks, the application establishes TCP connections to three storage targets and listens for an incoming connection from a single initiator using **doca_comch_server**.

6.20.8.3 Architecture

The **doca_storage_comch_to_rdma_gga_offload** application is divided into two main functional areas:

- Control-time and shared resources
- Per-thread data path resources



The application execution follows two primary phases:

- Control phase
- Data path phase

6.20.8.3.1 Control Phase

This phase begins with establishing connections to the required storage targets, followed by awaiting a client connection. Once all connections are in place, the application waits for specific control commands:

- Query storage
- Init storage
- Start storage

Each control command is processed using the following sequence:

1. Relay the command to each connected storage target.
2. Wait for responses from all storage targets.
3. Perform required post-processing and consistency checks on the responses.
4. Send a response back to the client.

Issuing the start storage command initiates the data path phase. While the data threads begin execution, the main thread continues to wait for final control commands to complete the application's lifecycle:

- Stop storage
- Shutdown

6.20.8.3.2 Data Path Phase

This phase is executed per thread and involves each thread performing I/O operations requested by the client. For a read I/O operation, one of two flows is used:

- Regular read (no recovery)
- Recovery read

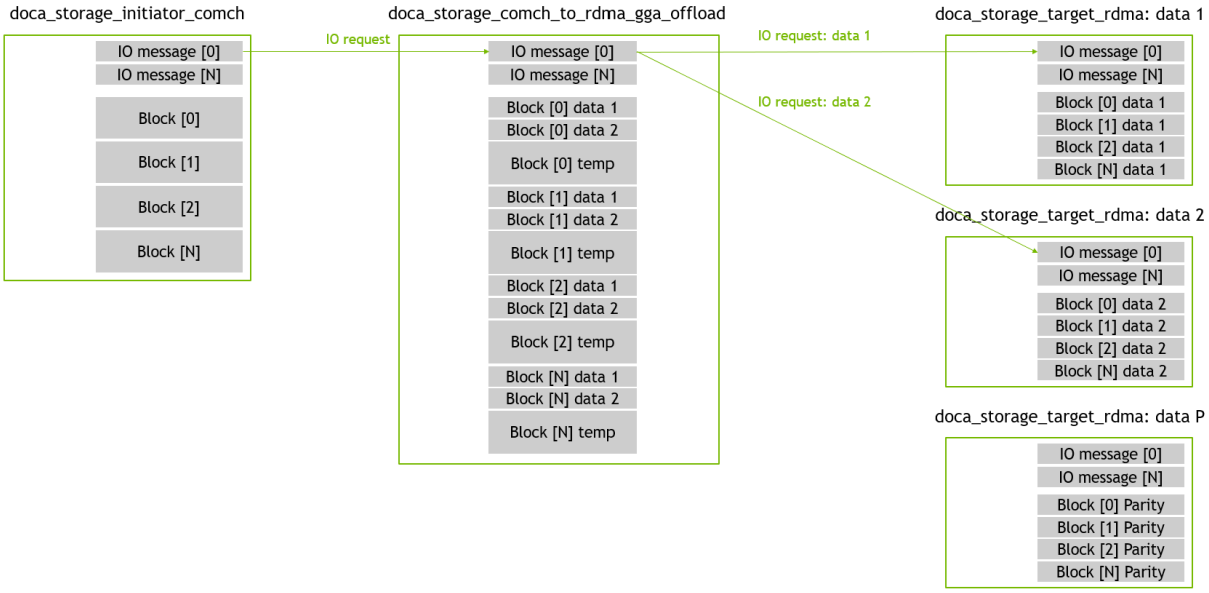
By default, only the regular read flow is executed. However, periodic recovery operations can be enabled by the user—see the "Command Line Flags" section for details.

6.20.8.3.2.1 Regular Read Data Flow

The regular read flow consists of the stages detailed in the following subsections.

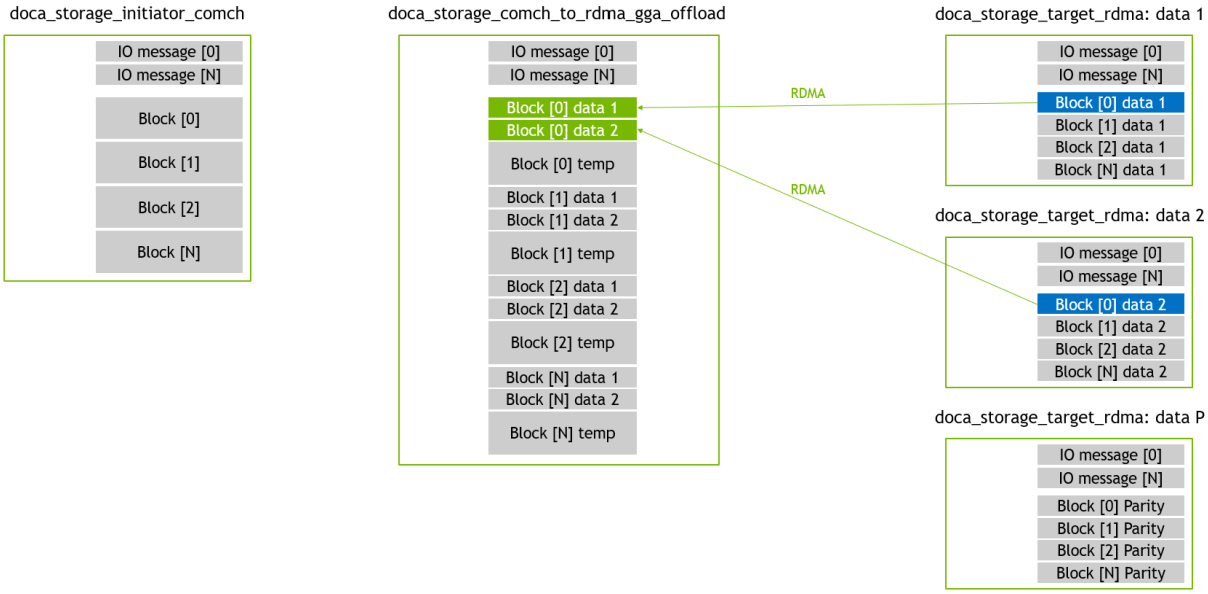
Initiator Request

1. The initiator sends an I/O request to the GGA offload application.
2. The GGA offload application generates two read requests: one for `data_1` and one for `data_2`. Since each storage target only stores a half block, these requests are resized to be half of the initiator's original block size. The output address for each request is set to place the resulting half-block of data into the correct slot (the `data_1` or `data_2` location) inside the service's larger temporary storage block.



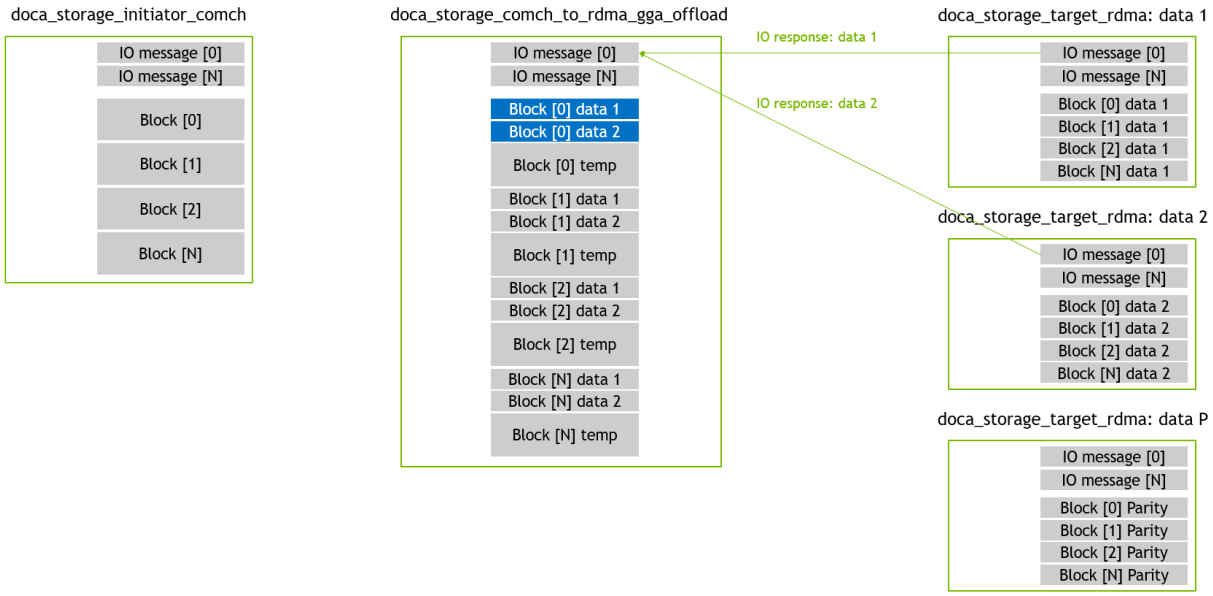
RDMA Transfers

Each storage target performs an RDMA write into the GGA offload memory block at the designated offsets.



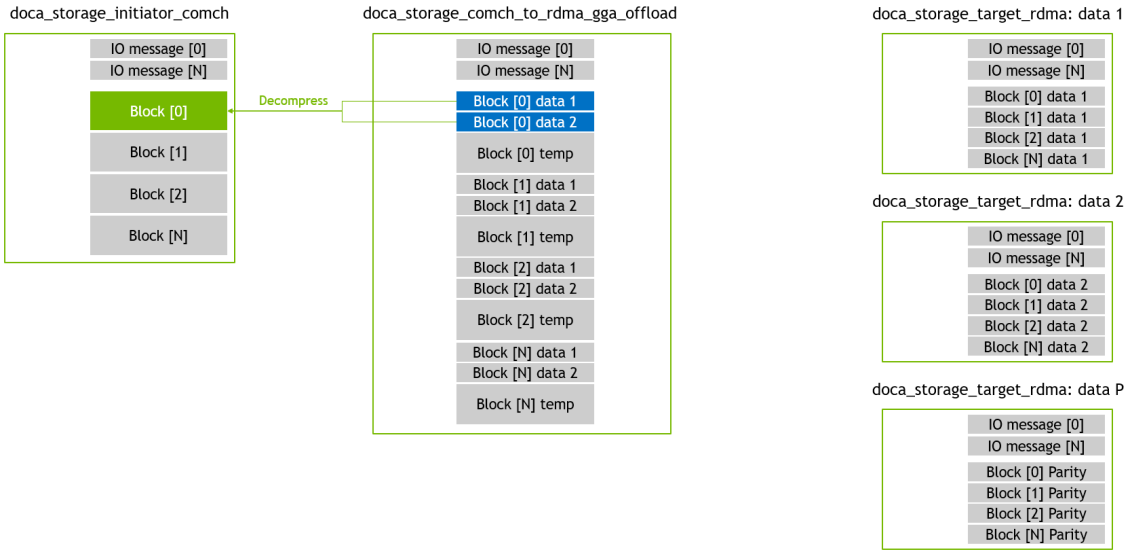
Target Responses

1. Both storage targets send I/O responses upon completing the RDMA transfers.
2. The GGA offload application waits until both responses are received.

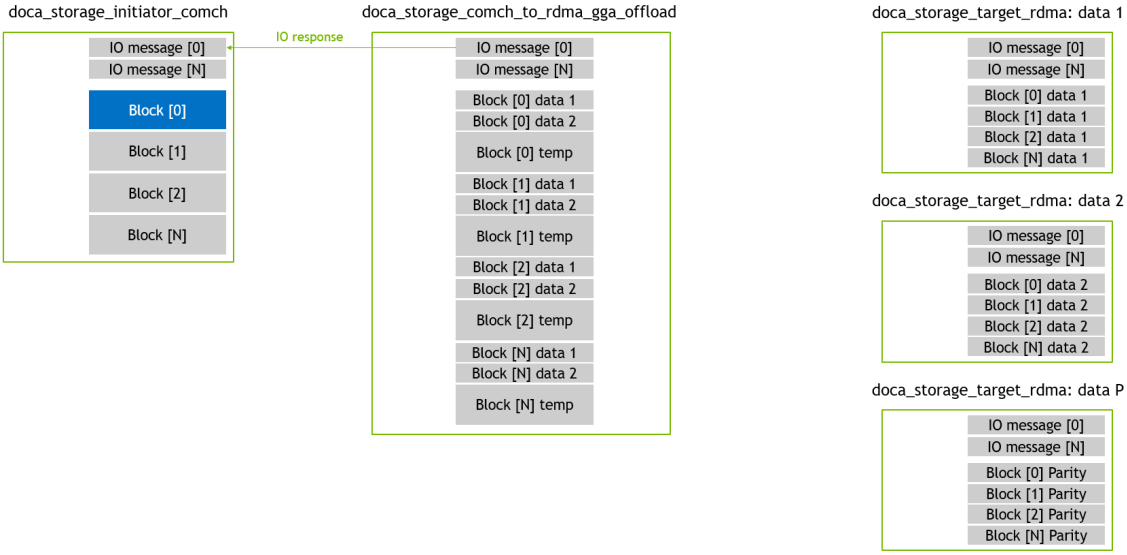


Decompression and Final Response

1. The application performs decompression using the received data blocks. Decompression output is written directly to the initiator memory using the **doca_decompress** task.



2. The application sends an I/O response to the initiator, completing the operation.

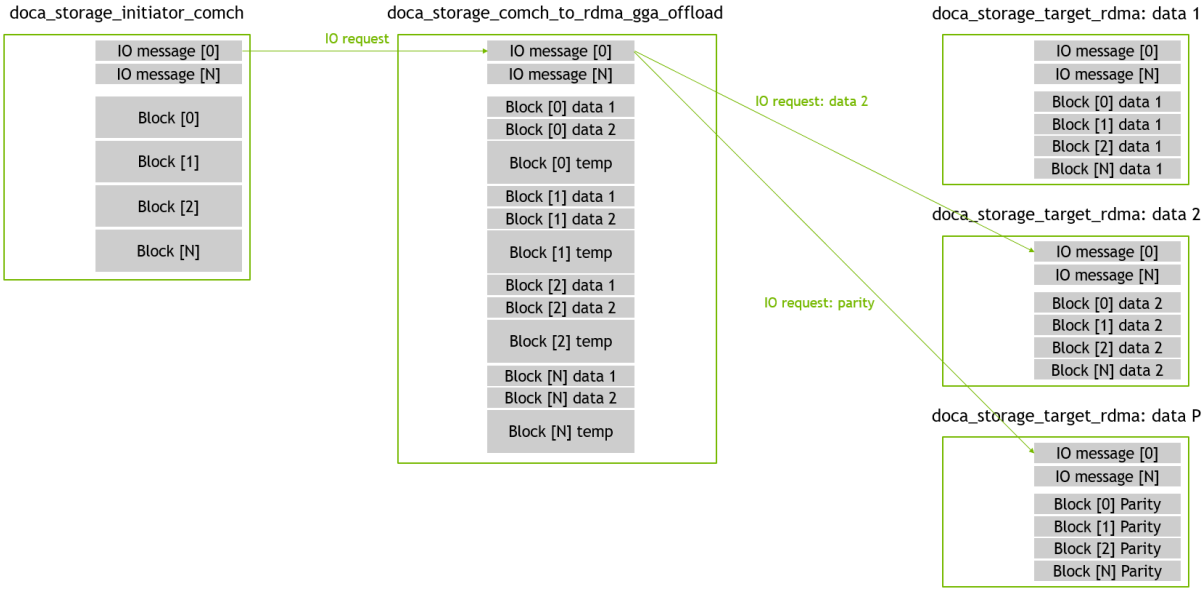


6.20.8.3.2.2 Recovery Read Data Flow

This flow is similar to the regular read, but with added steps for error correction using erasure coding. The process involves the stages detailed in the following subsections.

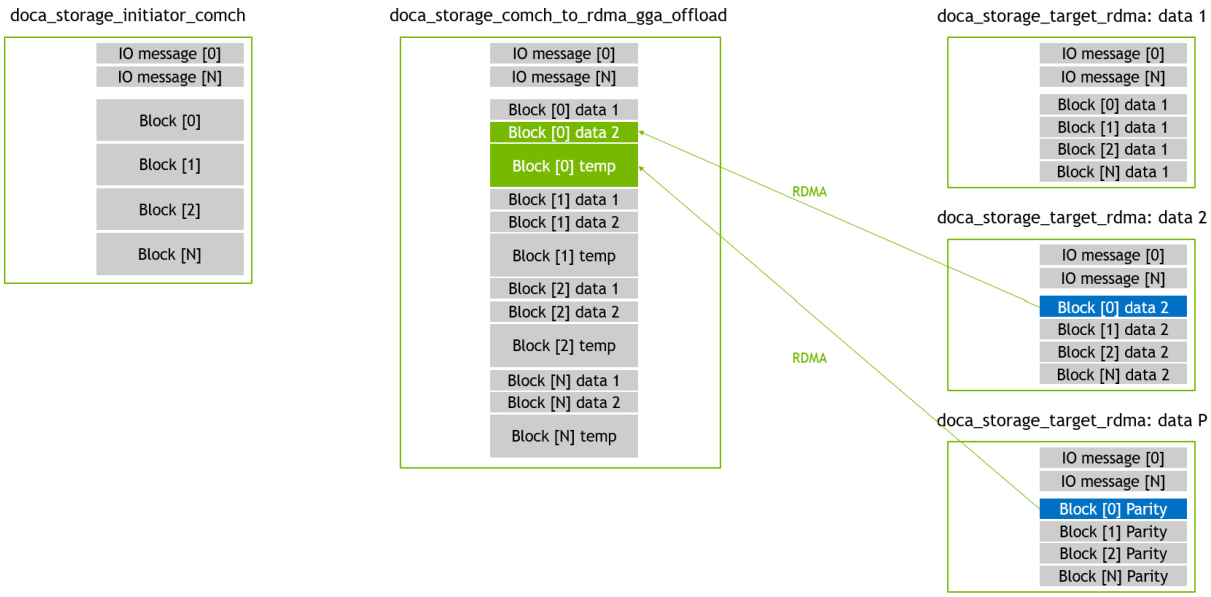
Initiator Request

1. The initiator sends an I/O request to the GGA offload application.
2. Given that recovery is required, the application:
    a. Translates the I/O address normally.
    b. Modifies one of the two storage requests to target the parity partition instead of a standard data partition.
    c. Adds an output offset field to redirect the parity data into a reserved region of GGA offload memory.



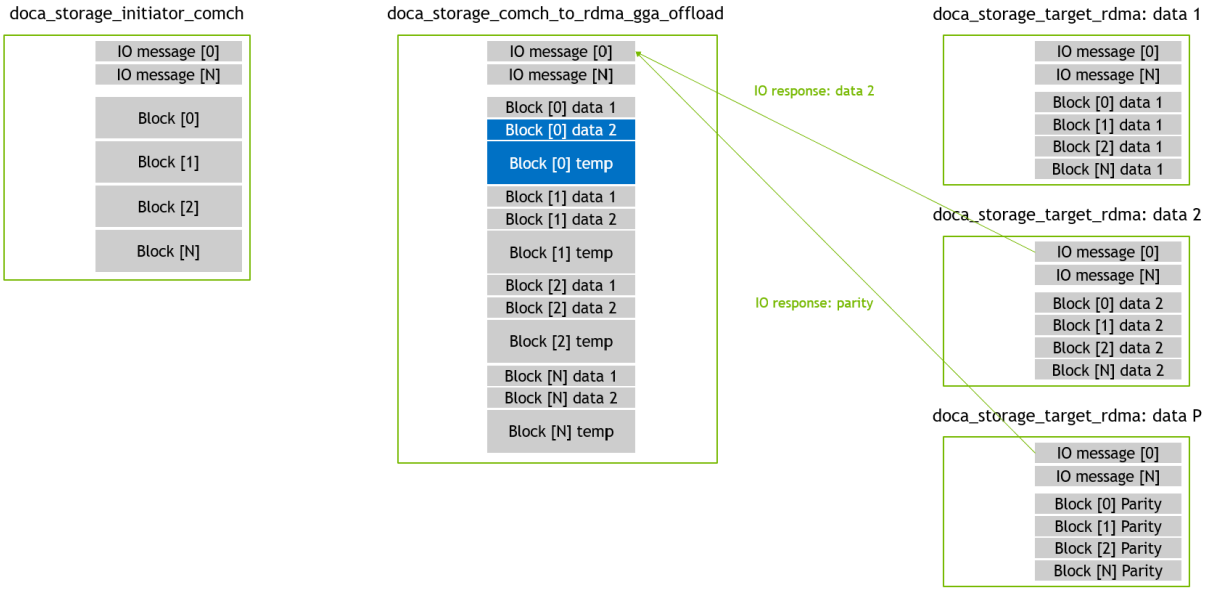
RDMA Transfers

1. Two RDMA transfers are issued:
   - One to the surviving data partition.
   - One to the parity partition.
2. The transferred blocks are written into the GGA offload memory with appropriate alignment.



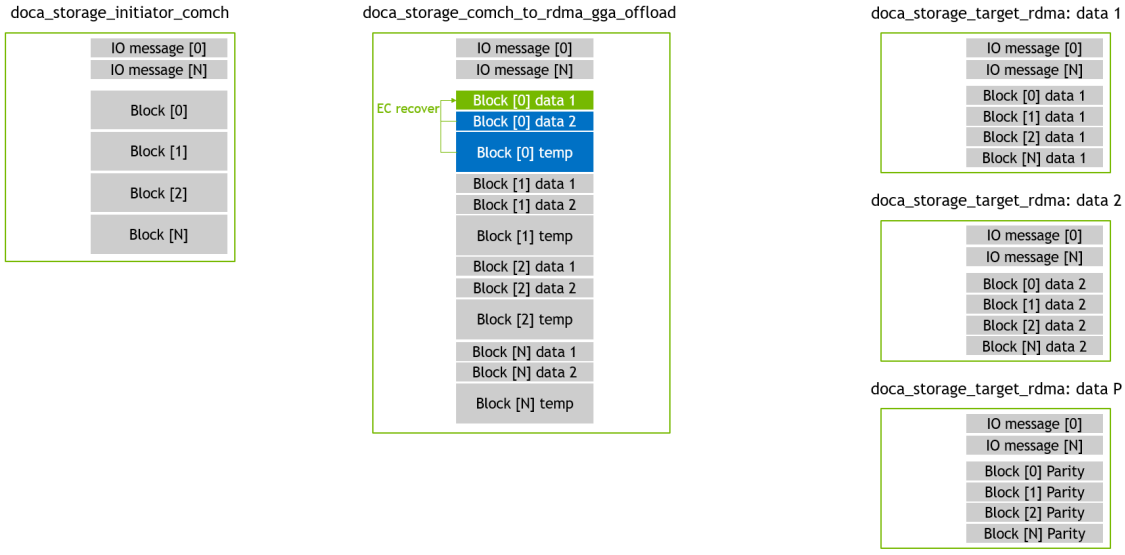
Target Responses

1. Both storage targets reply with I/O responses.
2. The GGA offload application waits for both to complete before proceeding.



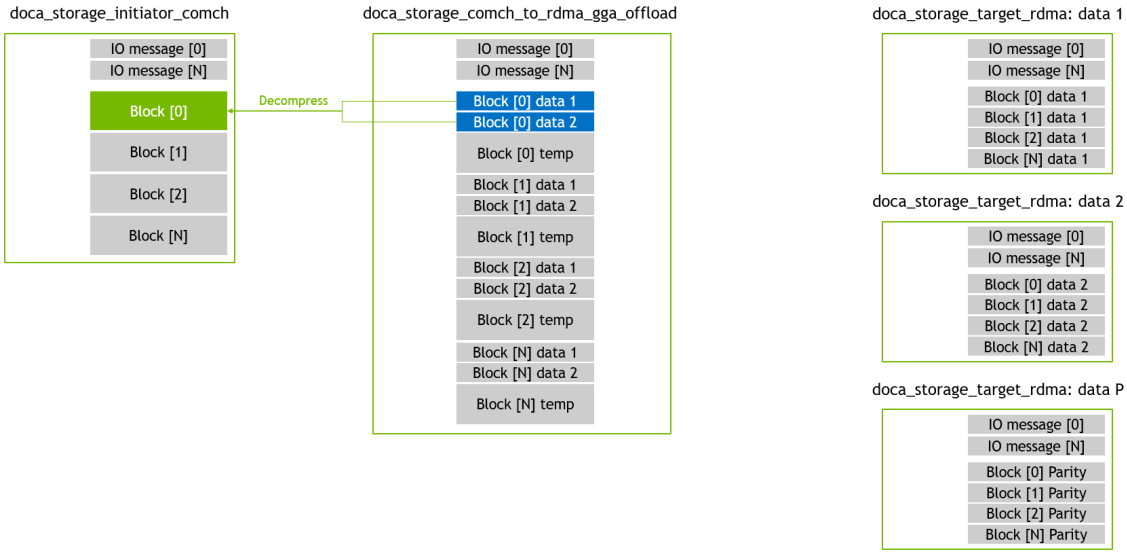
Data Recovery

1. The application performs recovery using the available half block and parity data to reconstruct the missing block.

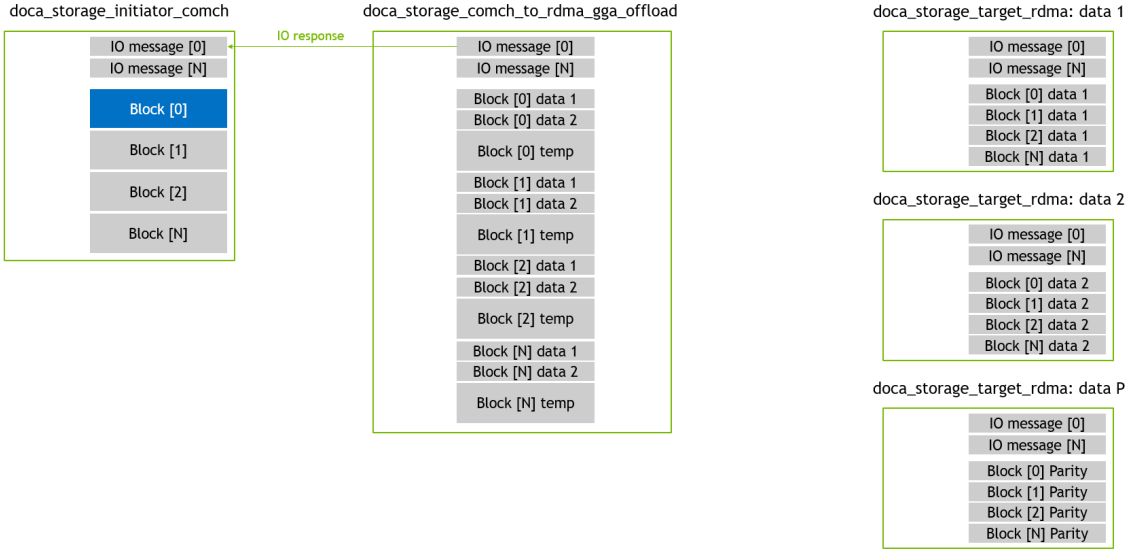


Decompression and Final Response

1. The recovered data is then decompressed. As with the regular read, decompression output is written directly to initiator memory.



- An I/O response is sent to the initiator, completing the operation.



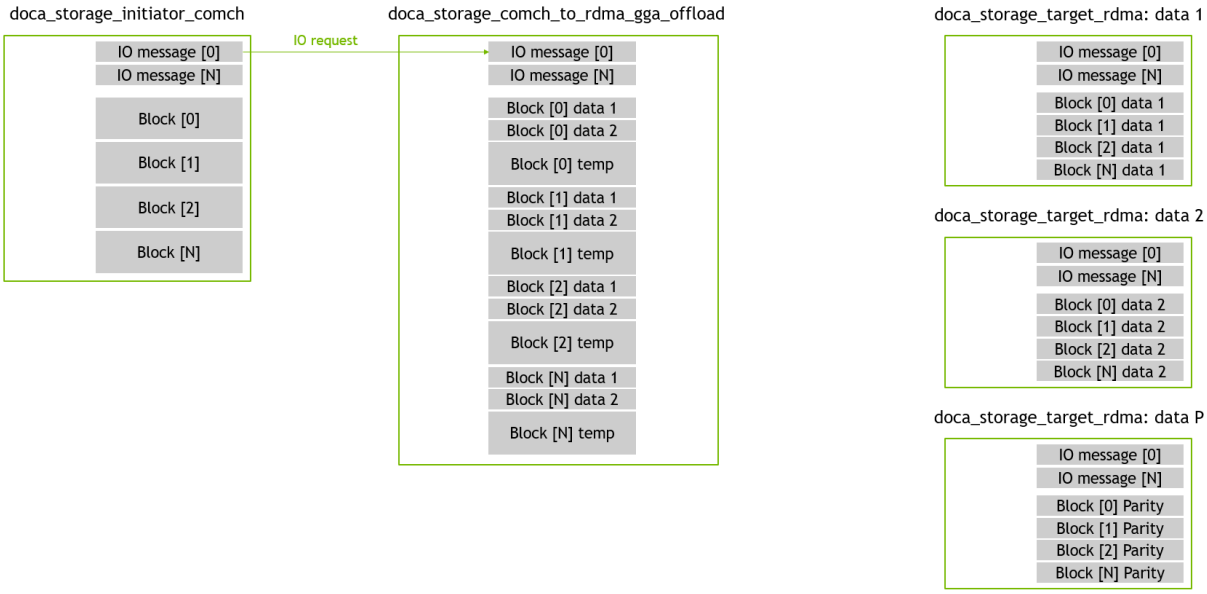
6.20.8.3.2.3 Write Data Flow

> The write path is only available when the service is compiled with the **liblz4** dev libraries.

The write data flow consists of the stages detailed in the following subsections.

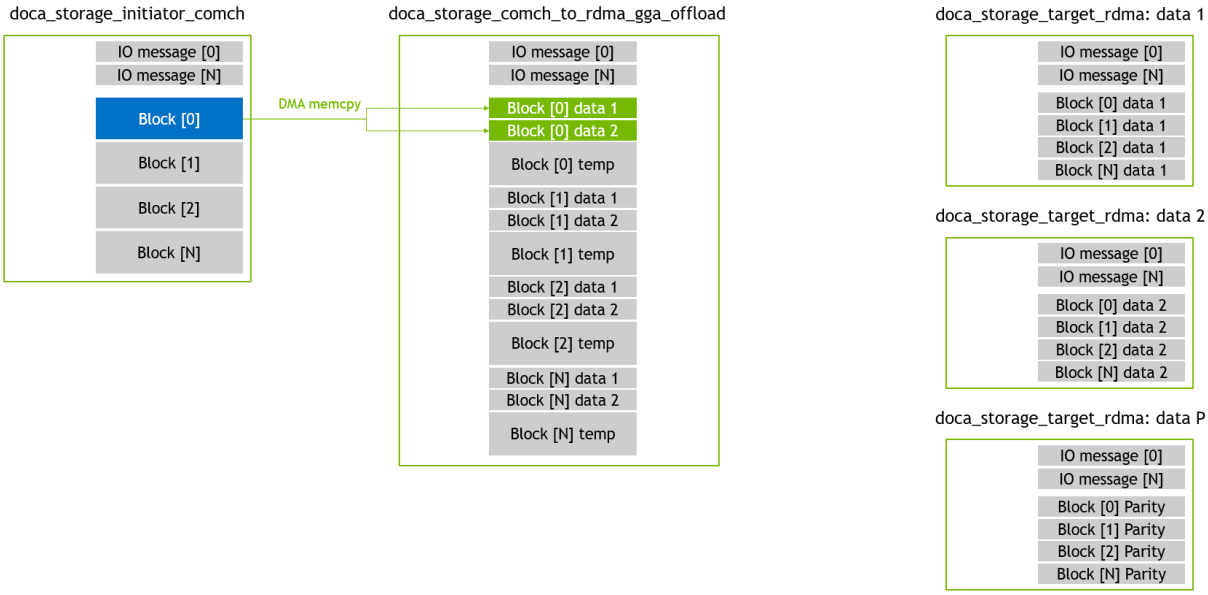
Initiator Request

The initiator sends an I/O request to the GGA offload application.



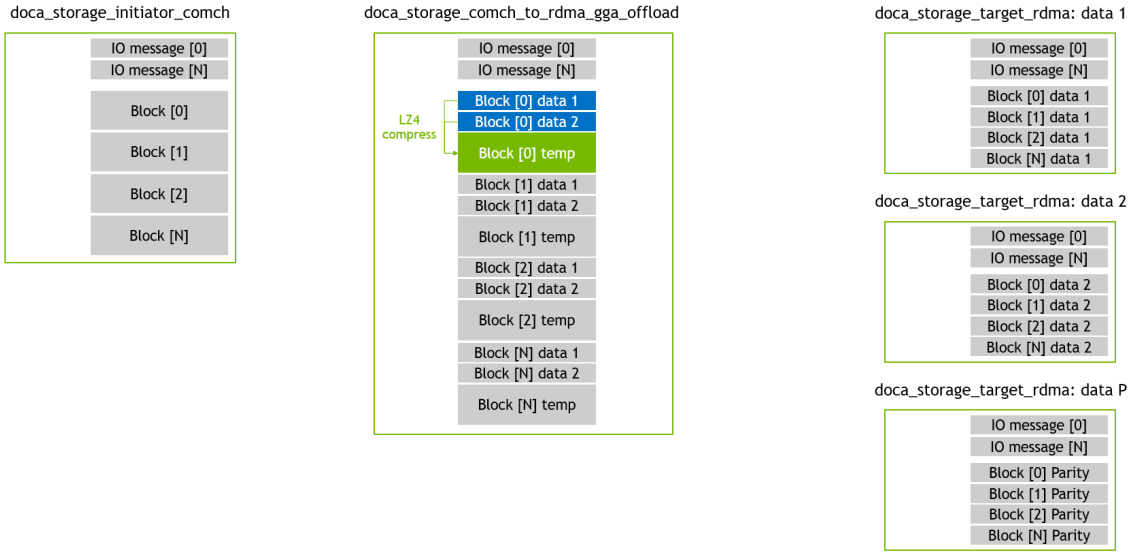
Service Fetches Data

The service fetches the data from the initiator as a single DMA read.

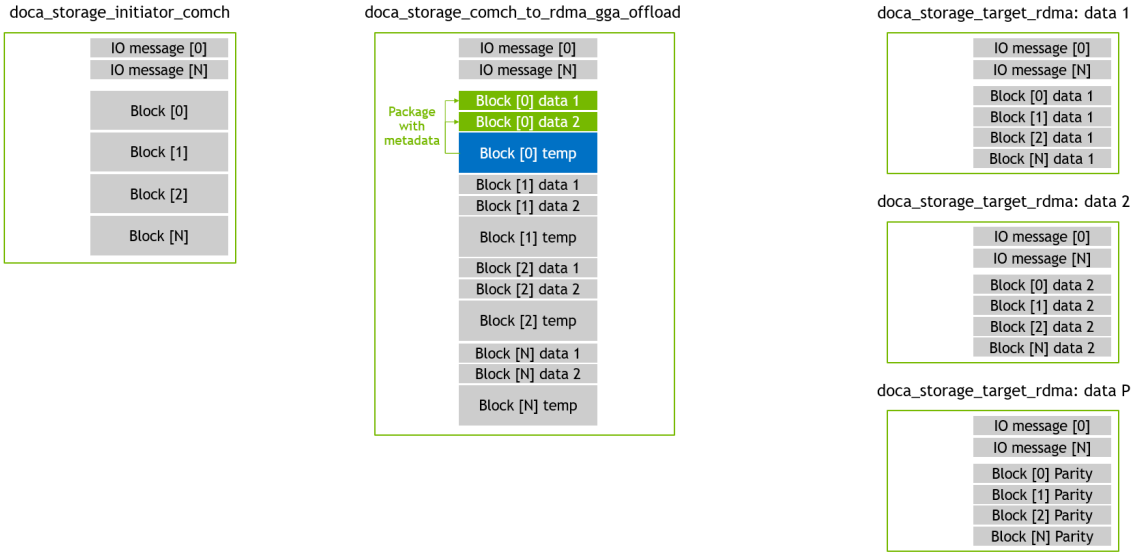


Data Compression

1. The service uses a **liblz4** to compress the data from block data1 and data 2 into the temp storage space.

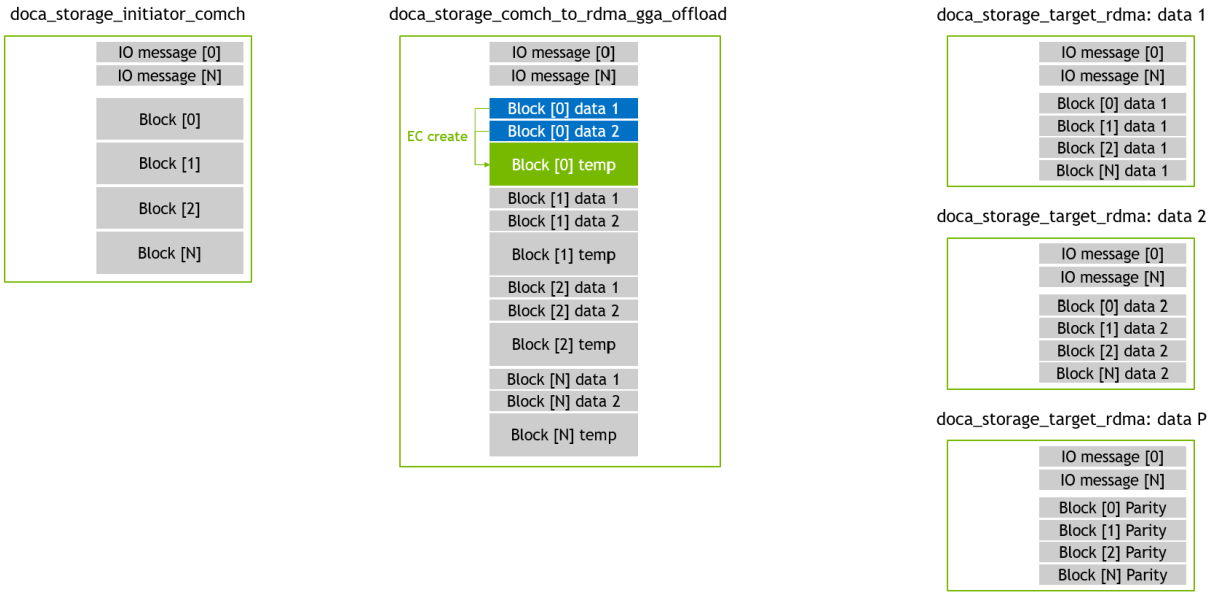


2. The compressed bytes are copied back to data 1 and data 2 location but with a metadata header and trailer added.



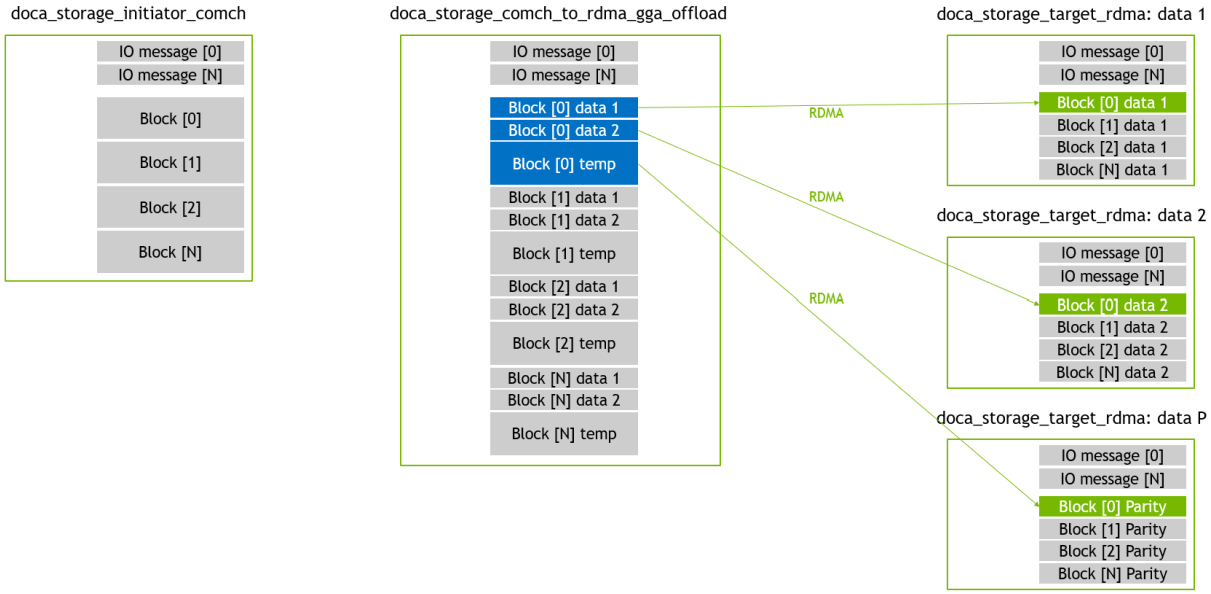
Parity Block Creation

A half block worth of EC redundancy bytes are created from the compressed data (including its metadata).



IO Requests Sent to Storage Targets

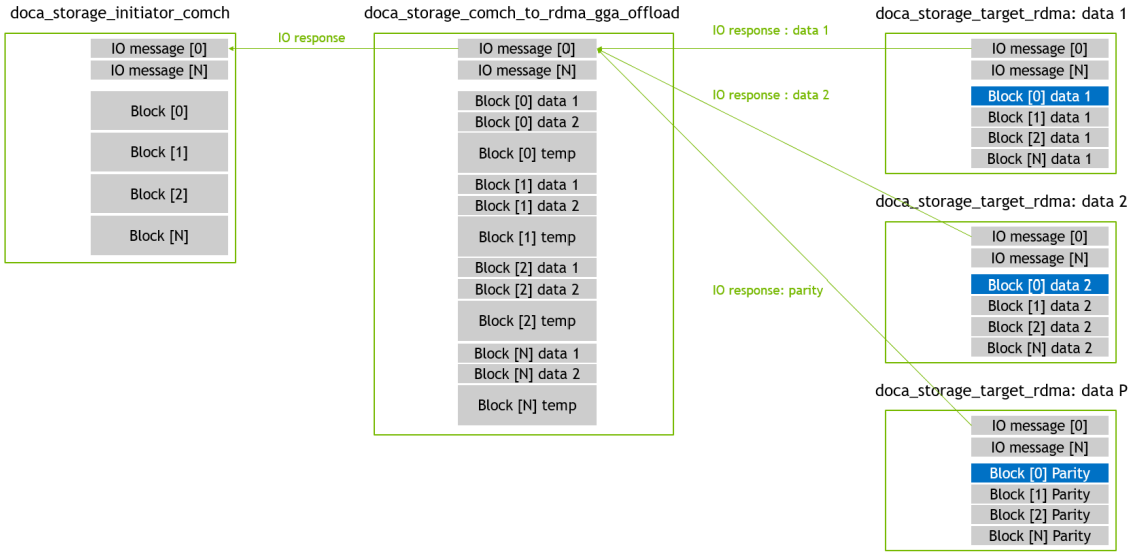
Each of the storage targets are sent an IO write request to store the appropriate half block of data (data 1, data 2, or parity data).



IO Responses

1. All three storage targets reply with I/O responses.
2. The GGA offload application waits for all three to complete.

3. The GGA offload application sends an IO response to the initiator.



6.20.8.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Comch
- DOCA Compress
- DOCA DMA
- DOCA Erasure Coding
- DOCA RDMA

6.20.8.5 Compiling the Application

This application is compiled as part of the set of storage applications. For compilation instructions, refer to the DOCA Storage Applications page.

6.20.8.6 Running the Application

6.20.8.6.1 Application Execution

This application can only run within the NVIDIA® BlueField® DPU.

DOCA Storage Comch to RDMA GGA Offload is provided in source form. Therefore, compilation is required before the application can be executed.

- Application usage instructions:

```
Usage: doca_storage_comch_to_rdma_gga_offload [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
```

```
  --sdk-log-level                    Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                  Parse command line flags from an input json file

Program Flags:
  -d, --device                       Device identifier
  -r, --representor                  Device host side representor identifier
  --cpu                              CPU core to which the process affinity can be set
  --data-1-storage                   Storage server addresses in <ip_addr>:<port> format
  --data-2-storage                   Storage server addresses in <ip_addr>:<port> format
  --data-p-storage                   Storage server addresses in <ip_addr>:<port> format
  --matrix-type                      Type of matrix to use. One of: cauchy, vandermonde Default: vandermonde
  --command-channel-name             Name of the channel used by the doca_comch_client. Default:
"doca_storage_comch"
  --control-timeout                  Time (in seconds) to wait while performing control operations. Default:
5
  --trigger-recovery-read-every-n    Trigger a recovery read flow every N th request. Default: 0 (disabled)
```

This usage printout can be printed to the command line using the **-h** (or **--help**) options:

```
./doca_storage_comch_to_rdma_gga_offload -h
```

For additional information, refer to section "Command-line Flags".

- CLI example for running the application on the BlueField:

```
./doca_storage_comch_to_rdma_gga_offload -d 03:00.0 -r 3b:00.0 --data-1-storage 172.17.0.1:12345 --data-2-
storage 172.17.0.2:12345 --data-p-storage 172.17.0.3:12345 --cpu 0
```

Both the DOCA Comch device PCIe address (**03:00.0**) and the DOCA Comch device representor PCIe address (**3b:00.0**) should match the addresses of the desired PCIe devices.

Storage target IP **address:port** tuples should be updated to refer to the running storage target applications.

6.20.8.6.2 Command-line Flags

6.20.8.6.2.1 General Flags

Short Flag	Long Flag	Description
-h	**--help**	Prints a help synopsis and exits
-v	**--version**	Prints program version information and exits

Short Flag	Long Flag	Description
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.20.8.6.2.2 Program Flags

Short Flag	Long Flag/JSON Key	Description
`d`	`device`	DOCA device identifier. One of: • PCIe address: `3b:00.0` • InfiniBand name: `mlx5_0` • Network interface name: `en3f0pf0sf0` This flag is a mandatory.
`r`	`representor`	DOCA Comch device representor PCIe address This flag is a mandatory.
N/A	`--cpu`	Index of CPU to use. One data path thread is spawned per CPU. Index starts at 0. The user can specify this argument multiple times to create more threads. This flag is a mandatory.

Short Flag	Long Flag/JSON Key	Description
N/A	`--data-1-storage`	IP address and port to use to establish the control TCP connection to the target. ⚠ This flag is a mandatory.
N/A	`--data-2-storage`	IP address and port to use to establish the control TCP connection to the target. ⚠ This flag is a mandatory.
N/A	`--data-p-storage`	IP address and port to use to establish the control TCP connection to the target. ⚠ This flag is a mandatory.
N/A	`--matrix-type`	Type of matrix to use. One of: • `cauchy` • `vandermonde`
N/A	`--command-channel-name`	Allows customizing the server name used for this application instance if multiple comch servers exist on the same device.
N/A	`--control-timeout`	Time, in seconds, to wait while performing control operations
N/A	`--trigger-recovery-read-every-n`	Trigger a recovery read flow every Nth request. Set to 0 to disable recovery reads entirely. For example, a value of 1 triggers a recovery read for every request, while a value of 10 triggers a recovery read for one out of every ten requests.

6.20.8.6.3 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.20.8.7 Application Code Flow

The flow of the application is broken down into key functions and steps:

```
gga_offload_app app{parse_cli_args(argc, argv)};

storage::install_ctrl_c_handler([&app]() {
    app.abort("User requested abort");
});

app.connect_to_storage();
app.wait_for_comch_client_connection();
app.wait_for_and_process_query_storage();
app.wait_for_and_process_init_storage();
app.wait_for_and_process_start_storage();
app.wait_for_and_process_stop_storage();
app.wait_for_and_process_shutdown();
app.display_stats();
```

6.20.8.7.1 Main/Control Thread Flow

1.
```
gga_offload_app app{parse_cli_args(argc, argv)}
```

Parse CLI arguments and use these to create the application instance. Initial resources are also created at this stage:

   a.
```
DOCA_LOG_INFO("Open doca_dev: %s", m_cfg.device_id.c_str());
m_dev = storage::open_device(m_cfg.device_id);
```

   Open a **doca_dev** as specified by the CLI argument: **-d** or **--device**

   b.
```
DOCA_LOG_INFO("Open doca_dev_rep: %s", m_cfg.representor_id.c_str());
m_dev_rep = storage::open_representor(m_dev, m_cfg.representor_id);
```

   Open a **doca_dev_rep** as specified by the CLI argument: **-r** or **--representor**

   c.
```
m_all_ctrl_channels[connection_role::data_1] = storage::control::make_tcp_client_control_channel(
    m_cfg.storage_server_address[connection_role::data_1]);

m_all_ctrl_channels[connection_role::data_2] = storage::control::make_tcp_client_control_channel(
    m_cfg.storage_server_address[connection_role::data_2]);

m_all_ctrl_channels[connection_role::data_p] = storage::control::make_tcp_client_control_channel(
    m_cfg.storage_server_address[connection_role::data_p]);
```

   Create three TCP client control channels (control channel objects provide a unified API so that a TCP client, TCP server, **doca_comch_client**, and **doca_comch_server** all have a consistent API)

   > See **storage_common/control_channel.hpp** for more information about the control channel abstraction.

   d.
```
m_all_ctrl_channels[connection_role::client] =
    storage::control::make_comch_server_control_channel(m_dev,
                                m_dev_rep,
                                m_cfg.command_channel_name.c_str(),
                                this,
                                new_comch_consumer_callback,
                                expired_comch_consumer_callback);
```

   Create a comch server control channel (containing a **doca_comch_server** instance) using the device, representor, and channel name as specified by the CLI argument **--command-channel-name** or the default value if none was specified.

2.
```
storage::install_ctrl_c_handler([&app]() {
    app.abort("User requested abort");
});
```

Set a signal handler for Ctrl+c keyboard inputs so the app can shutdown gracefully.

3.
```
app.connect_to_storage();
```

Connect to the TCP server hosted by each of the three storage targets as defined by the CLI arguments: **--data-1-storage**, **--data-2-storage**, and **--data-p-storage**

   a.
```
void gga_offload_app::connect_to_storage(void)
{
    for (auto *storage_channel : m_storage_ctrl_channels) {
```

```
        DOCA_LOG_DBG("Connect control channel...");
        for (;;) {
            if (m_abort_flag) {
                throw storage::runtime_error{DOCA_ERROR_CONNECTION_ABORTED,
                                "Aborted while connecting to storage"};
            }

            if (storage_channel->is_connected())
                break;
        }
    }
}
```

For each storage control channel (the TCP contexts) poll each context in an infinite loop until either it connects, or the user aborts the application. (The servers should have been started before the service, so no delay is desired between polls of the control context).

4.
```
app.wait_for_comch_client_connection();
```

Wait for a **doca_comch_client** to connect.

a.
```
void gga_offload_app::wait_for_comch_client_connection(void)
{
    while (!m_all_ctrl_channels[connection_role::client]->is_connected()) {
        std::this_thread::sleep_for(std::chrono::milliseconds{100});
        if (m_abort_flag) {
            throw storage::runtime_error{DOCA_ERROR_CONNECTION_ABORTED,
                            "Aborted while connecting to client"};
        }
    }
}
```

Poll the Comch server control channel until a **doca_comch_client** has connected, or the user aborts the application. If any further Comch client attempts to connect to the server it will be automatically rejected by the control channel which is designed for a 1:1 relationship between clients and servers. A sleep is placed in this loop as it may take the user / operator a few seconds to start the client so there is no gain to polling any faster.

5.
```
app.wait_for_and_process_query_storage();
```

Wait for the initiator to send a **query_storage_request** control message and then perform the required actions to fulfill the request:

a.
```
for (auto *storage_ctrl : m_storage_ctrl_channels) {
    auto storage_request = storage::control::message{
        storage::control::message_type::query_storage_request,
        storage::control::message_id{m_message_id_counter++},
        client_request.correlation_id,
        {},
    };

    msg_ids.push_back(storage_request.message_id);
    storage_ctrl->send_message(storage_request);
}
```

Send a query storage request to each storage target server.

b.
```
wait_for_responses(msg_ids, default_control_timeout_seconds);
```

Wait for storage targets to respond.

c. Check the response from each storage target.

   i. If they all report the same dimensions (block size and block count) respond to the initiator with a **query_storage_response** message reporting double the values returned by each storage target (as each storage holds half blocks not

full blocks) I.E. If the storage target reports "64K capacity 2K block size" report "128K capacity 4K block size".

ii. If the dimensions do not match or any other error occurs respond to the initiator with a **error_response** message detailing the error that occurred.

6.
```
app.wait_for_and_process_init_storage();
```

Wait for the initiator to send a **init_storage_request** control message and then perform the required actions to fulfill the request:

a. Use the **init_storage_payload** data to:

i. Set core count (**m_core_count**) as the number of cores requested by the initiator (number of **--cpu** arguments provided to the initiator) OR fail if this is more than the number of **--cpu** arguments provided to the service.

ii. Set number of transactions per core (**m_num_transactions**) to: the number of transactions requested by the initiator doubled. This is doubled to allow for batched task submission and avoid race conditions where the initiator can see a response to a transaction and try to re-submit it before the associated comch producer event callback is received by the server meaning the initiator will continually re-try to send the task and degrade performance until the service catches up and re-submits the consumer task. This should be uncommon, but to make sure it can never happen double the transaction count is allocated so even if every single transaction on the initiator hit this issue there is a full second set of transactions ready on the service side to receive the tasks and avoid any contention. A user can experiment with reducing this value to save memory if desired).

iii. Calculate the per transaction storage block size (**m_per_transaction_chunk_size**) as: **block size * 2.5** if write flow is enabled or **block size * 1.5** when the write flow is disabled.

iv. Allocate storage memory blocks (**m_local_io_region**) as an array of **m_core_count * m_num_transactions * m_per_transaction_chunk_size** bytes.

v.
```
m_remote_io_mmap = storage::make_mmap(m_dev,
                      init_storage_details->mmap_export_blob.data(),
                      init_storage_details->mmap_export_blob.size(),
                      storage::thread_safety::yes);
```

Import the initiator mmap so that its IO blocks can be accessed by the service.

vi.
```
m_local_io_mmap = storage::make_mmap(m_dev,
                    reinterpret_cast<char *>(m_local_io_region),
                    local_storage_size,
                    DOCA_ACCESS_FLAG_LOCAL_READ_WRITE | DOCA_ACCESS_FLAG_PCI_READ_WRITE |
                        DOCA_ACCESS_FLAG_RDMA_WRITE | DOCA_ACCESS_FLAG_RDMA_READ,
                    storage::thread_safety::yes);

std::vector<uint8_t> mmap_export_blob = [this]() {
    uint8_t const *reexport_blob = nullptr;
    size_t reexport_blob_size = 0;
    auto const ret = doca_mmap_export_rdma(m_local_io_mmap,
                          m_dev,
                          reinterpret_cast<void const **>(&reexport_blob),
                          &reexport_blob_size);
    if (ret != DOCA_SUCCESS) {
        throw storage::runtime_error{ret, "Failed to re-export host mmap for rdma"};
    }

    return std::vector<uint8_t>{reexport_blob, reexport_blob + reexport_blob_size};
}();
```

Create and export the service IO blocks memory region. This will be shared with the storage targets so they can access storage service IO blocks memory for RDMA read / write operations.

vii.
```
for (auto *storage_ctrl : m_storage_ctrl_channels) {
    auto storage_request = storage::control::message{
        storage::control::message_type::init_storage_request,
        storage::control::message_id{m_message_id_counter++},
        client_request.correlation_id,
        std::make_unique<storage::control::init_storage_payload>(m_num_transactions,
                                    init_storage_details->core_count,
                                    mmap_export_blob),
    };

    storage_ctrl->send_message(storage_request);
}
```

Send an init storage request to each storage target using:

1. The service transaction count (double the initiator value).
2. The initiator core count.
3. The service IO blocks mmap.

viii. Wait for storage targets to respond.

ix. Send a response to the initiator:

1. Send a **init_storage_response** message upon success or an **error_response** message if anything failed

x. Perform the first stages of the worker threads initialization. These steps are carried out for each thread, but only one thread performs the steps at any time this simplifies the sending and receiving of control messages, the user could modify this flow to execute in parallel if they so desired.

1. Create thread bound to the N^{th} CPU provided to the service via the **--cpu** CLI arguments.

2.
```
m_workers[ii].execute_control_command(
    worker_create_objects_control_command {
        m_dev,
        comch_channel->get_comch_connection(),
        m_local_io_mmap,
        m_remote_io_mmap,
        m_num_transactions,
        m_storage_block_size,
        m_cfg.ec_matrix_type,
        m_cfg.recover_freq}
    );
```

Initialize thread context (asynchronously).

3.
```
connect_rdma(ii, storage::control::rdma_connection_role::io_data, cid);
```

Create RDMA data connections (asynchronously) The thread will connect to each of the three storage targets and create a RDMA context which will be idle from the service's point of view, but is used by the storage target to perform RDMA read / write operations. See the DOCA Storage Target RDMA Application Guide for an explanation why there are two RDMA contexts per thread.

4.
```
connect_rdma(ii, storage::control::rdma_connection_role::io_control, cid);
```

Create RDMA data connections (asynchronously) The thread will connect to each of the three storage targets and create a RDMA context which will be used to exchange IO requests and responses

using RDMA send / recv tasks. See the DOCA Storage Target RDMA Application Guide for an explanation why there are two RDMA contexts per thread.

7. ```
   app.wait_for_and_process_start_storage();
   ```

   Wait for the initiator to send a **start_storage_request** control message and then perform the required actions to fulfill the request:
   a. Forward the start storage request to each of the storage targets.
   b. Wait for storage targets to respond.
   c. Signal all work threads to begin data path operation.
   d. Send a response to the initiator:
      i. Send a **start_storage_response** message upon success or an **error_response** message if anything failed.
8. Data path execution takes place now until either the user abort the program or a stop message is received.
9. ```
   app.wait_for_and_process_stop_storage();
   ```

   Wait for the initiator to send a **stop_storage_request** control message and then perform the required actions to fulfill the request:
   a. Forward the stop storage request to each of the storage targets.
   b. Wait for storage targets to respond.
   c. Signal all work threads to stop data path operation.
   d. Collect run time stats.
   e. Send a response to the initiator:
      i. Send a **stop_storage_response** message upon success or an **error_response** message if anything failed.
10. ```
    app.wait_for_and_process_shutdown();
    ```

    Wait for the initiator to send a **shutdown_request** control message and then perform the required actions to fulfill the request:
    a. Forward the shutdown request to each of the storage targets.
    b. Wait for storage targets to respond.
    c. Destroy worker thread objects.
    d. Send a response to the initiator:
       i. Send a **stop_storage_response** message upon success or an **error_response** message if anything failed.
11. ```
    app.display_stats()
    ```

    Display runtime statistics.
12. Application destructor is triggered:
    a. Destroy control channels.
    b. Destroy storage IO blocks **doca_mmap**.
    c. Destroy storage IO block memory.
    d. Destroy initiator IO blocks **doca_mmap**.

e. Close **doca_dev_rep**.
f. Close **doca_dev**.

13. Program exits.

6.20.8.7.2 Worker/Data Path Thread Flow

The work thread proc executes in two phases: Control / configuration phase, followed by data path phase where read, write, and recovery operations take place.

6.20.8.7.2.1 Worker Init Process

```
void gga_offload_app_worker::thread_proc(gga_offload_app_worker *self, uint16_t core_idx) noexcept
```

The worker starts by executing a loop of:

1. Lock mutex
2. If message pointer is not null:
   a. Process the configuration message.
   b. Set the operation result.
3. Unlock the mutex.
4. Yield.

The following configuration operations can be performed by the worker thread:

1. ```
   void gga_offload_app_worker::create_worker_objects(worker_create_objects_control_command const &cmd)
   ```
   Create general worker objects:
   a. Creates LZ4 software library context for compression operations (when enabled).
   b. Create N transaction objects.
   c. Create IO message memory.
   d. Create IO message **doca_mmap** (to allow the messages to be accessed by comch and RDMA).
   e. Create **doca_buf_inventory**.
   f. Create **doca_pe** to drive the DOCA contexts (**doca_rdma**, **doca_comch_consumer**, **doca_comch_producer**, **doca_compress**, **doca_ec**, **doca_dma**).
   g. Create **doca_comch_consumer**, **doca_comch_producer**, **doca_compress**, **doca_ec**, and **doca_dma** contexts.
   h. Initialize and start contexts.

2. ```
   void gga_offload_app_worker::export_local_rdma_connection_blob(worker_export_local_rdma_connection_command &cmd)
   ```
   Export RDMA context connection binary blob.

3. ```
   void gga_offload_app_worker::import_remote_rdma_connection_blob(worker_import_local_rdma_connection_command const &cmd)
   ```
   Import remote RDMA context connection binary blob.
   ```

4.
```
void gga_offload_app_worker::are_contexts_ready(worker_are_contexts_ready_control_command &cmd) const
 noexcept
```
Poll all contexts to check they are ready to perform data path operations.

5.
```
void gga_offload_app_worker::prepare_tasks(worker_prepare_tasks_control_command const &cmd)
```
Prepare transaction contexts by:
   a. Allocating **doca_buf** objects.
   b. Allocating **doca_task** objects.
   c. Setting task user data.
   d. Setting task pointers into the transaction context.
   e. Setting per transaction IO block addresses.

6.
```
void gga_offload_app_worker::start_data_path(void)
```
Break out of the wait for configuration event loop and start the data path loop.

> After the configuration phase the mutex is not used again

After breaking out of the initial configuration loop, the thread submits receive tasks (Comch consumer tasks, and RDMA recv tasks) then enters the data path function: **run_data_path_ops**.

6.20.8.7.2.2 Worker Data Path Process

```
void gga_offload_app_worker::run_data_path_ops(gga_offload_app_worker::hot_data &hot_data)
{
    DOCA_LOG_INFO("Core: %u running", hot_data.core_idx);

    while (hot_data.run_flag) {
        doca_pe_progress(hot_data.pe) ? ++(hot_data.pe_hit_count) : ++(hot_data.pe_miss_count);
    }

    while (hot_data.error_flag == false && hot_data.in_flight_transaction_count != 0) {
        doca_pe_progress(hot_data.pe) ? ++(hot_data.pe_hit_count) : ++(hot_data.pe_miss_count);
    }
}
```

During the data path phase the thread simply polls the **doca_pe** as quickly as possible to check for a task completion from one of the thread DOCA contexts (**doca_rdma** , **doca_comch_consumer** , **doca_comch_producer** , **doca_compress** , **doca_ec** , **doca_dma**) The interesting work is done in the callbacks of these tasks. The flow will always start with a consumer task completion. This is the reception of the IO message from the initiator.

doca_comch_consumer_task_post_recv_cb

```
void gga_offload_app_worker::doca_comch_consumer_task_post_recv_cb(doca_comch_consumer_task_post_recv *task,
                                   doca_data task_user_data,
                                   doca_data ctx_user_data) noexcept
{
    static_cast<void>(task_user_data);

    auto const transaction_idx = doca_task_get_user_data(doca_comch_consumer_task_post_recv_as_task(task)).u64;
    auto *const hot_data = static_cast<gga_offload_app_worker::hot_data *>(ctx_user_data.ptr);

    auto const ret = hot_data->start_transaction(hot_data->transactions[transaction_idx]);
    if (ret != DOCA_SUCCESS) {
        DOCA_LOG_ERR("Failed to start transaction: %s", doca_error_get_name(ret));
    }
}
```

This callback simply interprets the task callback parameters to call the **start_transaction** function. The start transaction flow performs the following steps:

1. Check transaction request type
2. Set transaction action flags based on the transaction type (read, write)
3. Calls the **progress_transaction** function to execute as many of the pending actions as possible. Some actions like sending IO requests to the storage targets, decompressing data, creating erasure coding blocks, amongst others are themselves asynchronous so processing will stop until a future task callback resumes the processing.

The transaction will result in a read or write operation being performed. A read operation may result in a regular read or a recovery read depending on value provides by the **--trigger-recovery-read-every-n** CLI argument. When non 0 even Nth read will perform a recovery read flow.

The flow of these operations are:

- Write flow
    - Fetch data from initiator memory using DMA
    - Compress data
    - Generate EC redundancy blocks
    - Store **data_1** , **data_2** , and **data_p** to the respective storage targets
    - Respond with IO response
- Regular read flow
    - Fetch **data_1** and **data_2** from the respective storage targets
    - Decompress the data into initiator memory
    - Respond with IO response
- Recovery read flow
    - Fetch **data_1** OR **data_2** from the respective storage target
    - Fetch **data_p** from the **data_p** storage target
    - Recover the missing data half using the existing data and the redundancy data
    - Decompress the data into initiator memory
    - Respond with IO response

doca_comch_producer_task_send_cb

```
void gga_offload_app_worker::doca_comch_producer_task_send_cb(doca_comch_producer_task_send *task,
                                  doca_data task_user_data,
                                  doca_data ctx_user_data) noexcept
{
    static_cast<void>(task);

    auto *const hot_data = static_cast<gga_offload_app_worker::hot_data *>(ctx_user_data.ptr);
    auto &transaction = hot_data->transactions[task_user_data.u64];
    transaction.pending_actions = 0;
    --(hot_data->in_flight_transaction_count);
    ++(hot_data->completed_transaction_count);

    static_cast<void>(
        doca_buf_reset_data_len(doca_comch_consumer_task_post_recv_get_buf(transaction.host_request_task)));

    auto const ret = doca_task_submit(doca_comch_consumer_task_post_recv_as_task(transaction.host_request_task));
    if (ret != DOCA_SUCCESS) {
        DOCA_LOG_ERR("Failed to submit doca_comch_consumer_task_post_recv: %s", doca_error_get_name(ret));
        hot_data->error_flag = true;
        hot_data->run_flag = false;
    }
}
```

The producer task completion callback signals that the transaction response has been delivered to the initiator. The transaction can now be re-used by re-submitting the transaction consumer task.

doca_rdma_task_send_cb

```
void gga_offload_app_worker::doca_rdma_task_send_cb(doca_rdma_task_send *task,
                            doca_data task_user_data,
                            doca_data ctx_user_data) noexcept
{
    static_cast<void>(task);

    auto *const hot_data = static_cast<gga_offload_app_worker::hot_data *>(ctx_user_data.ptr);
    auto &transaction = hot_data->transactions[task_user_data.u64];

    --(transaction.pending_storage_response_count);
    if (transaction.pending_storage_response_count == 0) {
        transaction.pending_actions ^= static_cast<uint32_t>(transaction_action::wait_for_storage_completion);
        hot_data->progress_transaction(transaction);
    }
}
```

The RDMA send task callback signals that an IO request has been delivered to one of the storage target machines. A counter is decremented to know when all IO requests and IO responses have been received so that the next step of processing the transaction can be carried out. For a read IO operation this could be data recovery, or decompression next, for a write operation the transaction is completed and the IO response to the initiator can be sent.

doca_rdma_task_receive_cb

```
void gga_offload_app_worker::doca_rdma_task_receive_cb(doca_rdma_task_receive *task,
                              doca_data task_user_data,
                              doca_data ctx_user_data) noexcept
{
    static_cast<void>(task_user_data);

    auto *const hot_data = static_cast<gga_offload_app_worker::hot_data *>(ctx_user_data.ptr);

    auto *const rdma_io_message = storage::get_buffer_bytes(doca_rdma_task_receive_get_dst_buf(task));
    auto const transaction_idx = storage::io_message_view::get_user_data(rdma_io_message).u64;

    auto &transaction = hot_data->transactions[transaction_idx];

    auto const io_result = storage::io_message_view::get_result(rdma_io_message);
    if (io_result != DOCA_SUCCESS) {
        // store error
        storage::io_message_view::set_result(io_result, transaction.initiator_io_message);
    }

    --(transaction.pending_storage_response_count);
    if (transaction.pending_storage_response_count == 0) {
        transaction.pending_actions ^= static_cast<uint32_t>(transaction_action::wait_for_storage_completion);
        hot_data->progress_transaction(transaction);
    }

    if (hot_data->run_flag) {
        static_cast<void>(doca_buf_reset_data_len(doca_rdma_task_receive_get_dst_buf(task)));
        auto const ret = doca_task_submit(doca_rdma_task_receive_as_task(task));
        if (ret != DOCA_SUCCESS) {
            DOCA_LOG_ERR("Failed to resubmit doca_rdma_task_receive");
            hot_data->run_flag = false;
            hot_data->error_flag = true;
        }
    }
}
```

The RDMA receive task callback signals that an IO response has been received from one of the storage targets. As with a send task completion the counter is decremented to know if processing can continue. In addition the IO response status is inspected as if the storage operation failed then the transaction result should be marked as a failure. The receive task is then re-submitted to be ready to receive another message from that storage target in a future transaction.

doca_ec_task_recover_cb

```
void gga_offload_app_worker::doca_ec_task_recover_cb(doca_ec_task_recover *task,
```

```
                            doca_data task_user_data,
                            doca_data ctx_user_data) noexcept
{
    static_cast<void>(task);
    auto *hot_data = static_cast<gga_offload_app_worker::hot_data *>(ctx_user_data.ptr);
    auto &transaction = hot_data->transactions[task_user_data.u64];

    transaction.pending_actions &= ~(static_cast<uint32_t>(transaction_action::recover_a) |
                    static_cast<uint32_t>(transaction_action::recover_b));
    hot_data->progress_transaction(transaction);
}
```

The EC recover task callback signals the completion of data recovery using erasure coding blocks. The transaction can now proceed to the next setp, which will be data decompression to continue the read process.

doca_compress_task_decompress_lz4_stream_cb

```
void gga_offload_app_worker::doca_compress_task_decompress_lz4_stream_cb(doca_compress_task_decompress_lz4_stream
*task,
                                    doca_data task_user_data,
                                    doca_data ctx_user_data) noexcept
{
    static_cast<void>(task);
    static_cast<void>(task_user_data);

    auto *hot_data = static_cast<gga_offload_app_worker::hot_data *>(ctx_user_data.ptr);
    auto &transaction = hot_data->transactions[task_user_data.u64];

    transaction.pending_actions ^= static_cast<uint32_t>(transaction_action::decompress);
    hot_data->progress_transaction(transaction);
}
```

The decompress LZ4 stream task callback signals the completion of data decompression. As the decompressed data is stored directly into initiator memory the transaction can continue to send the IO response to the initiator.

doca_ec_task_create_cb

```
void gga_offload_app_worker::doca_ec_task_create_cb(doca_ec_task_create *task,
                            doca_data task_user_data,
                            doca_data ctx_user_data) noexcept
{
    static_cast<void>(task);
    static_cast<void>(task_user_data);

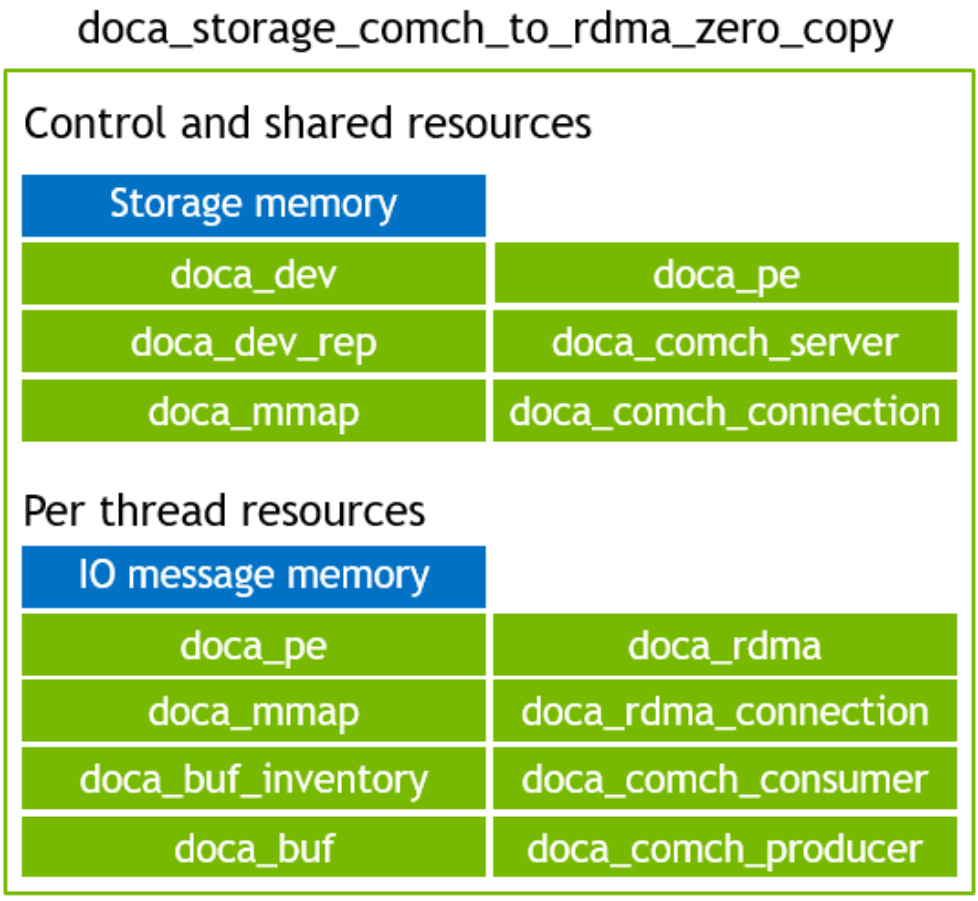
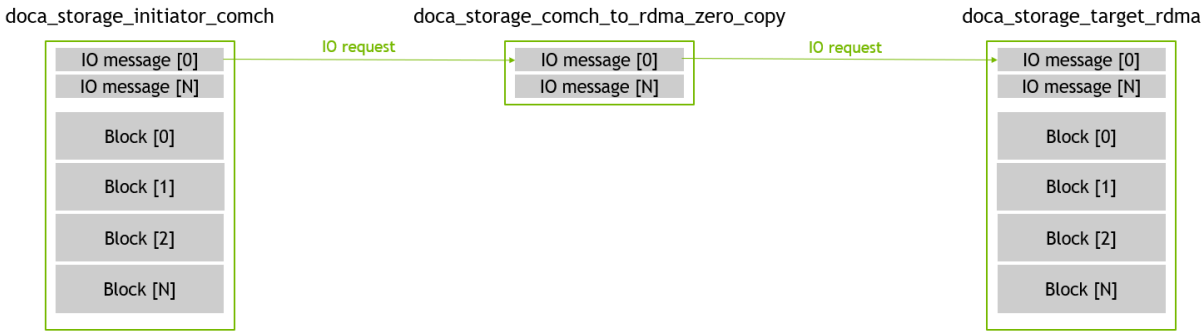
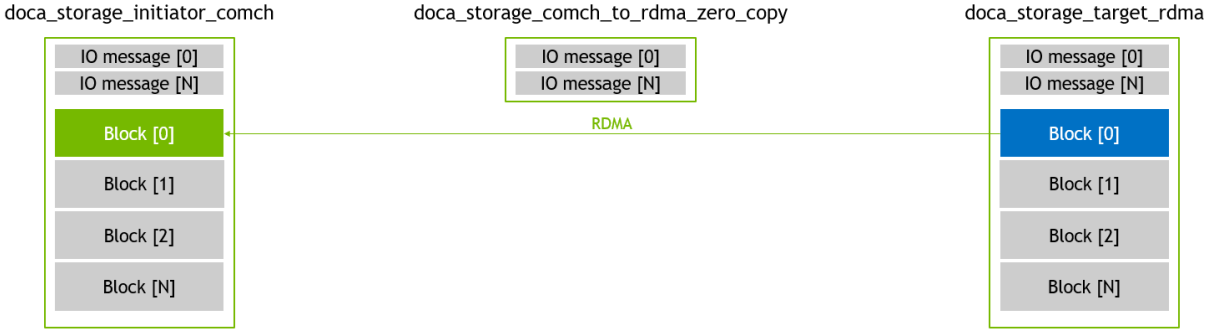
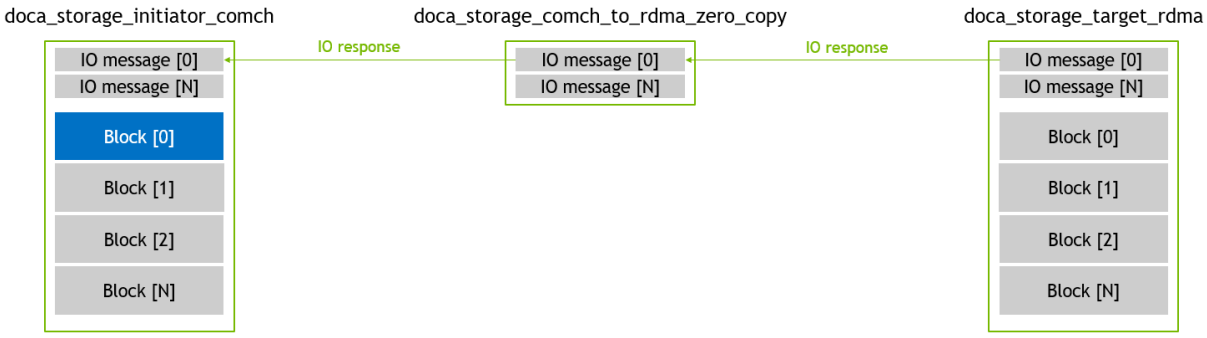
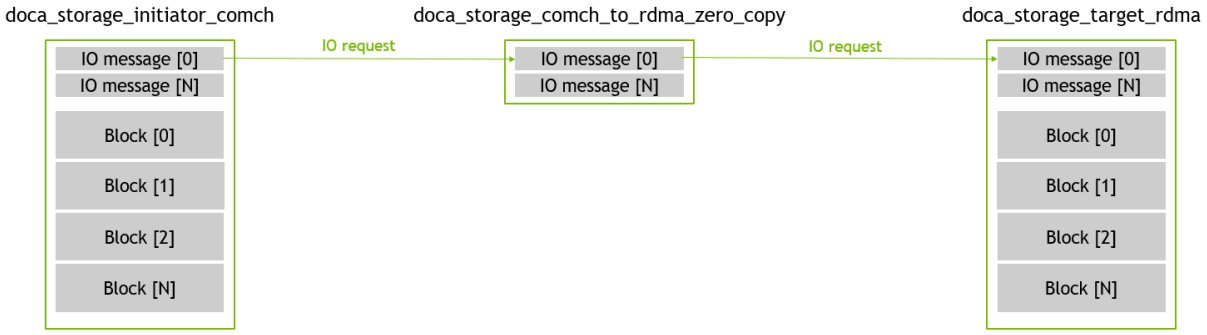
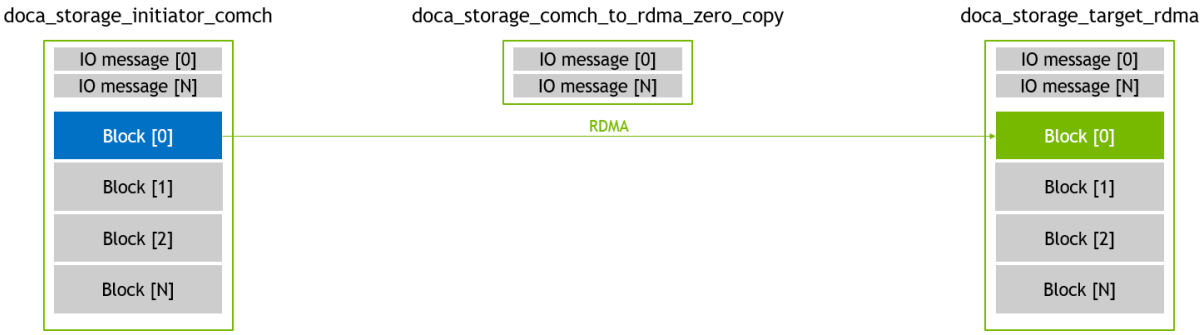
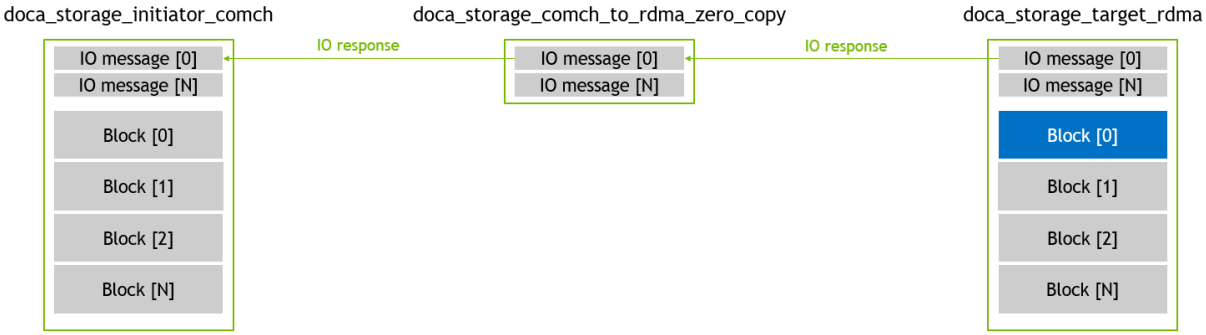
    auto *hot_data = static_cast<gga_offload_app_worker::hot_data *>(ctx_user_data.ptr);
    auto &transaction = hot_data->transactions[task_user_data.u64];

    transaction.pending_actions ^= static_cast<uint32_t>(transaction_action::produce_ec_blocks);
    hot_data->start_commit_to_storage(transaction);
}
```

The EC create task callback signals the completion of the generation of EC redundancy data. The transaction processing can now continue by sending IO requests to all three storage targets to store the three half blocks: `data_1`, `data_2`, and `data_p` in the respective storage targets IO blocks.

doca_dma_task_memcpy_cb

```
void gga_offload_app_worker::doca_dma_task_memcpy_cb(doca_dma_task_memcpy *task,
                            doca_data task_user_data,
                            doca_data ctx_user_data) noexcept
{
    static_cast<void>(task);
    static_cast<void>(task_user_data);

    auto *hot_data = static_cast<gga_offload_app_worker::hot_data *>(ctx_user_data.ptr);
    auto &transaction = hot_data->transactions[task_user_data.u64];

    transaction.pending_actions ^= static_cast<uint32_t>(transaction_action::fetch_from_initiator);

    auto const ret = hot_data->compress_data(transaction);
    if (ret != DOCA_SUCCESS) {
        transaction.set_error(ret);
    }

    hot_data->progress_transaction(transaction);
```

```
}
```

The DMA memcpy task callback signals that the fetch of initiator data into service memory is now complete. The software compression of the data is immediately executed to completion as the software LZ4 library does not use events to signal completion. The transaction processing now continues which will start the creation of EC redundancy data.

6.20.8.8 References

- `/opt/mellanox/doca/applications/storage/`

6.20.9 DOCA Storage Comch to RDMA Zero Copy Application Guide

6.20.9.1 Introduction

The `doca_storage_comch_to_rdma_zero_copy` application serves as a bridge between the initiator and a single storage target. It's only role in the data path is to forward the io requests and io responses between the initiator and storage target.

6.20.9.2 System Design

The `doca_storage_comch_to_rdma_zero_copy` application performs the following functions:

- Relay of io requests from the initiator to the storage target
- Relay of io responses from the storage target to the initiator

To achieve this it expects to be able to connect to a storage target using TCP connections and will then listen for an incoming connection from a single initiator using `doca_comch_server`.

6.20.9.3 Architecture

The `doca_storage_comch_to_rdma_zero_copy` application is split into to two functional areas:

- Control time and shared resources
- Per thread data path resources



The flow of the application similarity executes in two main phases:

- Control phase
- Data path phase

6.20.9.3.1 Control Phase

The state starts by connecting to the storage target, then waiting for a client connection. Once all connections are established the application waits for the appropriate control commands:

- Query storage
- Init storage
- Start storage

Processing each control command follows a similar pattern of:

- Relay the command to the storage target
- Wait for the storage target to respond
- Do the required post processing and consistency checks on the storage responses
- Respond to the client

The start storage control command will kick off the data path phase. Data threads will begin executing while the main thread proceeds to wait for the final control messages to complete the application lifecycle:

- Stop storage
- Shutdown

6.20.9.3.2 Data Path Phase

This phase happens per thread and involves each thread performing the requested IO operations requested by the client. Read and write requests are simply forwarded to the storage target, no actual processing is carried out by the data threads.

6.20.9.3.2.1 Read Data Flow

The regular read flow consists of the stages detailed in the following subsections.

1. Initiator Request

1. The initiator sends an I/O request to the zero copy application.
2. The zero copy application forwards the request verbatim to the storage target



2. RDMA Transfer

1. The storage target performs a RDMA write operation



3. Target Response

1. The zero copy application receives a response from the storage target
2. The zero copy application forwards the request verbatim to the initiator



6.20.9.3.2.2 Write Data Flow

1. Initiator Request

1. The initiator sends an I/O request to the zero copy application.
2. The zero copy application forwards the request verbatim to the storage target



2. RDMA Transfer

The storage target performs a RDMA read operation.



3. Target Response

1. The zero copy application receives a response from the storage target
2. The zero copy application forwards the request verbatim to the initiator



6.20.9.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Comch
- DOCA RDMA

6.20.9.5 Compiling the Application

This application is compiled as part of the set of storage applications. For compilation instructions, refer to the DOCA Storage page.

6.20.9.6 Running the Application

6.20.9.6.1 Application Execution

This application can only run within the NVIDIA® BlueField® DPU.

DOCA Storage Comch to RDMA Zero Copy is provided in source form. Therefore, compilation is required before the application can be executed.

- Application usage instructions:

```
Usage: doca_storage_comch_to_rdma_zero_copy [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -d, --device                      Device identifier
  -r, --representor                 Device host side representor identifier
  --cpu                             CPU core to which the process affinity can be set
  --storage-server                  Storage server addresses in <ip_addr>:<port> format
  --command-channel-name            Name of the channel used by the doca_comch_client. Default:
"doca_storage_comch"
  --control-timeout                 Time (in seconds) to wait while performing control operations. Default:
5
```

This usage printout can be printed to the command line using the **-h** (or **--help**) options:

```
./doca_storage_comch_to_rdma_zero_copy -h
```

For additional information, refer to section "Command-line Flags".

- CLI example for running the application on the BlueField:

```
./doca_storage_comch_to_rdma_zero_copy -d 03:00.0 -r 3b:00.0 --storage-server 172.17.0.1:12345 --cpu 0
```

Both the DOCA Comch device PCIe address (**03:00.0**) and the DOCA Comch device representor PCIe address (**3b:00.0**) should match the addresses of the desired PCIe devices.

Storage target IP **address:port** tuples should be updated to refer to the running storage target applications.

6.20.9.6.2 Command-line Flags

6.20.9.6.2.1 General Flags

Short Flag	Long Flag	Description
-h	**--help**	Prints a help synopsis and exits
-v	**--version**	Prints program version information and exits

Short Flag	Long Flag	Description
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.20.9.6.2.2 Program Flags

Short Flag	Long Flag	Description
`d`	`device`	DOCA device identifier. One of: • PCIe address: `3b:00.0` • InfiniBand name: `mlx5_0` • Network interface name: `en3f0pf0sf0` ⚠ This flag is a mandatory.
`r`	`representor`	DOCA Comch device representor PCIe address ⚠ This flag is a mandatory.
N/A	`--cpu`	Index of CPU to use. One data path thread is spawned per CPU. Index starts at 0. ⚠ The user can specify this argument multiple times to create more threads. ⚠ This flag is a mandatory.

Short Flag	Long Flag	Description
N/A	`--storage-server`	IP address and port to use to establish the control TCP connection to the target. ⚠ This flag is a mandatory.
N/A	`--command-channel-name`	Allows customizing the server name used for this application instance if multiple comch servers exist on the same device.
N/A	`--control-timeout`	Time, in seconds, to wait while performing control operations

6.20.9.6.3 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.20.9.7 Application Code Flow

The flow of the application is broken down into key functions / steps:

```
zero_copy_app app{parse_cli_args(argc, argv)};

storage::install_ctrl_c_handler([&app]() {
    app.abort("User requested abort");
});

app.connect_to_storage();
app.wait_for_comch_client_connection();
app.wait_for_and_process_query_storage();
app.wait_for_and_process_init_storage();
app.wait_for_and_process_start_storage();
app.wait_for_and_process_stop_storage();
app.wait_for_and_process_shutdown();
app.display_stats();
```

6.20.9.7.1 Main/Control Thread Flow

1. ```
   zero_copy_app app{parse_cli_args(argc, argv)};
   ```
   Parse CLI arguments and use these to create the application instance. Initial resources are also created at this stage:

   a. ```
      DOCA_LOG_INFO("Open doca_dev: %s", m_cfg.device_id.c_str());
      m_dev = storage::open_device(m_cfg.device_id);
      ```
      Open a **doca_dev** as specified by the CLI argument: **-d** or **--device**

   b. ```
      DOCA_LOG_INFO("Open doca_dev_rep: %s", m_cfg.representor_id.c_str());
      m_dev_rep = storage::open_representor(m_dev, m_cfg.representor_id);
      ```
      Open a **doca_dev_rep** as specified by the CLI argument: **-r** or **--representor**

   c. ```
      m_storage_control_channel =
      storage::control::make_tcp_client_control_channel(m_cfg.storage_server_address);
      ```

Create TCP client control channels (Control channel objects provide a unified API so that a TCP client, TCP server, **doca_comch_client**, and **doca_comch_server** all have a consistent API)

> See **storage_common/control_channel.hpp** for more information about the control channel abstraction.

d.
```
m_client_control_channel =
    storage::control::make_comch_server_control_channel(m_dev,
                                m_dev_rep,
                                m_cfg.command_channel_name.c_str(),
                                this,
                                new_comch_consumer_callback,
                                expired_comch_consumer_callback);
```

Create a Comch server control channel (Containing a **doca_comch_server** instance) using the device, representor and channel name as specified by the CLI argument **--command-channel-name** or the default value if none was specified.

2.
```
storage::install_ctrl_c_handler([&app]() {
    app.abort("User requested abort");
});
```

Set a signal handler for Ctrl+c keyboard inputs so the app can shutdown gracefully.

3.
```
app.connect_to_storage();
```

Connect to the TCP server hosted by the storage target as defined by the CLI argument: **--storage-server**

a.
```
void zero_copy_app::connect_to_storage(void)
{
    while (!m_storage_control_channel->is_connected()) {
        std::this_thread::sleep_for(std::chrono::milliseconds{100});
        if (m_abort_flag) {
            throw storage::runtime_error{DOCA_ERROR_CONNECTION_ABORTED,
                                "Aborted while connecting to storage"};
        }
    }
}
```

Poll the storage target control channel until either it connects, or the user aborts the application.

4.
```
app.wait_for_comch_client_connection();
```

Wait for a **doca_comch_client** to connect.

a.
```
void zero_copy_app::wait_for_comch_client_connection(void)
{
    while (!m_client_control_channel->is_connected()) {
        std::this_thread::sleep_for(std::chrono::milliseconds{100});
        if (m_abort_flag) {
            throw storage::runtime_error{DOCA_ERROR_CONNECTION_ABORTED,
                                "Aborted while connecting to client"};
        }
    }
}
```

Poll the Comch server control channel until a **doca_comch_client** has connected, or the user aborts the application. If any further Comch client attempts to connect to the server it will be automatically rejected by the control channel which is designed for a 1:1 relationship between clients and servers. A sleep is placed in this loop as it

may take the user / operator a few seconds to start the client so there is no gain to polling any faster.

5. ```
   app.wait_for_and_process_query_storage();
   ```

   Wait for the initiator to send a **query_storage_request** control message and then perform the required actions to fulfill the request:

   a. Forward the query storage request to the storage target.
   b. Wait for storage target to respond.
   c. Send a response to the initiator:
      i. Send a **start_storage_response** message upon success or an **error_response** message if anything failed

6. ```
   app.wait_for_and_process_init_storage();
   ```

   Wait for the initiator to send a **init_storage_request** control message and then perform the required actions to fulfill the request:

   a. use the **init_storage_payload** data to:
      i. Set core count (**m_core_count**) as the number of cores requested by the initiator (number of **--cpu** arguments provided to the initiator) OR fail if this is more than the number of **--cpu** arguments provided to the service.
      ii. Set number of transactions per core (**m_transaction_count**) to: the number of transactions requested by the initiator doubled. This is doubled to allow for batched task submission and avoid race conditions where the initiator can see a response to a transaction and try to re-submit it before the associated Comch producer event callback is received by the server meaning the initiator will continually re-try to send the task and degrade performance until the service catches up and re-submits the consumer task. This should be uncommon, but to make sure it can never happen double the transaction count is allocated so even if every single transaction on the initiator hit this issue there is a full second set of transactions ready on the service side to receive the tasks and avoid any contention. A user can experiment with reducing this value to save memory if desired).
      iii. Import then re-export initiator IO blocks mmap, this allows the storage target to read / write directly to / from the initiator memory.
      iv. Send init storage request to storage target using:
         1. The service transaction count (double the initiator value).
         2. The initiator core count.
         3. The re-exported IO blocks mmap.
      v. Send a response to the initiator:
         1. Send a **init_storage_response** message upon success or an **error_response** message if anything failed
      vi. Perform the first stages of the worker threads initialization. These steps are carried out for each thread, but only one thread performs the steps at any time this simplifies the sending and receiving of control messages, the user could modify this flow to execute in parallell if they so desired.

1. Create thread bound to the N^{th} CPU provided to the service via the `--cpu` CLI arguments
2.
```
m_workers[ii].execute_control_command(
    worker_create_objects_control_command {
        m_dev,
        m_client_control_channel->get_comch_connection(),
        m_transaction_count}
    );
```
Initialize thread context (asychronously)
3.
```
connect_rdma(ii, storage::control::rdma_connection_role::io_data, cid);
```
Create RDMA data connections (asynchronously) The thread will connect to the storage target and create a RDMA context which will be idle from the service's point of view, but is used by the storage target to perform RDMA read / write operations. See the DOCA Storage Target RDMA Application Guide for an explanation why there are two RDMA contexts per thread
4.
```
connect_rdma(ii, storage::control::rdma_connection_role::io_control, cid);
```
Create RDMA data connections (asynchronously) The thread will connect to the storage target and create a RDMA context which will be used to exchange IO requests and responses using RDMA send/recv tasks. See the DOCA Storage Target RDMA Application Guide for an explanation why there are two RDMA contexts per thread.

7.
```
app.wait_for_and_process_start_storage();
```
Wait for the initiator to send a **start_storage_request** control message and then perform the required actions to fulfill the request:
    a. Forward the start storage request to the storage target.
    b. Wait for storage target to respond.
    c. Signal all work threads to begin data path operation.
    d. Send a response to the initiator:
        i. Send a **start_storage_response** message upon success or an **error_response** message if anything failed.
8. Data path execution takes place now until either the user abort the program or a stop message is received.
9.
```
app.wait_for_and_process_stop_storage();
```
Wait for the initiator to send a **stop_storage_request** control message and then perform the required actions to fulfill the request:
    a. Forward the stop storage request the storage target.
    b. Wait for storage target to respond.
    c. Signal all work threads to stop data path operation.
    d. Collect run time stats.
    e. Send a response to the initiator:

i. Send a **stop_storage_response** message upon success or an **error_response** message if anything failed.

10. `app.wait_for_and_process_shutdown();`

    Wait for the initiator to send a **shutdown_request** control message and then perform the required actions to fulfill the request:
    a. Forward the shutdown request the storage target.
    b. Wait for storage target to respond.
    c. Destroy worker thread objects.
    d. Send a response to the initiator:
        i. Send a **stop_storage_response** message upon success or an **error_response** message if anything failed.

11. `app.display_stats();`

    Display runtime statistics.
12. Application destructor is triggered:
    a. Destroy control channels.
    b. Destroy initiator IO blocks **doca_mmap**.
    c. Close **doca_dev_rep**.
    d. Close **doca_dev**.
13. Program exits.

6.20.9.7.2 Worker/Data Path Thread Flow

The work thread proc executes in two phases: Control / configuration phase, followed by data path phase where read, write, and recovery operations take place.

6.20.9.7.2.1 Worker Init Process

```
void zero_copy_app_worker::thread_proc(zero_copy_app_worker *self, uint16_t core_idx) noexcept
```

The worker starts by executing a loop of:

1. Lock mutex.
2. If message pointer is not null:
    a. Process the configuration message.
    b. Set the operation result.
3. Unlock the mutex.
4. Yield.

The following configuration operations can be performed by the worker thread:

1. `void zero_copy_app_worker::create_worker_objects(worker_create_objects_control_command const &cmd)`

    Create general worker objects:
    a. Create IO message memory.

b. Create IO message mmap (to allow the messages to be accessed by DOCA comch and DOCA RDMA).
c. Allocate doca buffer inventory.
d. Create **doca_pe** to drive the DOCA contexts (**doca_rdma** , **doca_comch_consumer** , **doca_comch_producer**).
e. Create **doca_comch_consumer** and **doca_comch_producer** contexts.
f. Initialize and start contexts.

2. ```
   void zero_copy_app_worker::export_local_rdma_connection_blob(worker_export_local_rdma_connection_command &cmd)
   ```
   Export RDMA context connection binary blob.

3. ```
   void zero_copy_app_worker::import_remote_rdma_connection_blob(worker_import_local_rdma_connection_command const &cmd)
   ```
   Import remote RDMA context connection binary blob.

4. ```
   void zero_copy_app_worker::are_contexts_ready(worker_are_contexts_ready_control_command &cmd) const noexcept
   ```
   Poll all contexts to check they are ready to perform data path operations.

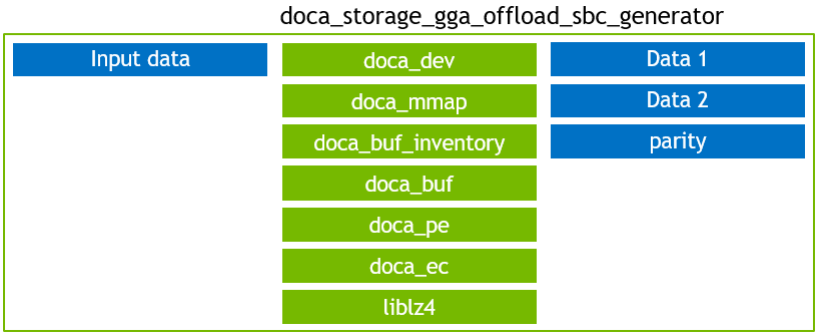
5. ```
   void zero_copy_app_worker::prepare_tasks(worker_prepare_tasks_control_command const &cmd)
   ```
   Prepare transaction contexts by:
   a. Allocating **doca_buf** objects.
   b. Allocating **doca_task** objects.
   c. Setting task user data.

6. ```
   void zero_copy_app_worker::start_data_path(void)
   ```
   Break out of the wait for configuration event loop and start the data path loop.

> After the configuration phase the mutex is not used again

After breaking out of the initial configuration loop, the thread submits receive tasks (comch consumer tasks, and RDMA recv tasks) then enters the data path function: **run_data_path_ops** .

6.20.9.7.2.2 Worker Data Path Process

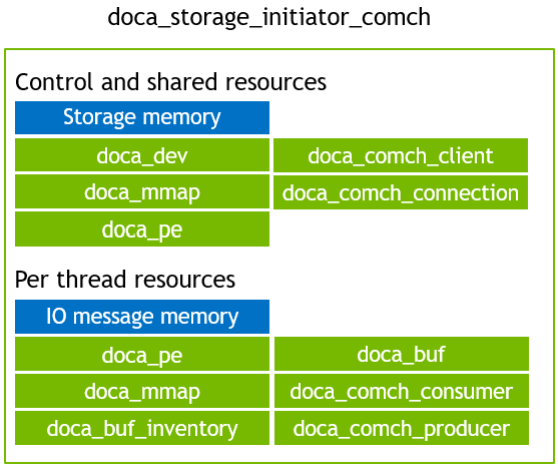
```
void zero_copy_app_worker::run_data_path_ops(zero_copy_app_worker::hot_data &hot_data)
{
    DOCA_LOG_INFO("Core: %u running", hot_data.core_idx);

    while (hot_data.run_flag) {
        doca_pe_progress(hot_data.pe) ? ++(hot_data.pe_hit_count) : ++(hot_data.pe_miss_count);
    }

    while (hot_data.error_flag == false && hot_data.in_flight_transaction_count != 0) {
        doca_pe_progress(hot_data.pe) ? ++(hot_data.pe_hit_count) : ++(hot_data.pe_miss_count);
    }

    DOCA_LOG_INFO("Core: %u complete", hot_data.core_idx);
}
```
```

During the data path phase the thread simply polls the `doca_pe` as quickly as possible to check for a task completion from one of the thread DOCA contexts (`doca_rdma` , `doca_comch_consumer` , `doca_comch_producer`) The interesting work is done in the callbacks of these tasks. The flow will always start with a consumer task completion. This is the reception of the IO message from the initiator. For the zero copy use-case the callbacks simply forward the IO requests:

- Comch consumer → RDMA send
- RDMA recv → Comch producer

6.20.9.8 References

- `/opt/mellanox/doca/applications/storage/`

6.20.10 DOCA Storage GGA Offload SBC Generator Application Guide

6.20.10.1 Introduction

The `doca_storage_gga_offload_sbc_generator` application provides a utility for generating storage-ready binary content files for use with DOCA storage target applications in the GGA offload use case.

This tool processes a single input file, breaks it into chunks, compresses the data, generates parity, and outputs three `.sbc` files suitable for use by the GGA Offload pipeline.

6.20.10.2 System Design

The application performs the following operations in sequence:

1. Load input data from disk.
2. Compress data using the LZ4 compression algorithm.
3. Generate parity data via error correction coding (2:1 ratio).
4. Write the resulting binary output files to disk, organized into three partitions.

6.20.10.3 Architecture

The `doca_storage_gga_offload_sbc_generator` is not performance-critical and follows a straightforward, linear processing flow. The application comprises the following key steps:



1. Load the source file from disk.
2. Divide the file content into chunks.
3. Compress each chunk using the LZ4 library.
    - If a chunk is not compressible enough (i.e., it cannot be reduced by at least the size of the metadata header and trailer), the application reports an error and exits.
4. Wrap each compressed chunk with a metadata header and trailer to form a storage block.
    - This metadata enables the GGA Offload application to correctly reconstruct decompression tasks.
5. Generate EC (Erasure Coding) parity for each storage block.
    - The parity is generated at a 2:1 ratio: for every 2 bytes of data, 1 byte of parity is produced. This allows for 50% of the data to be lost and still be recoverable using the parity.
6. Split the content into three logical partitions:
    - Data 1 and Data 2 are currently identical copies.
    - Parity contains duplicated parity data.

    > ⚠ This replication strategy is for simplicity and does not reflect a realistic storage deployment.
7. Write each partition to disk, including high-level metadata such as:
    - Storage block size
    - Number of storage blocks

6.20.10.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Erasure Coding

6.20.10.5 Compiling the Application

This application is compiled as part of the set of storage applications. For compilation instructions, refer to the DOCA Storage page.

6.20.10.6 Running the Application

6.20.10.6.1 Application Execution

DOCA Storage GGA Offload SBC Generator is provided in source form. Therefore, compilation is required before the application can be executed.

- Application usage instructions:

```
Usage: doca_storage_gga_offload_sbc_generator [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -d, --device                      Device identifier
  --original-input-data             File containing the original data that is represented by the storage
  --block-size                      Size of each block. Default: 4096
  --matrix-type                     Type of matrix to use. One of: cauchy, vandermonde Default: vandermonde
  --data-1                          First half of the data in storage
  --data-2                          Second half of the data in storage
  --data-p                          Parity data (used to perform recovery flow)
```

This usage printout can be printed to the command line using the **-h** (or **--help**) options:

```
./doca_storage_gga_offload_sbc_generator -h
```

For additional information, refer to section "Command-line Flags".

- CLI example for running the application on the BlueField:

```
./doca_storage_gga_offload_sbc_generator -d 03:00.0 --original-input-data original_data.txt --block-size
4096 --data-1 data_1.sbc --data-2 data_2.sbc --data-p data_p.sbc
```

The device PCIe address (**03:00.0**) should match the addresses of the desired PCIe device.

6.20.10.6.2 Command-line Flags

6.20.10.6.2.1 General Flags

Short Flag	Long Flag	Description
-h	**--help**	Prints a help synopsis and exits
-v	**--version**	Prints program version information and exits

Short Flag	Long Flag	Description
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.20.10.6.2.2 Program Flags

Short Flag	Long Flag	Description
`d`	`device`	DOCA device identifier. One of: • PCIe address: `3b:00.0` • InfiniBand name: `mlx5_0` • Network interface name: `en3f0pf0sf0` ⚠ This flag is a mandatory.
N/A	`--original-input-data`	File containing the original data that is represented by the storage ⚠ This flag is a mandatory.
N/A	`--block-size`	IP address and port to use to establish the control TCP connection to the target. ⚠ This flag is a mandatory.
N/A	`--data-1`	File in which to store the data 1 partition ⚠ This flag is a mandatory.
N/A	`--data-2`	File in which to store the data 2 partition ⚠ This flag is a mandatory.

Short Flag	Long Flag	Description
N/A	`--data-p`	File in which to store the parity partition ⚠ This flag is a mandatory.
N/A	`--matrix-type`	Type of matrix to use. One of: • `cauchy` • `vandermonde`

6.20.10.6.3 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.20.10.7 Application Code Flow

6.20.10.7.1 General Application Flow

The high-level flow of the `doca_storage_gga_offload_sbc_generator` application proceeds as follows:

1. The high level application flow is as follows:

```
auto const cfg = parse_cli_args(argc, argv);
gga_offload_sbc_gen_app app{cfg.device_id, cfg.ec_matrix_type, cfg.block_size};
```

2. Load input data from disk:

```
auto input_data = storage::load_file_bytes(cfg.original_data_file_name);
```

3. Pad input data to ensure it aligns with the block size:

```
pad_input_to_multiple_of_block_size(input_data, cfg.block_size);
```

4. Transform the data (compression, metadata wrapping, parity generation):

```
app.generate_binary_content(input_data);
```

5. Write the output partitions to disk:

```
storage::write_binary_content_to_file(
    cfg.data_1_file_name,
    storage::binary_content{cfg.block_size, results.block_count, std::move(results.data_1_content)}
);

storage::write_binary_content_to_file(
    cfg.data_2_file_name,
    storage::binary_content{cfg.block_size, results.block_count, std::move(results.data_2_content)}
);

storage::write_binary_content_to_file(
    cfg.data_p_file_name,
    storage::binary_content{cfg.block_size, results.block_count, std::move(results.data_p_content)}
);
```

6.20.10.7.2 Transform Process

1. Compress the chunk using LZ4:

```
auto const compresed_size =
    m_lz4_ctx.compress(
        input_data.data() + (ii * m_block_size),
        m_block_size,
        m_compressed_bytes_buffer.data() + metadata_header_size,
        m_compressed_bytes_buffer.size() - metadata_overhead_size
    );
```

2. Create and insert the metadata header:

```
storage::compressed_block_header const hdr{
    htobe32(m_block_size),
    htobe32(compresed_size),
};

std::copy(
    reinterpret_cast<char const *>(&hdr),
    reinterpret_cast<char const *>(&hdr) + sizeof(hdr),
    m_compressed_bytes_buffer.data()
);
```

3. Set input buffer for parity generation and reset the output buffer:

```
doca_buf_set_data(m_input_buf, m_compressed_bytes_buffer.data(), m_block_size);
doca_buf_reset_data_len(m_output_buf);
```

4. Submit the parity generation task to the DOCA EC engine:

```
doca_task_submit(doca_ec_task_create_as_task(m_ec_task));
```

6.20.10.8 References

- **/opt/mellanox/doca/applications/storage/**

6.20.11 DOCA Storage Initiator ComCh Application Guide

6.20.11.1 Introduction

The **doca_storage_initiator_comch** application functions as a client to the DOCA storage service. It benchmarks the performance of data transfers over the ComCh interface, providing detailed measurements of throughput, bandwidth, and latency.

6.20.11.2 System Design

The **doca_storage_initiator_comch** application performs the following key tasks:

- Initiates a sequence of I/O requests to exercise the DOCA storage service.
- Measures and reports storage performance, including:
    - Millions of I/O operations per second (IOPS)
    - Effective data bandwidth
    - I/O operation latency:
        - Minimum latency
        - Maximum latency

- Mean latency

The application establishes a connection to the storage service running on an NVIDIA® BlueField® platform using the **doca_comch_client** interface.

6.20.11.3 Architecture

The application is logically divided into two functional areas:

- Control-time and shared resources
- Per-thread data path resources



The execution consists of two main phases:

- Control Phase
- Data Path Phase

6.20.11.3.1 Control Phase

The control phase is responsible for establishing and managing the storage session lifecycle. It performs the following steps:

1. Query storage – Retrieve storage service metadata.
2. Init storage – Initialize the storage window and prepare for data transfers.
3. Start storage – Begin the data path phase and launch data threads.

Once the data phase is initiated, the main thread waits for test completion. Afterward, it issues final cleanup commands:

1. Stop storage – Stop data movement.
2. Shutdown – Tear down the session and release resources.

6.20.11.3.2 Data Path Phase

The selected `--execution-strategy` is used to pre-initialise a set of per task flags. These flags are used to decide which action the task should take next. This allows the task to perform any combination of; read, write, memory set, memory clear, and validate. The order of exectuion of each action is hard coded for simplicity. The order of actions is:

1. Initialise transaction:
    a. Set actions bitmask to match `--execution-strategy`.
    b. Get a local memory address from the pool (of initiator IO blocks assigned to this thread).
    c. Get a remote address from the pool (of storage IO blocks assigned to this thread).
2. Set memory to initial value:
    a. Copy content from the data provided by `--storage-plain-content` at the address defined by the storage IO block and place it in the memory defined by initiator IO block.
3. Start an IO write:
    a. Send an IO write operation request.
    b. Wait for the IO response.
4. Clear current memory content:
    a. Clear the Initiator memory defined by initiator IO block.
5. Start an IO read:
    a. Send an IO write operation request.
    b. Wait for the IO response.
6. Validate memory content:
    a. Compare the content of the initiator IO block against the expected value in `--storage-plain-content`.
7. Finish transaction:
    a. Calculate and record stats.
    b. Return initiator IO block and storage IO block to the pool.
    c. End the test if the `--run-limit-operation-count` has been reached.
    d. Recycle the transaction (go back to step 1).

6.20.11.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Comch

6.20.11.5 Compiling the Application

This application is compiled as part of the set of storage applications. For compilation instructions, refer to the DOCA Storage page.

6.20.11.6 Running the Application

6.20.11.6.1 Application Execution

This application can only run from the host.

DOCA Storage Initiator Comch is provided in source form. Therefore, compilation is required before the application can be executed.

- Application usage instructions:

```
Usage: doca_storage_initiator_comch [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -d, --device                      Device identifier
  --cpu                             CPU core to which the process affinity can be set
  --storage-plain-content           File containing the plain data that is represented by the storage
  --execution-strategy              Define what to run. One of: read_throughput_test |
write_throughput_test | read_write_data_validity_test | read_only_data_validity_test
  --run-limit-operation-count       Run N operations (per thread) then stop. Default: 1000000
  --transaction-count               Number of concurrent IO transactions (per thread) to use. Default: 64
  --command-channel-name            Name of the channel used by the doca_comch_client. Default:
"doca_storage_comch"
  --control-timeout                 Time (in seconds) to wait while performing control operations. Default:
5
  --local-io-region-size            Size in bytes of the local region. Value can be 0 meaning automatically
size to the same size as the storage target. Default: 0
  --blocks-per-io                   Use multiple blocks per IO request. Default: 1
```

This usage printout can be printed to the command line using the **-h** (or **--help**) options:

```
./doca_storage_initiator_comch -h
```

For additional information, refer to section "Command-line Flags".

- CLI example for running the application on the BlueField:

```
./doca_storage_initiator_comch -d 3b:00.0 --execution-strategy read_throughput_test --run-limit-operation-
count 1000000 --cpu 0
```

Both the DOCA Comch device PCIe address (**3b:00.0**) should match the addresses of the desired PCIe devices.

6.20.11.6.2 Command-line Flags

6.20.11.6.2.1 General Flags

Short Flag	Long Flag	Description
`-h`	`--help`	Prints a help synopsis and exits
`-v`	`--version`	Prints program version information and exits
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.20.11.6.2.2 Program Flags

Short Flag	Long Flag	Description
`d`	`device`	DOCA device identifier. One of: • PCIe address: `3b:00.0` • InfiniBand name: `mlx5_0` • Network interface name: `en3f0pf0sf0` ⚠ This flag is a mandatory.
N/A	`--execution-strategy`	The data path routine to run. One of: • `read_throughput_test` • `write_throughput_test` • `read_only_data_validity_test` • `read_write_data_validity_test` ⚠ This flag is a mandatory.

Short Flag	Long Flag	Description
N/A	`--cpu`	Index of CPU to use. One data path thread is spawned per CPU. Index starts at 0. ⚠ The user can specify this argument multiple times to create more threads. ⚠ This flag is a mandatory.
N/A	`--storage-plain-content`	Expected plain content that is expected to be read from storage during a `read_only_data_validity_test`
N/A	`--run-limit-operation-count`	Number of IO operations to perform when performing a throughput test
N/A	`--transaction-count`	Number of parallel tasks per thread to use
N/A	`--command-channel-name`	Allows customizing the server name used for this application instance if multiple comch servers exist on the same device.
N/A	`--control-timeout`	Time, in seconds, to wait while performing control operations
N/A	`--local-io-region-size`	Set the size of the local memory region size, allows for asymetric memory setups where the local application using the storage engine has more or less memory thatn the storage target.
N/A	`--blocks-per-io`	Allow for testing the case where multiple sequential data blocks are read or written in a single IO operation.

6.20.11.6.3 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.20.11.7 Application Code Flow

The flow of the application is broken down into key functions / steps:

```
initiator_comch_app app{parse_cli_args(argc, argv)};

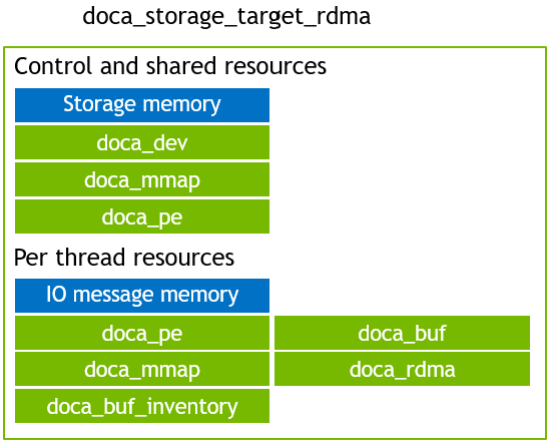
storage::install_ctrl_c_handler([&app]() {
    app.abort("User requested abort");
});

app.connect_to_storage_service();
app.query_storage();
app.init_storage();
app.prepare_threads();
app.start_storage();
auto const run_success = app.run();
app.join_threads();
app.stop_storage();

if (run_success) {
    app.display_stats();
} else {
    exit_value = EXIT_FAILURE;
    fprintf(stderr, "+================================================+\n");
```

```
    fprintf(stderr, "| Test failed!!\n");
    fprintf(stderr, "+================================================+\n");
}

app.shutdown();
```

6.20.11.7.1 Main/Control Thread Flow

1. ```
   initiator_comch_app app{parse_cli_args(argc, argv)};
   ```

   Parse CLI arguments and use these to create the application instance. Initial resources are also created at this stage:

   a. ```
      DOCA_LOG_INFO("Open doca_dev: %s", m_cfg.device_id.c_str());
      m_dev = storage::open_device(m_cfg.device_id);
      ```

      Open a **doca_dev** as specified by the CLI argument: **-d** or **--device**

   b. ```
      m_service_control_channel =
          storage::control::make_comch_client_control_channel(m_dev,
                                      m_cfg.command_channel_name.c_str(),
                                      this,
                                      new_comch_consumer_callback,
                                      expired_comch_consumer_callback);
      ```

      Create a comch client control channel (Containing a **doca_comch_client** instance) using the device and channel name as specified by the CLI argument **--command-channel-name** or the default value if none was specified. (Control channel objects provide a unified API so that a TCP client, TCP server, **doca_comch_client**, and **doca_comch_server** all have a consistent API)

      > See **storage_common/control_channel.hpp** for more information about the control channel abstraction.

2. ```
   storage::install_ctrl_c_handler([&app]() {
       app.abort("User requested abort");
   });
   ```

   Set a signal handler for control+c keyboard inputs so the app can shutdown gracefully.

3. ```
   app.connect_to_storage_service();
   ```

   Connect to the **doca_comch_server** in the service.

4. ```
   app.query_storage();
   ```

   Query the storage details:

   a. Send a **query_storage_request** to the service.
   b. Wait for the service to respond.
   c. Set the iop size (**m_effective_block_size**) based on the reported storage block size and the **--blocks-per-io** CLI argument.
   d. If performing a **read_write_data_validity_test** create storage for the "expected content" (if **--storage-plain-content** CLI argument was not provided).

e. Verify that the size of the "expected content" matches the size of the reported storage capacity.

5.
```
app.init_storage();
```

Prepare to use the storage:

a. Allocate IO block memory.
b. Create IO block **doca_mmap**.
c. Export IO block **doca_mmap**.
d. Send **init_storage_request** to the service.
e. Wait for the service to respond.
f. Create worker thread objects.
g. Configure the ops actions mask based on the **--execution-strategy** CLI argument
h. Partition local and storage IO blocks so that each worker gets a unique region of the IO blocks. This avoids race conditions when performing data validation. In a real use-case each thread could be allowed to access any region of the memory delegating the responsibility for the neccessary concurrency control to a higher level.
i. Perform the first stages of the worker threads initialization. These steps are carried out for each thread, but only one thread performs the steps at any time this simplifies the sending and receiving of control messages, the user could modify this flow to execute in parallell if they so desired.
    i. Create thread bound to the N^{th} CPU provided to the service via the **--cpu** CLI arguments
    ii. Init the worker thread:

> Init actions are performed by the thread itself after its core affinity has been set to improve memory allocation efficiency as the memory allocator is NUMA aware so allocating from the core that will be using the memory allows the allocator to pick memory that is local / closest to that core

        1. Create local IO address pool.
        2. Create remote IO address pool.
        3. Allocation IO messages memory.
        4. Allocate N transaction contexts.
        5. Create IO messages **doca_mmap**.
        6. Create **doca_buf_inventory**.
        7. Create **doca_pe**.
        8. Create **doca_comch_consumer**.
        9. Create **doca_comch_producer**.
j. Wait for all threads to be ready:
    i. Consumer and producer contexts are connected.

6.
```
app.prepare_threads();
```

Prepare thread for data path operations:

a. Prepare **doca_buf** objects.
b. Prepare **doca_task** objects.
c. Initialise IO message content.

7.
```
app.start_storage();
```
Start the storage service:
a. Send a **start_storage_request** to the service.
b. Wait for the service to respond.

8.
```
app.run();
```
Run the initiator data path:
a. Signal all worker threads to end configuration phase and begin data path phase.
b. Data path execution takes place now until either the user abort the program or a stop message is received.
c. Infinitly loop; polling workers utill all workers have finished performing data path operations.
d. Collect workers exit codes.
e. If all workers completed successfully collect runtime stats.

9.
```
app.join_threads();
```
Join the worker threads.

10.
```
app.stop_storage();
```
Stop the storage service:
a. Send a **stop_storage_request** to the service.
b. Wait for the service to respond.

11.
```
if (run_success) {
    app.display_stats();
} else {
    exit_value = EXIT_FAILURE;
    fprintf(stderr, "+=================================================+\n");
    fprintf(stderr, "| Test failed!!\n");
    fprintf(stderr, "+=================================================+\n");
}
```
If the worker threads all succeeded (encountered no errors while running data path operations) display stats, else display an error banner

12.
```
app.shutdown();
```
Destroy resources:
a. Destroy worker objects.
b. Send **shutdown_request** to the service.
c. Wait for storage consumers to complete de-registration (Don't tear down the **comch_client** ↔ **comch_server** yet as they use that to de-register)

13. Application destructor is triggered:
a. IO **doca_mmap** is destroyed
b. IO blocks memory is destroyed

c. Comch client control channel is destroyed (**doca_comch_client is destroyed**)

d. Close **doca_dev**

14. Application exits

6.20.11.8 References

- **/opt/mellanox/doca/applications/storage/**

6.20.12 DOCA Storage Target RDMA Application Guide

6.20.12.1 Introduction

The **doca_storage_target_rdma** application provides a simple, volatile memory-backed RDMA storage target. It is designed to interact with a DOCA storage service, offering direct read and write access to a dedicated memory region using RDMA.

6.20.12.2 System Design

The **doca_storage_target_rdma** application performs the following core functions:

- Exposes a memory-backed storage region for use by the storage service client.
- Handles RDMA-based I/O operations (reads and writes) using the DOCA RDMA library.

To perform these tasks, the application acts as a TCP server, waiting for an incoming connection from a storage service initiator. Once connected, it handles control and data path interactions as in this page.

6.20.12.3 Architecture

The application is divided into two main functional areas:

- Control-Time and Shared Resources – Includes TCP server setup, memory registration, and RDMA connection handling.
- Per-Thread Data Path Resources – Includes thread-local RDMA resources and task management structures.



The application executes in two main phases:

- Control Phase
- Data Path Phase

6.20.12.3.1 Control Phase

This phase begins when a connection is established with a storage service over TCP. The application then processes a sequence of control commands:

1. Query Storage
    - Reports the size and layout of the exposed storage region.
2. Init Storage
    - Validates the requested number of worker threads.
    - Allocates and registers local memory for RDMA operations.
    - Imports remote memory handles provided by the initiator.
    - Creates internal worker objects for task execution.
3. Wait for RDMA Connection Requests
    - Waits for one RDMA connection per requested core/thread.
    - Ensures each RDMA session is fully established before proceeding.
4. Start Storage
    - Waits for all RDMA connections to be ready.
    - Submits initial tasks to prepare the data path phase.
    - Launches the data path threads.

Once the Start Storage command is received and all threads are active, the application transitions to the data path phase. The main thread remains active, waiting for final control commands:

1. Stop Storage
    - Terminates active data threads cleanly.
2. Shutdown
    - Performs cleanup and resource deallocation, shutting down the application.

6.20.12.3.2 Data Path Phase

Each data path thread performs I/O processing independently based on client requests. The typical per-thread flow is:

1. Receive I/O Request
    - Handle incoming requests from the initiator via RDMA.
2. Perform RDMA Operation
    - Depending on the request type, either:
        - Execute an RDMA read from the local memory region.
        - Execute an RDMA write to the local memory region.
3. Send I/O Response
    - Return a response back to the initiator, indicating success or failure of the operation.

6.20.12.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA RDMA

6.20.12.5 Compiling the Application

This application is compiled as part of the set of storage applications. For compilation instructions, refer to the DOCA Storage page.

6.20.12.6 Running the Application

6.20.12.6.1 Application Execution

> DOCA Storage Target RDMA is provided in source form. Therefore, compilation is required before the application can be executed.

- Application usage instructions:

```
Usage: doca_storage_target_rdma [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                      Print a help synopsis
  -v, --version                   Print program version information
  -l, --log-level                 Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                 Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>               Parse command line flags from an input json file

Program Flags:
  -d, --device                    Device identifier
  --cpu                           CPU core to which the process affinity can be set
  --listen-port                   TCP listen port number
  --binary-content                Path to binary.sbc file containing the initial content to be
represented by this storage instance
  --block-count                   Number of available storage blocks. (Ignored when using content binary
file) Default: 128
  --block-size                    Block size used by the storage. (Ignored when using content binary
file) Default: 4096
```

This usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
./doca_storage_target_rdma -h
```

For additional information, refer to section "Command-line Flags".

- CLI example for running the application:

```
./doca_storage_target_rdma -d 3b:00.0 --listen-port 12345 --block-size 4096 --block-count 64 --cpu 0
```

The user DOCA device PCIe address (`3b:00.0`) should match the addresses of the desired PCIe device.

6.20.12.6.2 Command-line Flags

6.20.12.6.2.1 General Flags

Short Flag	Long Flag	Description
`-h`	`--help`	Prints a help synopsis and exits
`-v`	`--version`	Prints program version information and exits
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.20.12.6.2.2 Program Flags

Short Flag	Long Flag	Description
`d`	`device`	DOCA device identifier. One of: • PCIe address: `3b:00.0` • InfiniBand name: `mlx5_0` • Network interface name: `en3f0pf0sf0` ⚠ This flag is a mandatory.
N/A	`--cpu`	Index of CPU to use. One data path thread is spawned per CPU. Index starts at 0. ⚠ The user can specify this argument multiple times to create more threads. ⚠ This flag is a mandatory.
N/A	`--listen-port`	Port to listen upon for incomming TCP connections ⚠ This flag is a mandatory.
N/A	`--binary-content`	Path to a file to be used to provide initial content to the storage instance.
N/A	`--block-count`	Number of storage blocks to provide
N/A	`--block-size`	Size of each storage block

ℹ A user should provide one of:

- `--binary-content` : Where the file is a.sbc file
  - The sbc file provides storage dimmensions and data to populate the blocks

OR (Random / uninitialisaed bytes with a user defined dimmension)

- `--block-count`
- `--block-size`

OR (Initialised bytes with a user defined dimmension)

- `--block-count`
- `--block-size`
- `--binary-content` : Where the file is plain content to be distributed across the storage and its size == block count * block size

6.20.12.6.3 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.20.12.7 Application Code Flow

The flow of the application is broken down into key functions / steps:

```
target_rdma_app app{parse_target_rdma_app_cli_args(argc, argv)};

storage::install_ctrl_c_handler([&app]() {
    app.abort("User requested abort");
});

app.wait_for_client_connection();
app.wait_for_and_process_query_storage();
app.wait_for_and_process_init_storage();
app.wait_for_and_process_create_rdma_connections();
app.wait_for_and_process_start_storage();
app.wait_for_and_process_stop_storage();
app.wait_for_and_process_shutdown();
app.display_stats();
```

6.20.12.7.1 Main/Control Thread Flow

1. ```
   initiator_comch_app app{parse_cli_args(argc, argv)};
   ```

   Parse CLI arguments and use these to create the application instance. Initial resources are also created at this stage:

   a.

   Open a doca_dev as specified by the CLI argument: -d or **--device**.

   b. ```
      m_storage_block_count = m_cfg.block_count;
      m_storage_block_size = m_cfg.block_size;

      auto const page_size = storage::get_system_page_size();
      m_local_io_region_size = uint64_t{m_storage_block_count} * m_storage_block_size;
      m_local_io_region = static_cast<uint8_t *>(storage::aligned_alloc(page_size,
      m_local_io_region_size));
      ```

      Allocate storage IO blocks memory.

   c. ```
      if (!m_cfg.content.empty()) {
          std::copy(std::begin(m_cfg.content), std::end(m_cfg.content), m_local_io_region);
      }
      ```

      Copy user defined content into the storage IO blocks memory if it was provided.

   d. ```
      m_control_channel = storage::control::make_tcp_server_control_channel(m_cfg.listen_port);
      ```

      Create a TCP server control channel using the listen port specified by the CLI argument **--listen-port**. (Control channel objects provide a unified API so that a TCP client, TCP server, **doca_comch_client**, and **doca_comch_server** all have a consistent API).

> See **storage_common/control_channel.hpp** for more information about the control channel abstraction.

2.
```
storage::install_ctrl_c_handler([&app]() {
    app.abort("User requested abort");
});
```
Set a signal handler for control+c keyboard inputs so the app can shutdown gracefully.

3.
```
app.wait_for_client_connection();
```
Wait for a TCP client (the storage service) to connect.

4.
```
app.wait_for_and_process_query_storage();
```
Wait for the storage service to send a **query_storage_request** control message and then perform the required actions to fulfill the request:

a. Populate a **query_storage_response** with the storage capacity (block count * block size) and the block size.
b. Send response to the storage service.

5.
```
app.wait_for_and_process_init_storage();
```
Wait for the storage service to send a **init_storage_request** control message and then perform the required actions to fulfill the request:

a. Use the **init_storage_payload** data to:
   i. Set core count (**m_core_count**) as the number of cores requested by the initiator (number of **--cpu** arguments provided to the initiator) OR fail if this is more than the number of **--cpu** arguments provided to the service.
   ii. Set number of transactions per core (**m_num_transactions**).
   iii. Create **doca_mmap** object from the provided export blob.
   iv. Create **m_core_count** worker threads.
   v. Perform the first stages of the worker threads initialization. These steps are carried out for each thread, but only one thread performs the steps at any time this simplifies the sending and receiving of control messages, the user could modify this flow to execute in parallel if they so desired.
      1. Create thread bound to the N^{th} CPU provided to the service via the **--cpu** CLI arguments.
      2.
```
m_workers[ii].execute_control_command(
    worker_create_objects_control_command {
        m_dev,
        m_transaction_count,
        m_remote_io_mmap,
        m_local_io_mmap}
    );
```
      Initialize thread context (asychronously).
   vi. Send a response to the storage service:
      1. Send an **init_storage_response** message upon success or an **error_response** message if anything failed.

6.
```
app.wait_for_and_process_create_rdma_connections();
```
Wait for the storage service to send a **create_rdma_connection_request** control message and then perform the required actions to fulfill the request:

   a.
```
void const *blob = nullptr;
size_t blob_size = 0;

doca_rdma_export(rdma_ctx, &blob, &blob_size, &rdma_conn);
```
   Create local side of the RDMA connection by exporting RDMA context and store the created connection blob.

   b.
```
doca_rdma_connect(rdma_ctx, cmd.import_blob.data(), cmd.import_blob.size(), rdma_conn);
```
   Connect to remote RDMA side of the RDMA connection.

   > This RDMA connection activity will be performed twice per worker thread. This allows for the IO control messages to be transferred over one connection while the other is used for data transfer. The reason for this is to reduce latency as with a single connection **doca_rdma_task_send**, **doca_rdma_task_read**, and **doca_rdma_task_write** tasks execute in submission order. This could mean that if, for example, 32 IO requests arrive in close succession and the corresponding 32 read / write tasks are submitted, once the first operation completes its response is blocked until all of the remaining 31 data transfers complete. This can cause significant delay in responding to the service / initiator and greatly increases the round trip latency of a storage request. Having two connections means in the previous scenario when the first of the N tasks completes its IO response can be sent immediately while the remaining N-1 read / write operations continue to execute.

   c. Send a response to the storage service:
      i. Send a **create_rdma_connection_response** message upon success or an **error_response** message if anything failed.

7.
```
app.wait_for_and_process_start_storage();
```
Wait for the storage service to send a **start_storage_request** control message and then perform the required actions to fulfill the request:

   a.
```
doca_error_t result;
doca_ctx_states ctx_state;

result = doca_ctx_get_state(doca_rdma_as_ctx(rdma_ctx), &ctx_state);
if (result == DOCA_SUCCESS && ctx_state == DOCA_CTX_STATE_RUNNING) {
    // RDMA connection is ready
}
```
   Poll all worker connections until RDMA connections are fully established.
   b. Signal all work threads to begin data path operation.
   c. Send a response to the storage service:
      i. Send a **start_storage response** message upon success or an **error_response** message if anything failed.

8. Data path execution takes place now until either the user abort the program or a stop message is received.
9. ```
   app.wait_for_and_process_stop_storage();
   ```
   Wait for the storage service to send a **stop_storage_request** control message and then perform the required actions to fulfill the request:
   a. Join all work threads.
   b. Send a response to the storage service:
      i. Send a **stop_storage_response** message upon success or an **error_response** message if anything failed.
10. ```
    app.wait_for_and_process_shutdown();
    ```
    Wait for the storage service to send a **shutdown_request** control message and then perform the required actions to fulfill the request:
    a. Collect runtime stats.
    b. Destroy workers.
    c. Send a response to the storage service:
       i. Send a **shutdown_response** message upon success or an **error_response** message if anything failed.
11. ```
    app.display_stats();
    ```
    Display runtime statistics.
12. Application destructor
    a. Destroy TCP control channel
    b. Destroy **doca_mmap** objects.
    c. Destroy IO blocks memory.
    d. Close **doca_dev** .
13. Application exits.

6.20.12.7.2 Worker/Data Path Thread Flow

The work thread proc executes in two phases: Control / configuration phase, followed by data path phase where read and write operations take place.

6.20.12.7.2.1 Worker Init Process

```
void target_rdma_worker::thread_proc(target_rdma_worker *self, uint16_t core_idx) noexcept
```

The worker starts by executing a loop of:

1. Lock mutex
2. If message pointer is not null:
   a. Process the configuration message.
   b. Set the operation result.
3. Unlock the mutex.
```

4. Yield.

The following configuration operations can be performed by the worker thread:

1. ```
   void target_rdma_worker::create_worker_objects(worker_create_objects_control_command const &cmd)
   ```
   Create the objects required by the worker to carry out data path operations:
   a. Create IO messages memory.
   b. Create IO messages **doca_mmap** (to allow the messages to be accessed by RDMA)..
   c. Create **doca_buf_inventory**.
   d. Create **doca_pe** to drive the DOCA contexts.
   e. Create two **doca_rdma** contexts.
   f. Configure and start RDMA contexts.

2. ```
   void target_rdma_worker::create_rdma_connection(worker_create_rdma_connection_command &cmd)
   ```
   Create an RDMA connection from one of the two RDMA contexts held by the worker (which one to use is specified by the command):
   a. Call **doca_rdma_connect** to create the connection object, and connection blob.
   b. Call **doca_rdma_connect** to start connecting to the remote side of the connection.
   c. Store the local connection blob in the command object for use by the control thread to respond to the service.

3. ```
   void target_rdma_worker::are_contexts_ready(worker_are_contexts_ready_control_command &cmd) const noexcept
   ```
   Check each RDMA context and report when both are both ready for use.

4. ```
   void target_rdma_worker::prepare_tasks(void)
   ```
   Final preparations for data path execution:
   a. Cache frequently used static values into hot data (a single cache line).
   b. Allocate transaction objects.
   c. Allocate **doca_buf** objects.
   d. Allocate **doca_task** objects.
   e. Set task user data.
   f. Set task pointers into the transaction context.

5. ```
   void target_rdma_worker::start_data_path()
   ```
   Break out of the wait for configuration event loop and start the data path loop.

> After the configuration phase the mutex is not used again

After breaking out of the initial configuration loop, the thread submits receive tasks (RDMA recv tasks) then enters the data path function: **run_data_path_ops**.
   ```

6.20.12.7.2.2 Worker Data Path Process

```
void target_rdma_worker::run_data_path_ops(target_rdma_worker::hot_data &hot_data)
{
    DOCA_LOG_INFO("Core: %u running", hot_data.core_idx);

    while (hot_data.run_flag) {
        doca_pe_progress(hot_data.pe) ? ++(hot_data.pe_hit_count) : ++(hot_data.pe_miss_count);
    }

    while (hot_data.error_flag == false && hot_data.in_flight_transaction_count != 0) {
        doca_pe_progress(hot_data.pe) ? ++(hot_data.pe_hit_count) : ++(hot_data.pe_miss_count);
    }
}
```

During the data path phase the thread simply polls the **doca_pe** as quickly as possible to check for a task completion from one of the RDMA contexts. The interesting work is done in the callbacks of these tasks. The flow will always start with a RDMA recv task completion. This is the reception of the IO message from the storage service.

doca_comch_consumer_task_post_recv_cb

```
void target_rdma_worker::doca_rdma_task_receive_cb(doca_rdma_task_receive *task,
                                                   doca_data task_user_data,
                                                   doca_data ctx_user_data) noexcept
```

This callback is the main driver of the storage target. It carries out the following steps:

1. ```
   auto *const io_message = storage::get_buffer_bytes(doca_rdma_task_receive_get_dst_buf(task));
   auto const message_type = storage::io_message_view::get_type(io_message);
   ```
   Interpret the type of IO request that has been received.

2. ```
   char *const remote_addr = hot_data->remote_memory_start_addr +
   storage::io_message_view::get_requester_offset(io_message);

   char *const local_addr = hot_data->local_memory_start_addr +
   storage::io_message_view::get_storage_offset(io_message);

   uint32_t const transfer_size = storage::io_message_view::get_io_size(io_message);
   ```
   Extract IO addresses and transfer size from the IO request.

3. ```
   auto *const transfer_ctx = static_cast<transfer_context *>(task_user_data.ptr);
   ```
   Get a reference to the transaction object from the task user data.

4. ```
   switch (message_type) {
       case storage::io_message_type::read: {
           oca_buf_inventory_buf_reuse_by_data(transfer_ctx->storage_buf, local_addr, transfer_size);
           doca_buf_inventory_buf_reuse_by_addr(transfer_ctx->host_buf, remote_addr, transfer_size);

           doca_task_submit(doca_rdma_task_write_as_task(transfer_ctx->write_task));
       } break;
       case storage::io_message_type::write: {
           doca_buf_inventory_buf_reuse_by_data(transfer_ctx->host_buf, remote_addr, transfer_size);
           doca_buf_inventory_buf_reuse_by_addr(transfer_ctx->storage_buf, local_addr, transfer_size);

           doca_task_submit(doca_rdma_task_read_as_task(transfer_ctx->read_task));
       } break;
   ```
   Submit the appropriate RDMA task.
   a. IO read is moving data from storage IO blocks to the remote IO blocks; so from the storage point of view its a write.
   b. IO write is moving data from the remote IO blocks to the storage IO blocks; so from the storage point of view its a read.

5. If the any action in this process has failed send an IO error response to the caller, otherwise wait for the RDMA read / write task to complete.

on_transfer_complete

```
void target_rdma_worker::on_transfer_complete(doca_task *task,
                                              doca_data task_user_data,
                                              doca_data ctx_user_data) noexcept
{
    auto *const hot_data = static_cast<target_rdma_worker::hot_data *>(ctx_user_data.ptr);
    auto *const response_task = static_cast<doca_rdma_task_send *>(task_user_data.ptr);
    auto *const io_message =
        storage::get_buffer_bytes(const_cast<doca_buf *>(doca_rdma_task_send_get_src_buf(response_task)));

    ++(hot_data->completed_transaction_count);

    storage::io_message_view::set_type(storage::io_message_type::result, io_message);
    storage::io_message_view::set_result(DOCA_SUCCESS, io_message);

    auto const ret = doca_task_submit(doca_rdma_task_send_as_task(response_task));
    if (ret != DOCA_SUCCESS) {
        DOCA_LOG_ERR("Failed submit response task: %s", doca_error_get_name(ret));
}
```

A callback that is re-used for both **doca_rdma_task_read** and **doca_rdma_task_write**. The work is the same regardless of which task type completed and more detailed task type info is not required. This callback will simply update the IO message to change its type to a response, and set the result to success. The IO message is then sent back to the storage service.

doca_rdma_task_send_cb

```
void target_rdma_worker::doca_rdma_task_send_cb(doca_rdma_task_send *task,
                                                doca_data task_user_data,
                                                doca_data ctx_user_data) noexcept
{
    auto *const request_task = static_cast<doca_rdma_task_receive *>(task_user_data.ptr);

    doca_buf_reset_data_len(doca_rdma_task_receive_get_dst_buf(request_task));

    auto const ret = doca_task_submit(doca_rdma_task_receive_as_task(request_task));
    if (ret != DOCA_SUCCESS) {
        DOCA_LOG_ERR("Failed re-submit request task: %s", doca_error_get_name(ret));
    }

    auto *const hot_data = static_cast<target_rdma_worker::hot_data *>(ctx_user_data.ptr);
    --(hot_data->in_flight_transaction_count);
}
```

The RDMA task send callback resets the task data and resubmits the recv task allowing the transaction to be reused.

6.20.12.8 References

- **/opt/mellanox/doca/applications/storage/**

6.21 DOCA Stream Receive Performance Application Guide

This guide outlines the implementation of the DOCA Stream Receive Performance application, built on top of the NVIDIA® BlueField® DPU.

6.21.1 Introduction

The Stream Receive Performance application is designed to measure and evaluate RX performance using the NVIDIA DOCA RMAX library. It leverages the capabilities of DOCA RMAX and NVIDIA Rivermax to support efficient, high-performance media and data streaming.

6.21.1.1 Key Technologies

- DOCA RMAX API – A component of the NVIDIA DOCA framework, optimized for networking tasks in media streaming use cases.
- NVIDIA Rivermax SDK – Built to exploit BlueField DPU hardware acceleration, enabling direct data transfers between the NIC and GPU, minimizing CPU load.

This architecture delivers high throughput, ultra-low latency, and minimal CPU utilization making it an ideal solution for demanding real-time streaming workloads.

6.21.1.2 Deployment Notes

- DOCA Rivermax applications must run on BlueField target DPUs with root privileges or other additional permissions and capabilities
- Ensure the DPU has a valid IP address configured
- Allocate an appropriate number of huge pages for optimal performance. Refer to "*Rivermax Performance-oriented Development Guidelines*" for details.

  > To access this document, join the NVIDIA Rivermax SDK developers' program and access documentation in the Rivermax Developer page.

- Runtime configurations can be tuned even after the application starts, allowing dynamic performance optimization

For complete setup steps and advanced configurations, refer to DOCA RMAX documentation.

6.21.2 System Design

The application is designed to receive and process network packets using the DOCA library. It is structured around three core components:

- Configuration management – Manages the initialization, parsing, validation, and cleanup of application configuration parameters
- Global resources management – Handles the allocation and management of shared resources such as memory maps, buffer inventories, and progress engines
- Stream management – Manages the lifecycle of data streams used for packet reception, including setup, execution, and teardown

6.21.3 Architecture

The architecture comprises several key modules and their responsibilities.

6.21.3.1 Main Application

- Initialization – Sets up logging, parses command-line arguments, and initializes the configuration.
- Device listing – If the `--list` flag is passed, it enumerates and prints available devices, then exits.
- Stream processing – Initializes global resources, configures the stream, and enters the packet reception loop.

6.21.3.2 Configuration Management

- Initialization – Applies default values and creates the CPU affinity mask.
- Argument parsing – Parses command-line arguments and updates the configuration accordingly.
- Validation – Verifies that all required parameters are provided.
- Destruction – Frees any configuration-related resources.

6.21.3.3 Global Resources Management

- Initialization – Sets up shared memory maps, buffer inventories, and progress engines required for data handling
- Destruction – Cleans up and releases global resources

6.21.3.4 Stream Management

- Initialization – Configures and starts the stream, allocates memory buffers, and attaches the necessary flows
- Packet reception loop – Processes incoming packets, manages events, and collects runtime statistics
- Destruction – Detaches flows, stops the stream, and releases associated buffers

6.21.3.5 Application Functions and Roles

Function(s)	Role
`main`	Entry point of the application, handles overall flow control
`init_config` ; `destroy_config`	Manage application configuration
`register_argp_params`	Register command-line arguments
`init_globals` ; `destroy_globals`	Manage global resources
`init_stream` ; `destroy_stream`	Manage stream setup and teardown
`run_recv_loop`	Main loop for receiving and processing packets
`handle_completion` ; `handle_error`	Event handlers for packet reception and errors

6.21.3.6 Data Structures

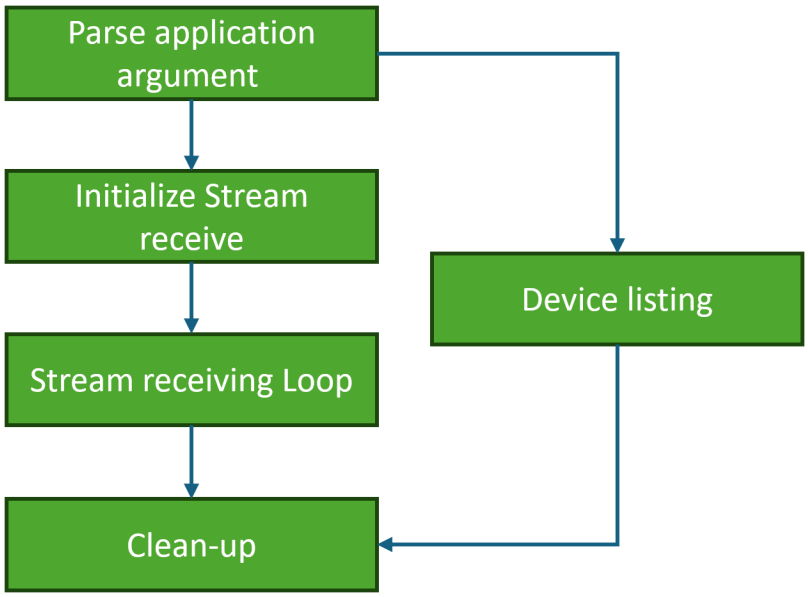
- `app_config` – Holds configuration parameters for the application
- `globals` – Holds global resources required by the application
- `stream_data` – Manages the state and data associated with streaming

6.21.3.7 Event Handling

- Completion Events – Handled by `handle_completion`, updates statistics and optionally dumps packet content
- Error Events – Handled by `handle_error`, logs errors and stops the receive loop

6.21.3.8 Flow

1. Initialization – Set up logging, configuration, and global resources.
2. Device listing – Optionally list available devices.
3. Stream setup – Configure and initialize the stream.
4. Packet reception – Enter the main loop to receive and process packets.
5. Teardown – Clean up resources and exit.



6.21.4 DOCA Libraries

This application leverages the following DOCA library:

- DOCA RMAX

6.21.5 Dependencies

The RMAX library must be compiled and run, and a Rivermax license is required to run this application, as is the case with every application using DOCA RMAX. Refer to NVIDIA Rivermax SDK page to obtain that license.

6.21.6 Compiling the Application

> Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

> For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for the application is located in the following directory:

```
/opt/mellanox/doca/applications/stream_receive_perf/
```

6.21.6.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_stream_receive_perf is created under **/tmp/build/stream_receive_perf/** .

6.21.6.2 Compiling Only the Current Application

1. To directly build only the stream receive performance application:

   ```
   cd /opt/mellanox/doca/applications/
   meson /tmp/build -Denable_all_applications=false -Denable_stream_receive_perf=true
   ninja -C /tmp/build
   ```

   doca_stream_receive_perf is created under **/tmp/build/stream_receive_perf/** .
2. Alternatively, one can set the desired flags in the **meson_options.txt** file instead of providing them in the compilation command line:
   a. Edit the following flags in **/opt/mellanox/doca/applications/meson_options.txt** :
      - Set **enable_all_applications** to **false**
      - Set **enable_stream_receive_perf** to **true**

b. The same compilation commands should be used, as were shown in the previous section:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_stream_receive_perf is created under **/tmp/build/stream_receive_perf/** .

6.21.7 Running the Application

6.21.7.1 Prerequisites

This application can run on the target DPU only.

This application must be run with root privileges or other additional permissions and capabilities.

For detailed instructions on running without root privileges, please refer to the Environment section in DOCA Rivermax.

- An IP address to the device being used must be set up .
- It is recommended to have at least 800 huge pages enabled to achieve maximum performance:

```
dpu> echo 1000000000 > /proc/sys/kernel/shmmax
dpu> echo 800 > /proc/sys/vm/nr_hugepages
```

6.21.7.2 Application Execution

The stream receive performance application is provided in source form, hence a compilation is required before the application can be executed.

- Application usage instructions

```
 Usage: doca_stream_receive_perf  [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
   --list                            List available devices
   --scatter-type                    Scattering type: RAW (default) or ULP
   --tstamp-format                   Timestamp format: raw (default), free-running or synced
    -s, --src-ip                      Source address to read from
    -d, --dst-ip                      Destination address to bind to
    -i, --local-ip                    IP of the local interface to receive data
    -p, --dst-port                    Destination port to read from
    -K, --packets                     Number of packets to allocate memory for (default 262144)
    -y, --payload-size                Packet's payload size (default 1500)
    -e, --app-hdr-size                Packet's application header size (default 0)
    -a, --cpu-affinity                Comma separated list of CPU affinity cores for the application main
thread
    --sleep                           Amount of microseconds to sleep between requests (default 0)
    --min                             Block until at least this number of packets are received (default 0)
    --max                             Maximum number of packets to return in one completion
    --dump                            Dump packet content
```

For additional information, please refer to the "Command Line Flags" section below.
The above usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
./doca_stream_receive_perf -h
```

- CLI example for listing available devices:

```
./doca_stream_receive_perf --list
```

- CLI example for receiving a stream sent from `1.1.63.5` to the local NIC address `1.1.64.67` and port `7000`:

```
./doca_stream_receive_perf --local-ip 1.1.64.67 --dst-ip 1.1.64.67 --src-ip 1.1.63.5 --dst-port 7000
```

- CLI example for receiving a stream receiving a stream sent on `239.0.0.1` to the local NIC `1.1.64.67` from `1.1.63.5` and port `7000`:

```
./doca_stream_receive_perf --local-ip 1.1.64.67 --dst-ip 239.0.0.1 --src-ip 1.1.63.5 --dst-port 7000
```

- CLI example for receiving a stream using header-data split mode. This example receives a stream sent from `1.1.63.5` to the local NIC address `1.1.64.67` and port `7000`. The application header size is 20 bytes, and the payload size is 1200 bytes:

```
./doca_stream_receive_perf --local-ip 1.1.64.67 --dst-ip 1.1.64.67 --src-ip 1.1.63.5 --dst-port 7000 --app-hdr-size 20 --payload-size 1200
```

> Setting the application header size enables header-data split mode which separates the application header from the payload.

6.21.7.3 Command Line Flags

6.21.7.3.1 General Flags

Short Flag	Long Flag	Description
`-h`	`--help`	Prints a help synopsis and exits
`-v`	`--version`	Prints program version information and exits

Short Flag	Long Flag	Description
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.21.7.3.2 Program Flags

Short Flag	Long Flag	Description
N/A	`list`	List all available devices, dump their IPv4 addresses, and tell whether or not the PTP clock is supported
N/A	`scatter-type`	Scattering type: • RAW (default) • ULP
N/A	`tstamp-format`	Timestamp format: • raw (default) • free-running • synced
`s`	`src-ip`	Source IP address to read from
`d`	`dst-ip`	Destination IP address to bind to
`i`	`local-ip`	IP of the local interface to receive data
`p`	`dst-port`	Destination port to read from
`K`	`packets`	Number of packets to allocate memory for (default 262144)
`y`	`payload-size`	Packet's payload size (default 1500)
`e`	`app-hdr-size`	Packet's application header size (default 0)

Short Flag	Long Flag	Description
a	cpu-affinity	list of CPU affinity cores for the application main thread
N/A	sleep	Amount of microseconds to sleep between requests
N/A	min	Block until at least this number of packets are received
N/A	max	Maximum number of packets to return in one completion
N/A	dump	Dump packet content

6.21.7.4 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.21.8 Application Code Flow

a. Parse application argument.
   i. Initialize arg parser resources and register DOCA general parameters.

```
init_config();
```

   ii. Register stream receive performance application parameters.

```
register_argp_params();
```

   iii. Parse the arguments.

```
doca_argp_start();
```

      1. Parse app parameters.

b. Device listing.
   If the list parameter is set to true, the application lists all available devices.
   i. Initializes the DOCA RMAX library.

```
doca_rmax_init();
```

   ii. Enumerates and lists all available devices.

```
list_devices();
```

c. Stream receive: if the list parameter is not set, the application proceeds to receive stream.
   i. Mandatory Arguments Check.

```
mandatory_args_set();
```

   ii. CPU Affinity Mask (if it is set).

```
doca_rmax_set_cpu_affinity_mask();
```

iii. Initializes the DOCA RMAX library.

```
doca_rmax_init();
```

iv. Device opening.

```
open_device();
```

v. Global Resources Initialization.

```
init_globals();
```

vi. Stream Initialization.

```
init_stream();
```

d. Main Loop.

```
run_recv_loop();
```

e. Clean-up.

i. Cleans up and destroys the stream.

```
destroy_stream();
```

ii. Releases and destroys global application resources.

```
destroy_globals();
```

iii. Closes the device.

```
doca_dev_close();
```

iv. Releases the DOCA RMAX library.

```
doca_rmax_release();
```

v. Destroys the ARGP resources.

```
doca_argp_destroy();
```

i. Releases resources allocated by the application configuration.

```
destroy_config();
```

6.21.9 References

- **/opt/mellanox/doca/applications/stream_receive_perf/**

6.22 DOCA Switch Application Guide

This guide provides an example of switch implementation on top of NVIDIA® BlueField® DPU.

6.22.1 Introduction

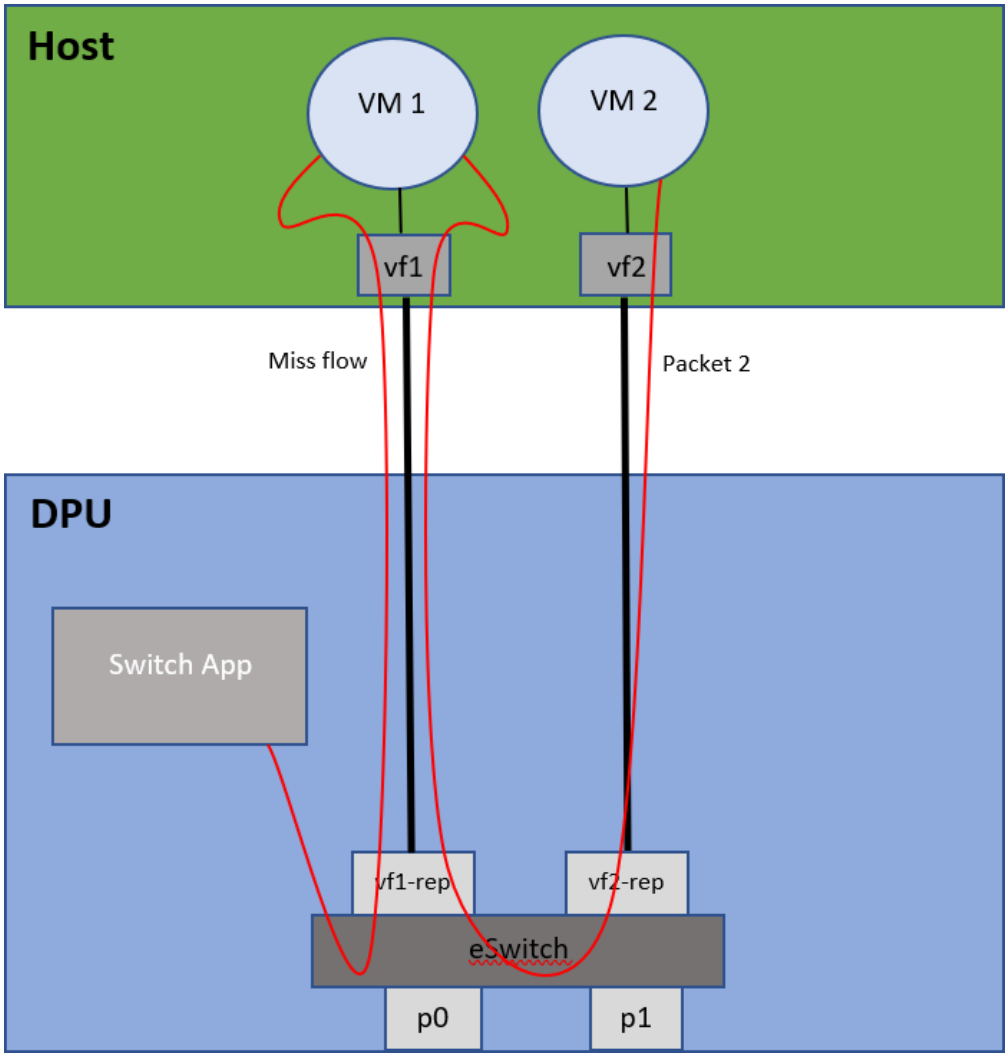
DOCA Switch is a network application that leverages the DPU's hardware capability for internal switching between representor ports on the DPU.

DOCA Switch is based on the DOCA Flow library. As such, it exposes a command line interface which receives DOCA Flow like commands to allow adding rules in real time.

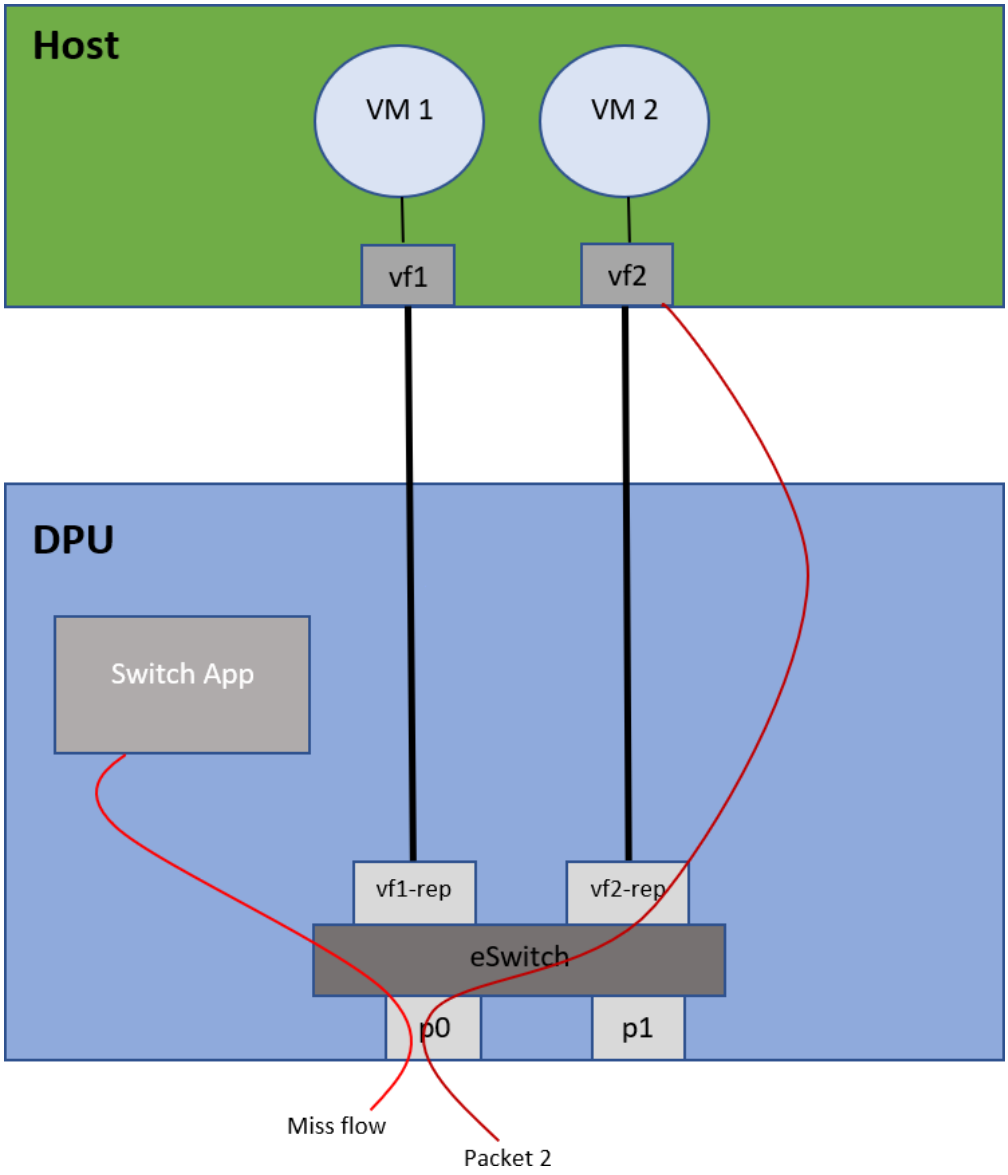
6.22.2 System Design

DOCA Switch is designed to run on the DPU as a standalone application (all network traffic goes directly through it).

Traffic flows between two VMs on the host:



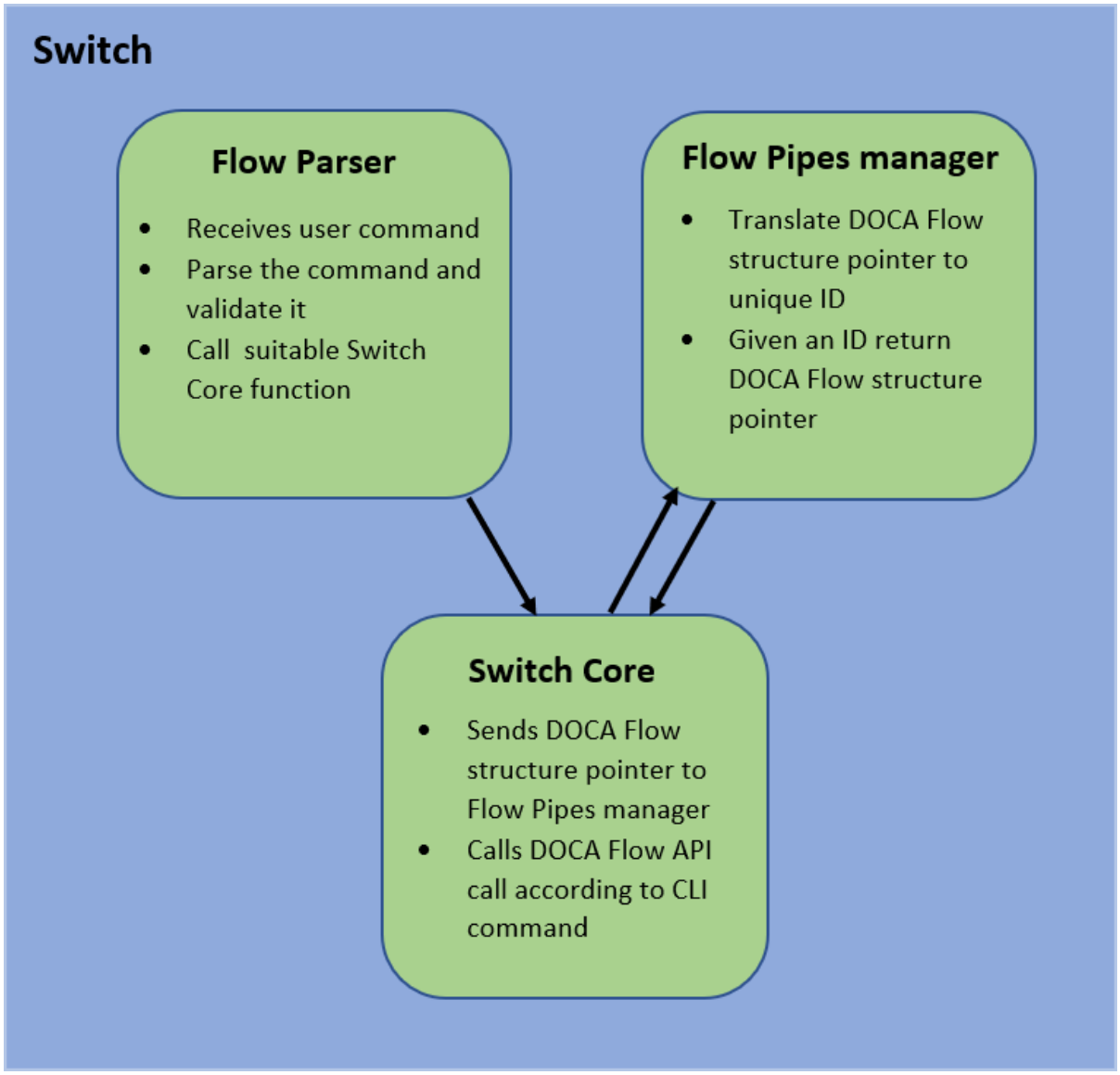
Traffic flow from a physical port to a VM on the host:



6.22.3 Application Architecture

DOCA Switch is based on 3 modules:

- Command line interface – receives pre-defined DOCA Flow-like commands and parses them
- Flow pipes manger – generates a unique identification number for each DOCA Flow structure created
- Switch core – combines all modules together and calls necessary DOCA Flow API



Port initialization cannot be made dynamically. All ports must be defined when running the application with standard DPDK flags.

- When adding a pipe or an entry, the user must run commands to create the relevant structs beforehand
- Optional parameters must be specified by the user in the command line; otherwise, `NULL` is used
- After a pipe or an entry is created successfully, the relevant ID is printed for future use

6.22.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Flow

Refer to its respective programming guide for more information.

6.22.5 Compiling the Application

> Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

> For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for the application is located in the following directory:

```
/opt/mellanox/doca/applications/switch/
```

6.22.5.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

> **doca_switch** is created under **/tmp/build/switch/**.

6.22.5.2 Recompiling Only the Current Application

To directly build only the switch application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_switch=true
ninja -C /tmp/build
```

> **doca_switch** is created under **/tmp/build/switch/**.

Alternatively, one can set the desired flags in the **meson_options.txt** file instead of providing them in the compilation command line:

1. Edit the following flags in **/opt/mellanox/doca/applications/meson_options.txt**:
   - Set **enable_all_applications** to **false**
   - Set **enable_switch** to **true**
2. Run the following compilation commands:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_switch is created under **/tmp/build/switch/**.

6.22.6 Running the Application

6.22.6.1 Prerequisites

The switch application is based on DOCA Flow. Therefore, the user is required to allocate huge pages.

```
$ echo '1024' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
$ sudo mkdir /mnt/huge
$ sudo mount -t hugetlbfs -o pagesize=2M nodev /mnt/huge
```

6.22.6.2 Application Execution

The switch application is provided in source form. Therefore, hence a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_switch [DPDK Flags] -- [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -a, --device                      device
  -r, --rep                         device representor
  -exp, --expert-mode               set expert mode
```

This usage printout can be printed to the command line using the **-h** (or **--help**) options:

```
./doca_switch -- -h
```

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on BlueField:

```
./doca_switch -- -r pci/03:00.0,pf0vf[0-1] -l 60
```

The PCIe address (`03:00.0`) should match the address of the desired PCIe device.

For additional information regarding the command line syntax of the device and device representor identifiers, please referr to the matching section in the DOCA Arg Parser Guide.

6.22.6.3 Command Line Flags

6.22.6.3.1 General Flags

Short Flag	Long Flag	Description
`-h`	`--help`	Prints a help synopsis and exits
`-v`	`--version`	Prints program version information and exits
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.22.6.3.2 Program Flags

Short Flag	Long Flag	Description
`a`	`device`	Device identifier
`r`	`rep`	Device representor identifiers for the ports indices
exp	expert-mode	Run switch in expert mode

6.22.6.4 Supported Commands

- `create pipe port_id=[port_id][,<optional_parameters>]`
  Available optional parameters:
  - `name=<pipe-name>`
  - `root_enable=[1|0]`
  - `monitor=[1|0]`
  - `match_mask=[1|0]`
  - `fwd=[1|0]`
  - `fwd_miss=[1|0]`
  - `type=[basic|control]`
- `add entry pipe_id=<pipe_id>,pipe_queue=<pipe_queue>[,<optional_parameters>]`
  Available optional parameters:
  - `monitor=[1|0]`
  - `fwd=[1|0]`
- `add control_pipe entry priority=<priority>,pipe_id=<pipe_id>,pipe_queue=<pipe_queue>[,<optional_parameters>]`
  Available optional parameters:
  - `match_mask=[1|0]`
  - `fwd=[1|0]`
- `destroy pipe pipe_id=<pipe_id>`
- `rm entry pipe_queue=<pipe_queue>,entry_id=[entry_id]`
- `port pipes flush port_id=[port_id]`
- `port pipes dump port_id=[port_id],file=[file_name]`
- `query entry_id=[entry_id]`
- `create [struct] [field=value,...]`
  Struct options: `pipe_match`, `entry_match`, `match_mask`, `actions`, `monitor`, `fwd`, `fwd_miss`
  - Match struct fields:

Fields	Field Options
`flags`	
`port_meta`	
`outer.eth.src_mac`	
`outer.eth.dst_mac`	
`outer.eth.type`	
`outer.vlan_tci`	
`outer.l3_type`	`ipv4`, `ipv6`
`outer.src_ip_addr`	
`outer.dst_ip_addr`	
`outer.l4_type_ext`	`tcp`, `udp`, `gre`

Fields	Field Options
outer.tcp.flags	FIN , SYN , RST , PSH , ACK , URG , ECE , CWR
outer.tcp_src_port	
outer.tcp_dst_port	
outer.udp_src_port	
outer.udp_dst_port	
tun_type	
vxlan_tun_id	
gre_key	
gtp_teid	
inner.eth.src_mac	
inner.eth.dst_mac	
inner.eth.type	
inner.vlan_tci	
inner.l3_type	ipv4 , ipv6
inner.src_ip_addr	
inner.dst_ip_addr	
inner.l4_type_ext	tcp , udp
inner.tcp.flags	FIN , SYN , RST , PSH , ACK , URG , ECE , CWR
inner.tcp_src_port	
inner.tcp_dst_port	
inner.udp_src_port	
inner.udp_dst_port	

- Actions struct fields:

Fields	Field Options
decap	true , false
mod_src_mac	
mod_dst_mac	
mod_src_ip_type	ipv4 , ipv6
mod_src_ip_addr	
mod_dst_ip_type	ipv4 , ipv6

Fields	Field Options
mod_dst_ip_addr	
mod_src_port	
mod_dst_port	
ttl	
has_encap	true , false
encap_src_mac	
encap_dst_mac	
encap_src_ip_type	ipv4 , ipv6
encap_src_ip_addr	
encap_dst_ip_type	ipv4 , ipv6
encap_dst_ip_addr	
encap_tup_type	vxlan , gtpu , gre
encap_vxlan-tun_id	
encap_gre_key	
encap_gtp_teid	

- FWD struct fields:

Fields	Field Options
type	rss , port , pipe , drop
rss_flags	
rss_queues	
num_of_queues	
port_id	
next_pipe_id	

- Monitor struct fields:
  - flags
  - cir
  - cbs
  - aging

The physical port number (only one physical port is supported) will always be 0 and all representor ports are numbered from 1 to N where N is the number of representors being used. For example:

- Physical port ID: 0

- VF0 representor port ID: 1
- VF1 representor port ID: 2
- VF2 representor port ID: 3

The following is an example of creating a pipe and adding one entry into it:

```
create fwd type=port,port_id=0xffff
create pipe port_id=0,name=p0_to_vf1,root_enable=1,fwd=1
create fwd type=port,port_id=1
add entry pipe_queue=0,fwd=1,pipe_id=1012
        ....
rm entry pipe_queue=0,entry_id=447
```

1. Pipe is configured on port ID 0 (physical port).
2. Entry is configured to forward all traffic directly into port ID 1 (VF0).
3. When the forwarding rule is no longer needed, the entry is deleted.
4. Ultimately, both entries are deleted, each according to the unique random ID it was given:

6.22.6.5 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

6.22.7 Application Code Flow

1. Parse application argument.
    a. Initialize the arg parser resources and register DOCA general parameters.

    ```
    doca_argp_init();
    ```

    b. Register DOCA Flow switch mode application parameters.

    ```
    register_doca_flow_switch_param();
    ```

    c. Parse app parameters.

    ```
    doca_argp_start();
    ```

2. Initiate DOCA Flow switch mode application parameters.

    ```
    init_doca_flow_switch_common();
    ```

3. Initialize DPDK ports and queues.

    ```
    dpdk_queues_and_ports_init();
    ```

4. Initialize DOCA Switch.

    ```
    switch_init();
    ```

    a. Initialize DOCA Flow.
    b. Create port pairs.
    c. Create Flow Pipes Manger module.

d. Register an action for each relevant CLI command.

5. Initialize Flow Parser.

```
flow_parser_init();
```

   a. Reset all internal Flow Parser structures.
   b. Start the command line interface.
   c. Receive user commands, parse them, and call the required DOCA Flow API command.
   d. Close the interactive shell once a "quit" command is entered.

6. Clean Flow Parser resources.

```
flow_parser_cleanup();
```

7. Destroy Switch resources.

```
switch_destroy();
```

   a. Destroy Flow Pipes Manager resources.

8. Destroy DOCA Flow.

```
switch_destroy();
```

9. Destroy DPDK ports and queues.

```
dpdk_queues_and_ports_fini();
```

10. DPDK finish.

```
dpdk_fini();
```

    a. Call **rte_eal_destroy()** to destroy initialized EAL resources.

11. Arg parser destroy.

```
doca_argp_destroy();
```

6.22.8 References

- **/opt/mellanox/doca/applications/switch/**

6.23 DOCA Accelerated UPF Application Guide

6.23.1 Terms and Abbreviations

Abbreviation	Definition
DN	Data Network
DPI	Deep Packet Inspection
FAR	Forwarding Action Rule

Abbreviation	Definition
gNB	Next Generation NodeB
GTP-U	GPRS Tunneling Protocol User Plane
N3	Interface connecting the Radio Access Network (gNB) to the User Plane Function (UPF) in 5G networks. Responsible for transporting user data using GTP-U protocol.
N6	Interface connection the User Plane Function (UPF) to external data networks (e.g. internet, enterprise network, cloud services), serving as the gateway between 5G network and external networks.
PDR	Packet Detection Rule
PDU	Protocol Data Unit
PSC	PDU Session Container. An extension header in GTP that is being used in 5G core networks.
QER	QoS Enforcement Rule
QoS	Quality of Service
RAN	Radio Access Network
RAP	Reference Application
RSS	Receive Side Scaling. A functionality that enables dispatching of packets into multiple queues.
SMF	Session Management Function
UE	User Equipment
UPF	User Plane Function
URR	Usage Reporting Rule
WAN	Wide Area Network

6.23.2 Introduction

The Accelerated UPF-RAP provides a reference implementation of an accelerated data path of UPF on top of the NVIDIA® BlueField-3®.

This project is intended for UPF vendors looking to hardware accelerate their UPF data-plane. It provides a reference implementation for developers as well as a platform to evaluate the functionality and performance. It is important to note that this project is not a UPF implementation by itself, and users should not expect it to work as a full-fledged UPF or comply fully with the 3GPP standard.

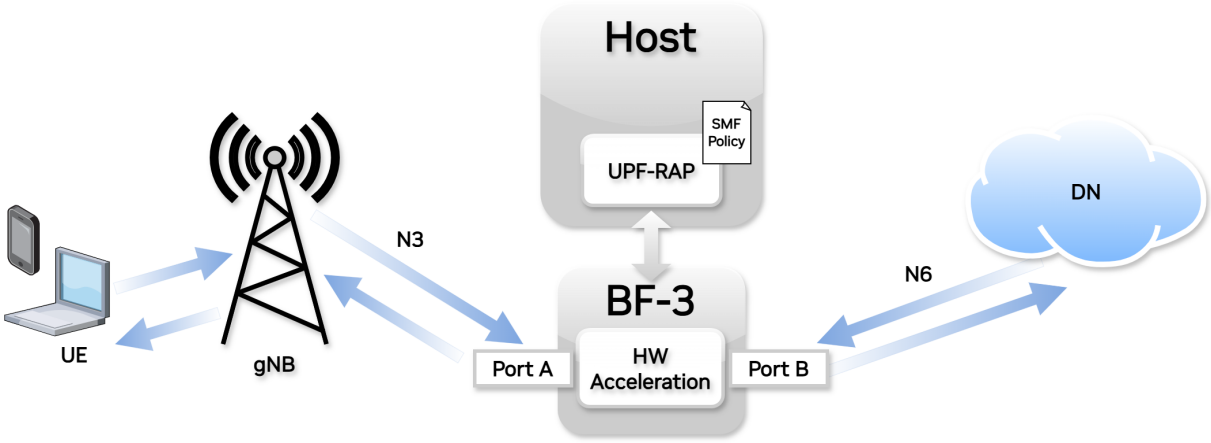
The Accelerated UPF RAP optimizes data traffic handling between RAN (gNB) and WAN (DN) by leveraging BlueField-3 hardware acceleration capabilities. Using DOCA Flow API, the application programs UPF's domain-specific network pipelines directly into BlueField-3's hardware, significantly reducing CPU overhead by processing packets in hardware rather than software.

Bidirectional Processing (Uplink, Downlink):

- RAN to DN Flow (Uplink):
  - Ingress: All N6 traffic is received on BlueField-3 Port A.

    - Processing: BlueField-3 applies appropriate processing to the N6 traffic, including GTP encapsulation, routing, policy enforcement, quota management, or any other relevant actions based on the traffic characteristics and network configuration.
    - Egress: After processing, BlueField-3 forwards the N6 traffic to Port B.
- DN to RAN Flow (Downlink):
    - Ingress: All N3 traffic is received on BlueField-3 Port B.
    - Processing: BlueField-3 processes the N3 traffic, applying actions such as GTP decapsulation, QoS enforcement, user-plane policy application, or routing as per the core network's requirements.
    - Egress: After processing, BueField-3 forwards the N3 traffic to Port A.

This flow ensures efficient and streamlined traffic processing and forwarding for both N6 and N3 interfaces, enabling robust user-plane functionality with clear traffic flow paths.



The Accelerated UPF-RAP is designed to efficiently handle and manage traffic while incorporating key features for functionality, performance, observability, and resource optimization. The detailed operation and capabilities are as follows:

HW Accelerated Features:

- Rate Meters: Configurable rate meters monitor and enforce traffic throughput on N3 and N6 interfaces.
- Quota Counters: Per-subscriber quota management is enabled, tracking data usage against assigned limits. Actions (e.g., throttling or redirecting traffic) can be applied when quotas are exceeded.
- Flow Aging: A flow aging mechanism efficiently tracks active flows and removes inactive ones based on configurable timeouts, ensuring resource efficiency and preventing stale flow accumulation.

Software Features with Hardware Assistance:

- Configurable Slow Path Packet Redirection: Allows configuring the number of packets from a flow to be processed by software before the flow is accelerated to hardware, which fits to use cases such as DPI.

- Connection Tracking Infrastructure: Maintains a shared software context for N3 and N6 flows of the same connection.
  This is achieved in a performance-oriented manner by utilizing symmetric RSS, which ensures that both flows are processed on the same core.
  This approach enables lock-free database access, enhancing efficiency and reducing latency.
- Debuggability and Telemetry: Real-time debug counters provide visibility into flow-level and packet-level operations.

The Accelerated UPF-RAP does not enforce a specific traffic type (N3/N6) on either port. Instead, it is designed to handle both traffic types on any port interchangeably.

These advanced features ensures the UPF is capable of meeting modern network demands with high efficiency, robust observability, and optimal resource management for N3 and N6 flows.

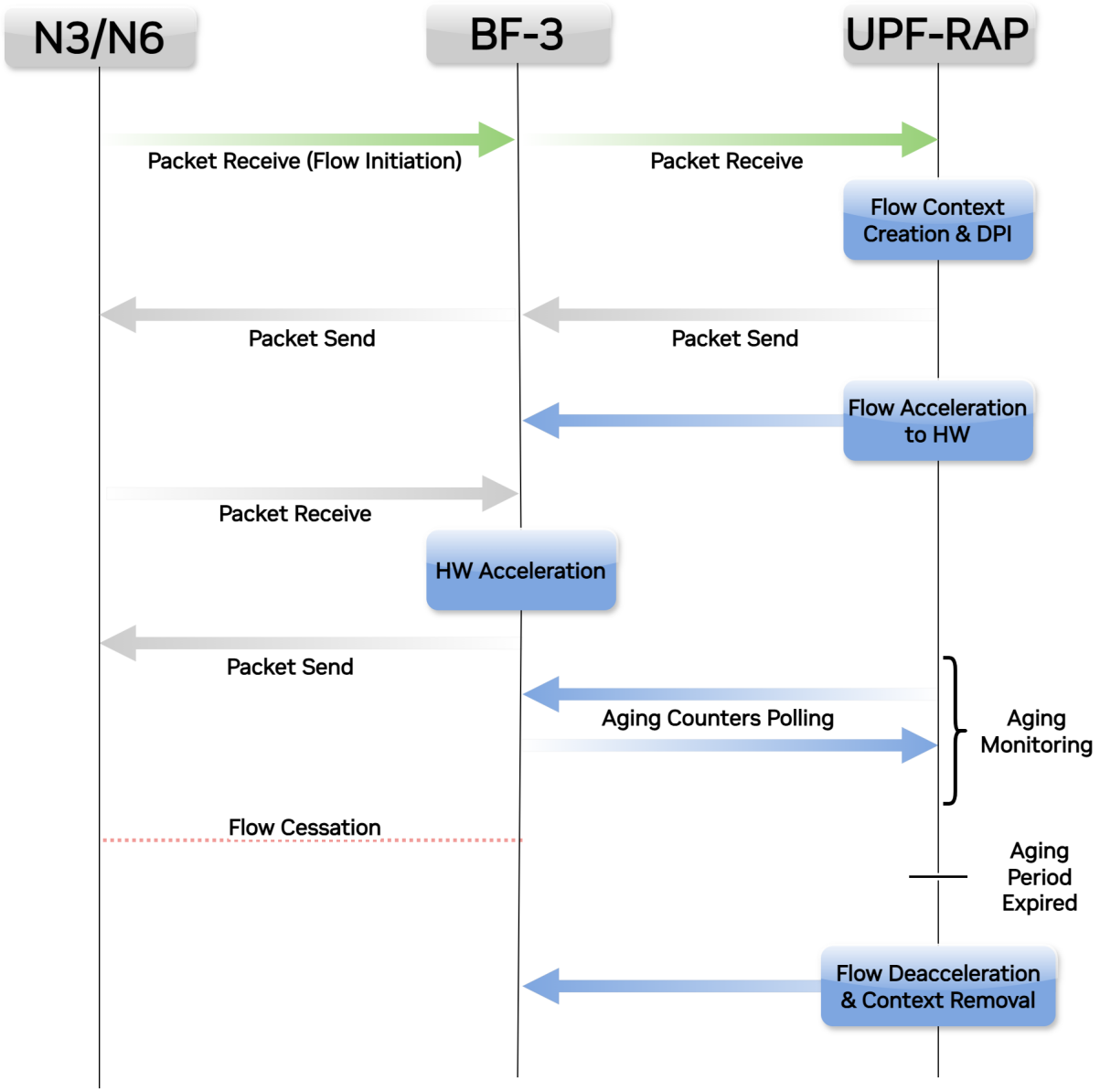
6.23.3 System Design

The Accelerated UPF-RAP runs on the host system and programs acceleration policies into the BlueField-3 acceleration engine.

6.23.4 Application Architecture

This Accelerated UPF-RAP architecture leverages DOCA Flow to efficiently manage N3 and N6 traffic by accelerating flow processing inside BlueField-3's steering mechanism.

The architecture is designed to optimize performance, reduce CPU utilization, reducing memory usage, and maintain efficient scale of flows by aging out inactive flows from the system, enabling rooms for newly added flows.



6.23.4.1 Key Features in UPF-RAP

- DOCA Flow Framework:
  - Acts as the foundational API for the Accelerated UPF-RAP, enabling direct interaction with BlueField-3 for rules (add/remove) and counters management.
  - Facilitates high-performance packet processing, flow steering, and HW acceleration operations.
- Flow Management Logic:
  - Packet Forwarding Acceleration:
    - DOCA Flow's pipes are pre-allocated with fixed size to contain future rules insertion.

        - New flows are identified by the Accelerated UPF-RAP and accelerated BlueField-3 via DOCA Flow APIs.
        - BlueField-3 handles packet forwarding, flow-specific actions (e.g., QoS), and telemetry collection.
    - Flow Aging and Removal:
        - Inactive flows are monitored using timeouts and telemetry feedback.
        - Flows that exceed their inactivity threshold are automatically removed from BlueField-3, freeing resources.
- Bidirectional Traffic Management Logic:
    - Uplink traffic (gNB → N3 → UPF → N6 → DN):
        - Received from the RAN and processed for interaction with the core network.
        - Managed via DOCA Flow to accelerate user-plane traffic to BlueField-3 for low-latency forwarding (QoS, decapsulation).
    - Downlink traffic (DN→ N6 → UPF → N3 → gNB):
        - Received from DN and processed for delivery to the RAN or other destinations.
        - Accelerated to BlueField-3 for efficient handling (QoS, encapsulation).
    - The Accelerated UPF-RAP leverages symmetric RSS to run multithreaded, enabling the following capabilities:
        - Processing N3 and N6 Traffic on the Same Core: Symmetric RSS ensures that both N3 and N6 traffic of the same flow are directed to the same CPU core.
          This is achieved by hashing packet headers in a way that packets belonging to the same flow are consistently mapped to the same core.
          This reduces the overhead of context switching and improves cache efficiency, leading to better performance.
          RSS is applied on both GTP-tunneled and non-tunneled directions of the traffic
        - Single Software Context for a Flow: With symmetric RSS, a single software context is maintained for each flow. This means that both N3 and N6 traffic of the same flow are handled within the same thread or process.
          This unified context simplifies flow management and ensures that all operations related to a flow are executed sequentially, avoiding race conditions and reducing synchronization overhead.
        - Lock-Free Data Access: To further enhance performance, the Accelerated UPF-RAP implements lock-free data structures for managing flow state information.
          This approach minimizes contention and latency, ensuring that database operations are performed efficiently.
        - RSS is applicable for new packets of a flow only, once a flow is accelerated it will be handled by HW.

6.23.4.2 SMF

The SMF is a crucial component of the 5G core network architecture. It is responsible for managing user sessions, including establishing, modifying, and terminating data sessions between UE and the DN.

The Accelerated UPF-RAP uses static SMF policies for entities creation provided by a JSON file, bypassing the need to integrate with a real SMF through N4 interface. Production-based UPF shall implement and accommodate the N4 interface in order to dynamically update the rules, following network updates.

An example configuration can be found here.

The Accelerated UPF-RAP supports the following fields:

- PDR

Field Path	Comments
`createPdr.pdrId`	PDR ID
`createPdr.pdi.sourceInterface.type`	Determines packet direction (Source Interface), supported values: 0 (Access/DL), 2 (SGi-LAN/N6-LAN).
`createPdr.pdi.localFT`	Defines the packet header matching the PDR
`createPdr.pdi.localFT.teid_start` , `createPdr.pdi.LocalFT.teid_end`	Defines GTPU TEID range to match
`createPdr.pdi.localFT.ip.v4`	Defines the outer IPV4 source/destination address to match, depending on “sourceInterface” value (2 or 0 respectively)
`createPdr.pdi.qfi`	Defines the match of QFI field in GTPU header
`createPdr.pdi.userEquipment.ip.v4`	Defines the outer IPV4 source/destination address to match, depending on “sourceInterface” value (0 or 2 respectively)
`createPdr.pdi.sdf.description`	Defines the criteria for packet matching
`createPdr.farId` , `createPdr.urrIds` , `createPdr.qerIds`	The FAR and lists of the URRs and a list of the QERs.

- FAR

Field Path	Comments
`createFar.farId`	FAR ID
`createFar.fp.outerHeader.ip.v4` , `CreateFar.fp.outerHeader.teid`	Defines the outer header that needs to be added to the packet (Outer Header Creation).

- URR

Field Path	Comment
`createUrr.urrId`	URR ID
`createUrr.volumeQuota.totalVolume`	Defines the total quota in bytes

- QER

Field Path	Comments
createQer.qurId	QER ID
createQer.maxBitRate.ulMBR , createQer.maxBitRate.dlMBR	Defines the UL/DL Maximum Bit Rate allowed in 1kbps units.
createQer.qfi	Identifies the QFI field in GTPU header

6.23.4.2.1 SMF Policy Example

The following JSON example is provided under the following license:

```
Copyright (c) 2025 NVIDIA CORPORATION AND AFFILIATES.  All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted
provided that the following conditions are met:
    * Redistributions of source code must retain the above copyright notice, this list of
      conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright notice, this list of
      conditions and the following disclaimer in the documentation and/or other materials
      provided with the distribution.
    * Neither the name of the NVIDIA CORPORATION nor the names of its contributors may be used
      to endorse or promote products derived from this software without specific prior written
      permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL NVIDIA CORPORATION BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TOR (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

This example illustrates a typical SMF policy JSON format and includes various policy elements such as FAR, URR and QER:

SMF Policy

```
{
    "createPdr": [
        {
            "pdrId": 1,
            "pdi": {
                "sourceInterface": {
                    "type": "0"
                },
                "localFT": {
                    "teid_start": 1073741824,
                    "teid_end": 1073807359,
                    "ip": {
                        "v4": "192.168.1.1"
                    }
                },
                "userEquipment": {
                    "ip": {
                        "v4": "172.0.0.0/16"
                    }
                },
                "sdf": [
                    {
                        "description": "permit out ip from any to assigned"
                    }
                ]
            },
            "farId": 1,
            "urrIds": [
                1
            ],
            "qerIds": [
                1
            ]
        },
        {
            "pdrId": 2,
            "pdi": {
                "sourceInterface": {
                    "type": "2"
                },
```

```
                "userEquipment": {
                    "ip": {
                        "v4": "172.0.0.0/16"
                    }
                },
                "sdf": [
                    {
                        "description": "permit out ip from any to assigned"
                    }
                ]
            },
            "farId": 2,
            "urrIds": [
                2
            ],
            "qerIds": [
                2
            ]
        },
        {
            "pdrId": 3,
            "pdi": {
                "sourceInterface": {
                    "type": "0"
                },
                "localFT": {
                    "teid_start": 1073741824,
                    "teid_end": 1073807359,
                    "ip": {
                        "v4": "192.168.1.1"
                    }
                },
                "qfi": 1,
                "userEquipment": {
                    "ip": {
                        "v4": "172.0.0.0/16"
                    }
                },
                "sdf": [
                    {
                        "description": "permit out ip from any to assigned"
                    }
                ]
            },
            "farId": 1,
            "urrIds": [
                1
            ],
            "qerIds": [
                1
            ]
        }
    ],

    "createFar": [
        {
            "farId": 1
        },
        {
            "farId": 2,
            "fp": {
                "outerHeader": {
                    "ip": {
                        "v4": "192.168.1.1"
                    },
                    "teid" : 1
                }
            }
        }
    ],

    "createUrr": [
        {
            "urrId": 1,
            "volumeQuota": {
                "totalVolume": 4000000000
            }
        },
        {
            "urrId": 2,
            "volumeQuota": {
                "totalVolume": 200000
            }
        }
    ],
    "createQer": [
        {
            "qerId": 1,
            "maxBitRate": {
                "ulMBR": "2000000000",
                "dlMBR": "2000000000"
            },
            "qfi":"20"
        },
        {
            "qerId": 2,
            "maxBitRate": {
                "ulMBR": "2000000000",
                "dlMBR": "2000000000"
            },
            "qfi":"20"
        }
    ]
}
```

6.23.4.3 Flow Processing

In the Accelerated UPF-RAP, the process from the moment the flow initiates by its first packet to the point it gets accelerated involves several key steps. Here's a detailed description of this process:

1. Initial Packet Reception: When a new packet of a flow arrives it will be forwarded to the Accelerated UPF-RAP, then the application identifies the flow and applies initial packet processing rules.
   The Accelerated UPF-RAP uses the PDR to recognize the flow and determine the necessary actions, such as forwarding, buffering, or dropping the packet.
2. Hardware Acceleration: With the flow identified and classified the UPF programs BlueField-3 to process this flow data plane. This hardware acceleration significantly reduces the CPU load and enhances the overall efficiency of the UPF.
   Tasks like GTP-U encapsulation/decapsulation, flow tagging, QoS enforcement, and packet forwarding are handled by the NIC, providing deterministic performance and scalability.
3. Accelerated Flows: Subsequent packets of that flow will be fully processed by Bluefiled-3 HW and forward to their destination without going through the host/UPF-RAP.
4. Flow Removal: If a flow remains inactive for the duration of its configured inactive time (e.g. 15 seconds). The Accelerated UPF-RAP then removes the flow from its tables and freeing up the associated resources.

6.23.4.4 Test Environment and Setup

The Accelerated UPF-RAP was tested in the following environment and setup.

Item	Description
Test Description	NVIDIA BlueField-3 DPU. Throughput and packet rate test at maximum 0.01% packet loss.
Server	Intel(R) Xeon(R) Platinum 8362 CPU @ 2.80GHz
DPU	Nvidia BlueField-3 P-Series DPU 200GbE/NDR200 VPI dual-port QSFP112; PCIe Gen5.0 x16 FHHL with x16 PCIe extension option; Crypto Enabled; SB Enabled 32GB on-board DDR; integrated BMC. Part Number: 900-9D3B6-00CV-AAA_DK_Ax
Operating System	Ubuntu 22.04
DPU firmware version	32.43.1014
DPDK version	22.11
DOCA version	2.10

Item	Description
Test Configuration	1 NIC/DPU, 2 ports. The ports receive a stream of IP flows from the traffic generator. 1 queue assigned per each core with a total of 10 cores. Port 0 receives RAN traffic of the form: `Ether() / IP() / UDP() / GTP_U_Header() / IP() / UDP()` Port 1 receives WAN traffic of the form: `Ether() / IP() / UDP()`
DPDK Settings	Compile DPDK using: meson ; ninja -C
Command Line	./doca_upf_accel -l 0-9 -- -a <PORT_0>,dv_flow_en=2 -a <PORT_1>,dv_flow_en=2 -f <SMF_POLICY_JSON>

6.23.5 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Flow

For additional information about the used DOCA libraries, please refer to the respective programming guides.

6.23.6 Dependencies

- NVIDIA BlueField-3 DPU
- Ubuntu 22.04 host (x86)

6.23.7 Compiling the Application

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for the application is located in the following directory:

```
/opt/mellanox/doca/applications/upf_accel/
```

The application relies on the **`json-c`** open source, requiring the following to be installed:

- Ubuntu/Debian:

```
$ sudo apt install libjson-c-dev
```

- CentOS/RHEL:

```
$ sudo yum install json-c-devel
```

6.23.7.1 Compiling All Applications

All DOCA applications are defined under a single Meson project. By default, the build process compiles all of them.

To build all applications:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

6.23.7.2 Compiling Only the Current Application

To build the Accelerated UPF-RAP:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_upf_accel=true
ninja -C /tmp/build
```

doca_upf_accel is created under **/tmp/build/applications/upf_accel/** .

Alternatively, one can set the desired flags in the **meson_options.txt** file instead of providing them in the compilation command line:

1. Edit the following flags in **/opt/mellanox/doca/applications/meson_options.txt** :
   a. Set **enable_all_applications** to **false**
   b. Set **enable_upf_accel** to **true**
2. Execute:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_upf_accel is created under **/tmp/build/applications/upf_accel/** .

6.23.8 Running the Application

6.23.8.1 Runtime Prerequisites

- The Accelerated UPF-RAP is based on DOCA Flow. Therefore, the user is required to allocate huge pages.

```
$ echo '1024' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
$ sudo mkdir /mnt/huge
$ sudo mount -t hugetlbfs -o pagesize=2M nodev /mnt/huge
```

- The Flex Parser Profile is a setting that enables flexible protocol parsing on NVIDIA NICs/DPUs. To enable GTP protocol support, set the Flex Parser Profile to 3 using mlxconfig. This configuration is mandatory and must be done manually in the system:

```
$ sudo mlxconfig -d <pcie_address> s FLEX_PARSER_PROFILE_ENABLE=3
```

Changing this configuration using mlxconfig requires a system (cold) reboot for the changes to take effect. For more information, refer to mlxconfig documentation.

6.23.8.2 Application Execution

The Accelerated UPF-RAP is provided in source form, hence a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_upf_accel [DPDK Flags] -- [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -a, --device                      device
  -r, --rep                         device representor
  -f, --smf-config-file-path        SMF JSON definitions file path
  -x, --vxlan-config-file-path      VXLAN JSON definitions file path
  -t, --aging-time-sec              Aging period in seconds
  -o, --dpi-packet-threshold        Number of packets processed in software before accelerating to
hardware, enabling DPI on initial packets
  -p, --fixed-port                  UL port number in fixed-port mode
```

For additional information, please refer to the Command Line Flags section below.

The above usage printout can be printed to the command line using the **-h** (or **--help**) options:

```
/tmp/build/upf_accel/doca_upf_accel -- -h
```

2. CLI example for running the application:

```
/tmp/build/upf_accel/doca_upf_accel -l 0-10 -- -a pci/03:00.0,dv_flow_en=2 -a pci/03:00.1,dv_flow_en=2 -f
<SMF_POLICY_JSON>
```

For additional information regarding the command line syntax of the device and device representor identifiers, please refer to the matching section in the DOCA Arg Parser Guide.

6.23.8.3 Command Line Flags

6.23.8.3.1 General Flags

Short Flag	Long Flag	Description
`-h`	`--help`	Prints a help synopsis and exits
`-v`	`--version`	Prints program version information and exits
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.23.8.3.2 Program Flags

Short Flag	Long Flag	Description
`a`	`device`	Device identifier
`r`	`rep`	Device representor identifiers for the ports indices
`f`	`smf-config-file-path`	Path to the SMF policy JSON file, see SMF Policy Example ⚠ This is a mandatory flag.
`t`	`aging-time-sec`	Aging period in seconds, Optional; default is 15.
`o`	`dpi-packet-threshold`	Specifies the number of packets processed in software before accelerating to hardware, enabling DPI on initial packets. Optional; default is 2.

6.23.8.4 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.23.9 References

- **/opt/mellanox/doca/applications/upf_accel/**

6.24 DOCA UROM RDMO Application Guide

This guide provides a DOCA Remote Direct Memory Operation implementation on top of NVIDIA® BlueField® DPU using Unified Communication X (UCX).

6.24.1 Introduction

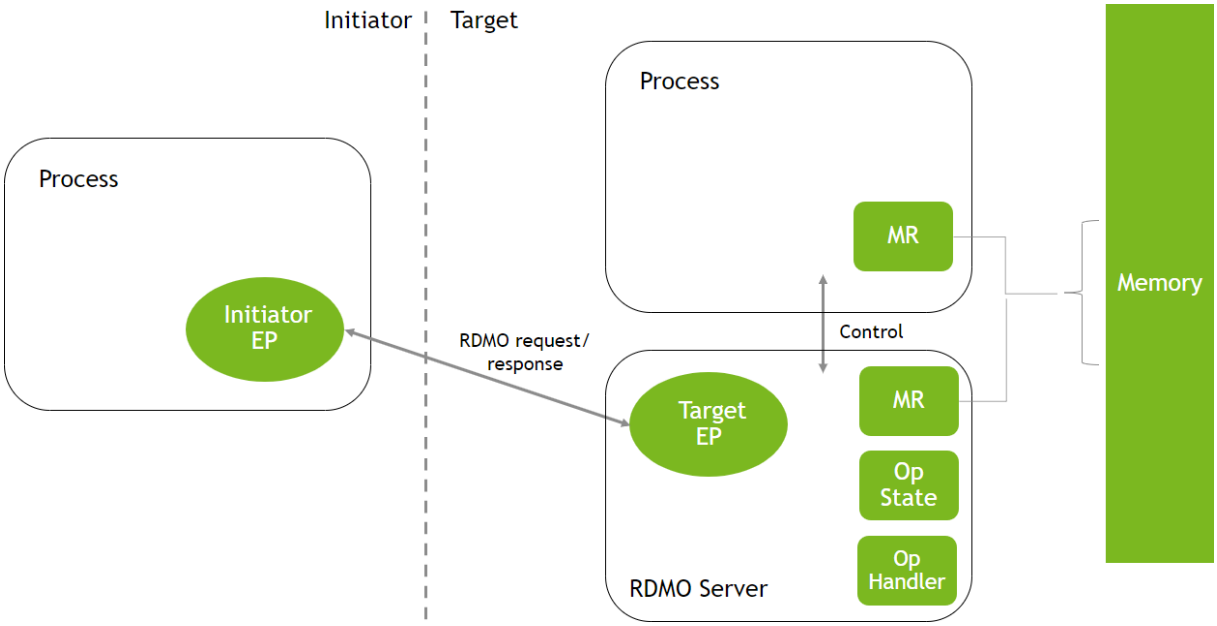
A remote direct memory operation (RDMO) is conceptionally an active message which is executed outside the context of the target process.

An RDMO involves the following entities:

- Target – establishes a connection to the server to use as the control path. The target interacts with the server to define target endpoints and memory regions. The target exchanges endpoint and memory region information with an initiator to facilitate its connection.
- Initiator – establishes a connection to the server to use as the data path. An RDMO is initiated by sending an RDMO command with an optional payload to the server. The server parses the commands and runs an associated RDMO handler. An RDMO handler interacts with the target process by performing one-sided memory accesses to target-defined memory regions.
- Server – responsible for executing RDMOs asynchronously from the target process. The server implements an RDMO handler for each supported operation. RDMO handlers may maintain a state within the server for optimization.

The DOCA UROM RDMO application includes the above three entities, split into the following parts:

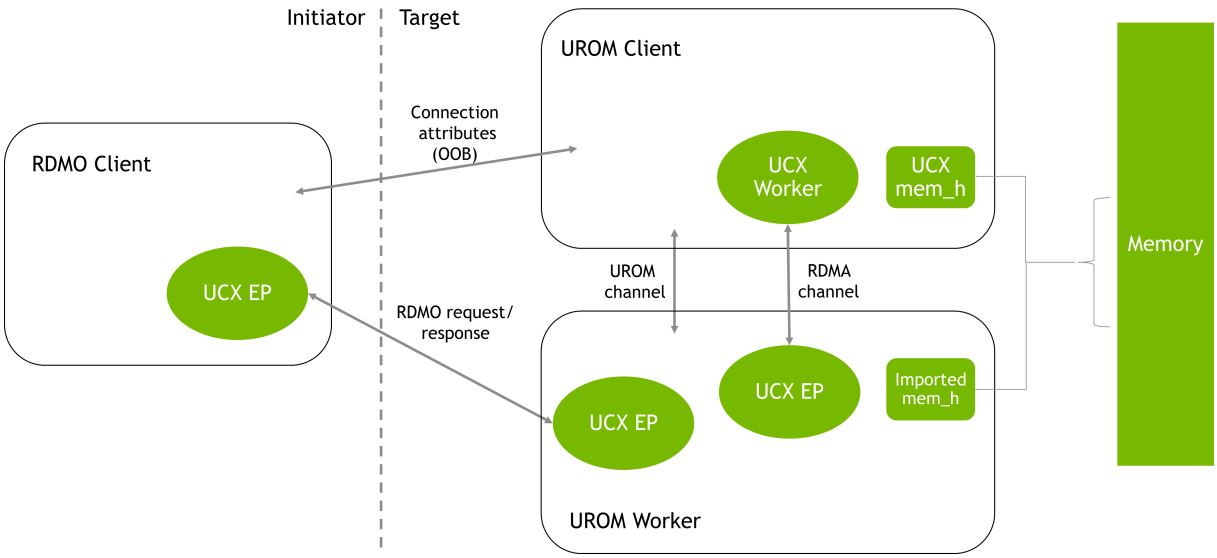
- BlueField side – the implementation of RDMO plugin component to be loaded by the DOCA UROM worker (which is the RDMO server)
- Host side – host application that runs using two modes: target and initiator



RDMOs are designed to take advantage of extra computing resources on a platform. While application processes run on the primary compute resources, an RDMO server can run on idle resources on the same host or be offloaded to run on a separate device (i.e., BlueField).

6.24.2 System Design

The application demonstrates the implementation of RDMO operations as a DOCA UROM worker plugin component. A target process would use the DOCA UROM API to create a worker with RDMO capabilities. An initiator process establishes an RDMO connection to the UROM worker. The plugin uses UCX as its transport.



6.24.2.1 Bootstrap Procedure

To connect the RDMO initiator and target, on the target side, UROM is used to retrieve an address for each created RDMO worker. This address would need to be delivered to the RDMO initiator side for connection establishment. The initiator address is obtained from the UCX worker created explicitly by the RDMO application. Both addresses are exchanged over the out-of-band (OOB) network and used to establish the connection:

- On the RDMO initiator side, a UCX endpoint is created using UCX API
- On the RDMO target side, the initiator's address is communicated to the RDMO worker using the UROM command channel

6.24.2.2 Memory Management

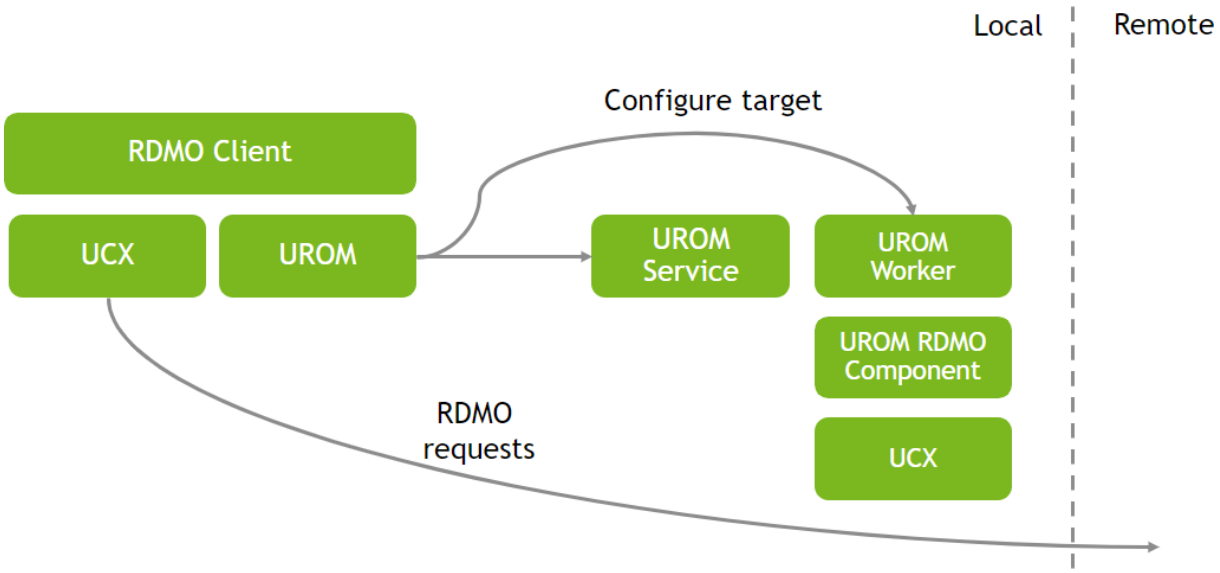
UROM returns an identifier (ID) for each memory region imported to the RDMO plugin component. This ID is used to refer to a target memory region in RDMO requests. It must be exchanged with the initiator process OOB.

6.24.2.3 RDMO UROM Worker Operation

Communication between the RDMO initiator and worker is implemented on top of UCX active messages. The worker’s active message handler is the entry point that identifies the type of the RDMO operation based on the RDMO request header. The request is then forwarded to the corresponding RDMO operation handler which determines the operation parameters by inspecting the operation-specific sub-header in the request.

UCX active messages support eager and rendezvous protocols. When using a rendezvous protocol, the worker can choose whether to pull data to the server or move it directly to a target memory using a UCX-imported memory handle.

An RDMO operation handler may perform any combination of computation, initiator and target memory accesses, server state updates, or responses.



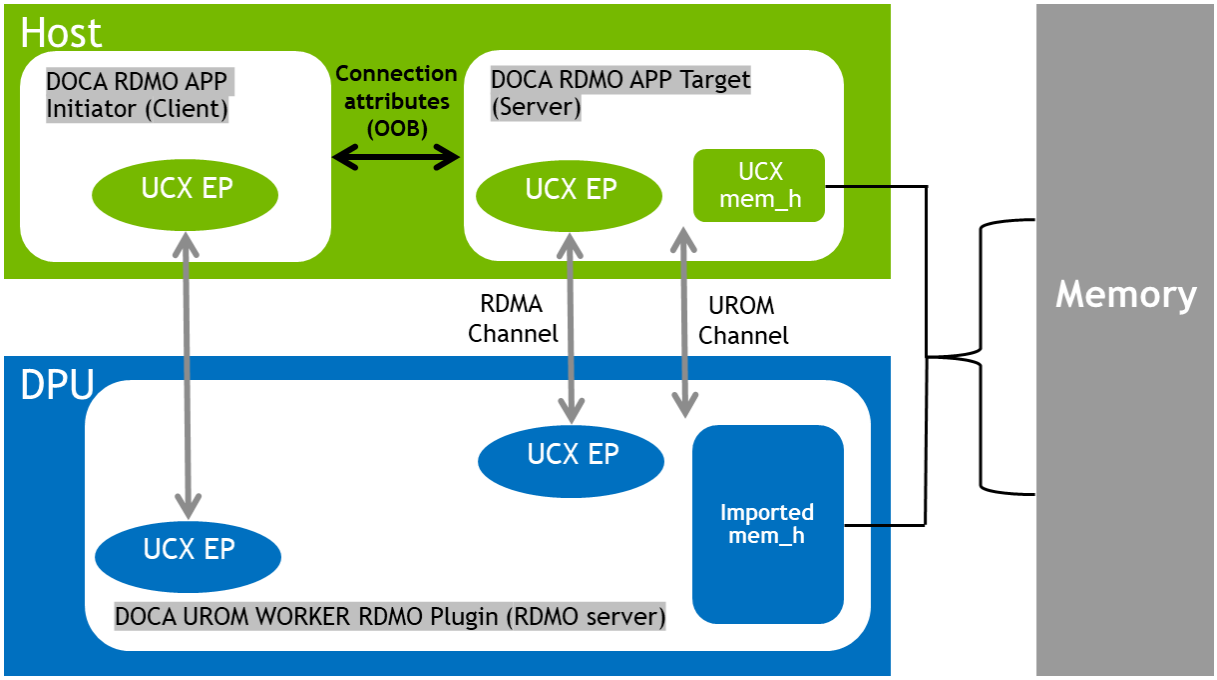
The RDMO client uses UROM to instantiate an RDMO worker and to configure target endpoints and memory regions. The client uses UCX directly to connect endpoints to the RDMO server. The client uses UCX to send formatted RDMO messages.

6.24.3 Architecture

DOCA's UROM RDMO application implementation uses UCX to support data exchange between endpoints. It utilizes UCX's sockaddr-based connection establishment and the UCX active messages (AM) API for communications, and UCX is responsible for all RDMO communications (control and data path).

The RDMO server application initiates a DOCA UROM worker RDMO component via the DOCA UROM service and shares the UROM worker UCX EP with the DOCA UROM RDMO client application. The RDMO server application imports memory regions into the UROM worker to facilitate RDMA operations from the BlueField on host memory.

The RDMO client application performs RDMO operations via the DOCA UROM worker. Upon receiving the UCX EP address from the server, the client application initially establishes a connection with the worker. It then proceeds to request the worker to execute the operation without the server application's awareness.



6.24.3.1 UROM RDMO Worker Component

The UROM RDMO worker plugin component defines a small set of commands to enable the target to:

- Establish a UCX communication channel between the client and the worker
- Create a UCX endpoint capable of receiving RDMO request
- Import memory regions that can be used as a source or target for RDMA initiated by the worker

The set of commands are:

```
enum urom_worker_rdmo_cmd_type {
    UROM_WORKER_CMD_RDMO_CLIENT_INIT,
    UROM_WORKER_CMD_RDMO_RQ_CREATE,
    UROM_WORKER_CMD_RDMO_RQ_DESTROY,
    UROM_WORKER_CMD_RDMO_MR_REG,
    UROM_WORKER_CMD_RDMO_MR_DEREG,
};
```

The associated notification types are:

```
enum urom_worker_rdmo_notify_type {
    UROM_WORKER_NOTIFY_RDMO_CLIENT_INIT,
    UROM_WORKER_NOTIFY_RDMO_RQ_CREATE,
    UROM_WORKER_NOTIFY_RDMO_RQ_DESTROY,
    UROM_WORKER_NOTIFY_RDMO_MR_REG,
    UROM_WORKER_NOTIFY_RDMO_MR_DEREG,
};
```

6.24.3.1.1 Init

The Client Init command initializes the client to receive RDMOs. This includes establishing a connection between worker and host to allow the RDMO worker to access client memory.

The command is of type **UROM_WORKER_CMD_RDMO_CLIENT_INIT**. Command format:

```
struct urom_worker_rdmo_cmd_client_init {
    uint64_t id;
    void *addr;
    uint64_t addr_len;
};
```

- **id** – client ID used to identify the target process in RDMO commands
- **addr** – pointer to the client's UCP worker address to use for a worker-to-host connection
- **addr_len** – length of the address

This command returns a notification of type **UROM_WORKER_NOTIFY_RDMO_CLIENT_INIT**. Notification format:

```
struct urom_worker_rdmo_notify_client_init {
    void *addr;
    uint64_t addr_len;
```

- **addr** – pointer to the component's UCP worker address to use for initiator-to-server connections
- **addr_len** – length of the address

6.24.3.1.2 RQ Create

This Receive Queue (RQ) Create command creates and connects a new endpoint on the server. The endpoint may be targeted by formatted RDMO messages.

This command is of type **UROM_WORKER_CMD_RDMO_RQ_CREATE**. Command format:

```
struct urom_worker_rdmo_cmd_rq_create {
    void *addr;
    uint64_t addr_len;
};
```

- **addr** – the UCP worker address to use to connect the new endpoint
- **addr_len** – the length of address

The command returns a notification of type **UROM_WORKER_NOTIFY_RDMO_RQ_CREATE**. Notification format:

```
struct urom_worker_rdmo_notify_rq_create {
    uint64_t rq_id;
};
```

- **rq_id** – the RQ ID to use to destroy the RQ

6.24.3.1.3 RQ Destroy

The RQ Destroy command destroys an RQ.

The RQ Destroy command is of type **UROM_WORKER_CMD_RDMO_RQ_DESTROY**. Command format:

```
struct urom_worker_rdmo_cmd_rq_destroy {
    uint64_t rq_id;
};
```

- **rq_id** – the ID of a previously created RQ

The RQ destroy command returns a notification of type **UROM_WORKER_NOTIFY_RDMO_RQ_DESTROY**. Notification format:

```
struct urom_worker_rdmo_notify_rq_destroy {
    uint64_t rq_id;
};
```

- **rq_id** – the destroyed receive queue id

6.24.3.1.4 MR Register

The Memory Region (MR) Register command registers a UCP memory handle with the RDMO component. An MR must be registered with the RDMO component before use in RDMOs.

The command is of type **UROM_WORKER_CMD_RDMO_MR_REG**. Command format:

```
struct urom_worker_rdmo_cmd_mr_reg {
    uint64_t va;
    uint64_t len;
    void *packed_rkey;
    uint64_t packed_rkey_len;
    void *packed_memh;
    uint64_t packed_memh_len;
};
```

- **va** – the virtual address of the MR
- **len** – the length of the MR
- **packed_rkey** – pointer to the UCP packed R-key for the MR
- **packed_rkey_len** – the length of **packed_rkey**
- **packed_mem_h** – pointer to the UCP-packed memory handle for the MR. The memory handle must be packed with flag **UCP_MEMH_PACK_FLAG_EXPORT**.
- **packed_memh_len** – the length of **packed_memh**

The command returns a notification of type **UROM_WORKER_NOTIFY_RDMO_MR_REG**. Notification format:

```
struct urom_worker_rdmo_notify_mr_reg {
    uint64_t rkey;
};
```

- **rkey** – the ID used in RDMOs to refer to the MR

6.24.3.1.5 MR Deregister

The MR deregister command deregisters an MR from the RDMO component.

The command is of type **UROM_WORKER_CMD_RDMO_MR_DEREG**. Command format:

```
struct urom_worker_rdmo_cmd_mr_dereg {
    uint64_t rkey;
};
```

- **rkey** – the ID of a previously registered MR

The command returns a notification of type **UROM_WORKER_NOTIFY_RDMO_MR_DEREG**. Notification format:

```
struct urom_worker_rdmo_notify_mr_dereg {
    uint64_t rkey;
};
```

- **rkey** – the deregistered memory region remote key

6.24.3.2 Command Format

An RDMO is initiated by sending an RDMO request via UCP active message to a UROM RDMO worker server.

The RDMO request format is:

The RDMO header identifies the operation type and flags, modifying how the RDMO is processed. The operation (op) header includes arguments specific to the operation type. Optionally, the operation type may include an arbitrary-sized payload.

RDMO header format:

```
struct urom_rdmo_hdr {
    uint32_t id;
    uint32_t op_id;
    uint32_t flags;
};
```

- **id** – the client ID
- **op_id** – the RDMO operation type ID
- **flags** – flags modifying how the RDMO is processed by the server

Valid flag values:

```
enum urom_rdmo_req_flags {
    UROM_RDMO_REQ_FLAG_FENCE,
};
```

- **UROM_RDMO_REQ_FLAG_FENCE** – Complete all outstanding RDMO requests on the connection before executing this request. This flag is required to implement a flush operation that guarantees remote completion.

Optionally, an operation may return a response to the initiator.

Response header format:

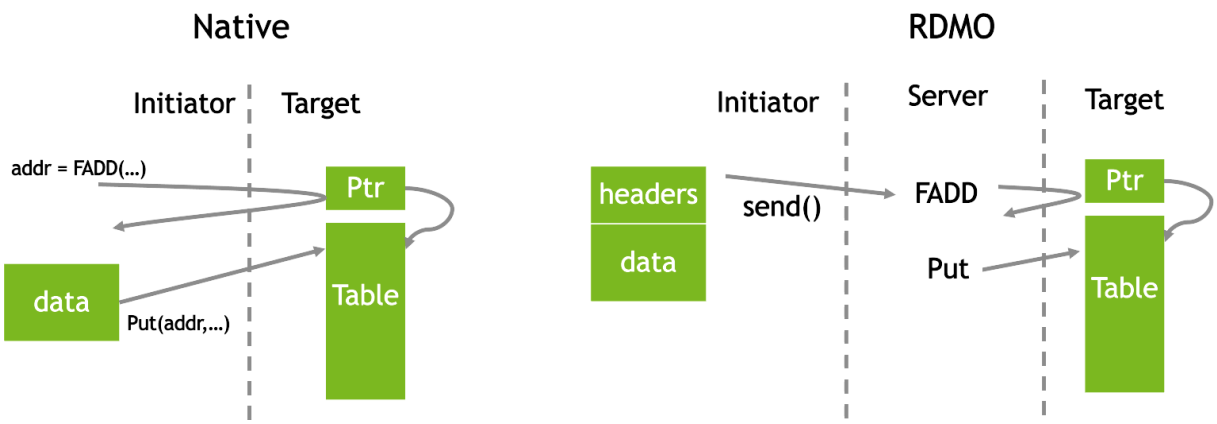
```
struct urom_rdmo_rsp_hdr {
    uint16_t op_id;
};
```

- **op_id** – the RDMO response type ID

6.24.3.2.1 Append

RDMO Append atomically appends data to a queue in remote memory. This can be achieved in a one-sided programming model with a Fetching-Add operation to the location of a pointer in remote memory, followed by a Put to the fetched address. RDMO Append allows these dependent operations to be offloaded to the target.

The following diagram provides a comparison of native and RDMO approaches to the Append operation:



Combining two dependent operations into a single RDMO allows the non-blocking implementation of Append, as the initiator does not need to wait between the Fetching Atomic and the data write operations. Using RDMO, the initiator can create a pipeline of operations and achieve a higher message rate.

The rate at which the RDMO server can perform operations on the target memory is expected to be a bottleneck. To improve the rate, the following optimizations can be looked at:

- The result of the Fetch-and-ADD (FADD) after the initial Append is performed can be cached in the server. Subsequent Appends can re-use the cached value, eliminating the atomic FADD operation. The modified pointer value is required to be synchronized during the flush command.

- For small Append sizes, the Append data can be cached in the RDMO server and coalesced into a single Put. As a result, the server requires, on average, a single Put access to target memory to execute several RDMOs.
- To avoid extra memory usage and lost bandwidth for large Append operations, the RDMO server may initiate direct transfers from the initiator to the target memory bypassing the acceleration device memory.

The Append operation uses an operation of type **UROM_RDMO_OP_APPEND**. Append header format:

```
struct urom_rdmo_append_hdr {
    uint64_t ptr_addr;
    uint16_t ptr_rkey;
    uint16_t data_rkey;
};
```

- **ptr_addr** – the address of the queue pointer in target memory
- **ptr_rkey** – the R-key used to access **ptr_addr**
- **data_rkey** – the R-key used to access the queue data

The RDMO payload is the local data buffer.

6.24.3.2.2 Flush

RDMO Flush is used to implement synchronization between the initiator and server. On execution, Flush sends a response message back to the initiator. Flush can be used to guarantee remote completion of a previously issued RDMO.

To achieve this, the initiator sends an in-order Flush command including the RDMO flag **UROM_RDMO_REQ_FLAG_FENCE**. This flag causes the server to complete all previously received RDMOs before executing the Flush. To complete previous operations, the server must write any cached data and make it visible in the target memory. Once complete, the server executes the Flush. Flush sends a response to the initiator. When the initiator receives the flush message, the result of all previously sent RDMOs is guaranteed to be visible in the target memory.

The Flush operation uses operation type **UROM_RDMO_OP_FLUSH**. Flush header format:

```
struct urom_rdmo_flush_hdr {
    uint64_t flush_id;
};
```

- **flush_id** – local ID used to track completion

Flush returns a response with the following header format:

```
struct urom_rdmo_flush_rsp_hdr {
    uint64_t    flush_id;
};
```

- **flush_id** – the ID of the completed Flush

Flush requests and responses do not include a payload.

6.24.3.2.3 Scatter

RDMO Scatter is used to support aggregating non-contiguous memory Puts. An RDMO may be defined to map non-contiguous virtual addresses into a single memory region using a network interface at the target platform, and then return a memory key for this region. The initiator may then perform Puts to this memory region, which are scattered by target hardware. Alternatively, an RDMO may be defined to post an IOV Receive. The initiator could then post a matching Send to scatter data at the target.

The Scatter operation uses operation type **UROM_RDMO_OP_SCATTER**. Scatter header format:

```
struct urom_rdmo_scatter_hdr {
    uint64_t count; /* Number of IOVs in the payload */
};
```

- **count** – Number of IOVs in the RDMO payload

IOVs are packed into the Scatter request payload, descriptor followed by data:

```
struct urom_rdmo_scatter_iov {
    uint64_t addr; /* Scattered data address */
    uint64_t rkey; /* Data remote key */
    uint16_t len;  /* Data length */
};
```

- **addr** – scattered data address
- **rkey** – data remote key
- **len** – data length

6.24.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA UROM
- UCX Framework DOCA Driver

Refer to their respective programming guide for more information.

6.24.5 Compiling the Application

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for the application is located in the following directory:

```
/opt/mellanox/doca/applications/urom_rdmo/
```

6.24.5.1 Compiling All Applications

All DOCA applications are defined under a single Meson project. By default, the build process compiles all of them.

To build all applications:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

On the host, **doca_urom_rdmo** is created under **/tmp/build/urom_rdmo/host/** . On the BlueField side, the RDMO worker plugin **worker_rdmo.so** is created under **/tmp/build/urom_rdmo/dpu/** .

6.24.5.2 Compiling Only the Current Application

To directly build only the UROM RDMO application (host) or plugin (DPU):

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_urom_rdmo=true
ninja -C /tmp/build
```

On the host, **doca_urom_rdmo** is created under **/tmp/build/urom_rdmo/host/** . On the BlueField side, the RDMO worker plugin **worker_rdmo.so** is created under **/tmp/build/urom_rdmo/dpu/** .

Alternatively, one can set the desired flags in the **meson_options.txt** file instead of providing them in the compilation command line:

1. Edit the following flags in **/opt/mellanox/doca/applications/meson_options.txt** :
   - Set **enable_all_applications** to **false**
   - Set **enable_urom_rdmo** to **true**
2. Run the following compilation commands:

   ```
   cd /opt/mellanox/doca/applications/
   meson /tmp/build
   ninja -C /tmp/build
   ```

   On the host, **doca_urom_rdmo** is created under **/tmp/build/urom_rdmo/host/** . On the BlueField side, the RDMO worker plugin **worker_rdmo.so** is created under **/tmp/build/urom_rdmo/dpu/** .

6.24.6 Running the Application

6.24.6.1 Host Application Execution

The UROM RDMO application is provided in source form; therefore, a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_urom_rdmo [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -d, --device <IB device name>     IB device name.
  -s, --server-name <server name>   server name.
  -m, --mode {server, client}       Set mode type {server, client}
```

> This usage printout can be printed to the command line using the **-h** (or **--help**) options:
>
> ```
> ./doca_urom_rdmo -h
> ```

> For additional information, refer to section "Command Line Flags".

2. CLI example for running the application with server mode:

```
./doca_urom_rdmo -d mlx5_0 -m server
```

3. CLI example for running the application with client mode:

```
./doca_urom_rdmo -m clinet -s <server_host_name>
```

6.24.6.2 RDMO DPU Plugin Component

The UROM RDMO plugin component is provided in source form, hence a compilation is required before the application can be executed in order when spawning UROM worker could load the plugin in runtime and it is compiled as **.so** file.

The plugin exposes the following symbols:

- Get DOCA worker plugin interface for RDMO plugin:

```
doca_error_t urom_plugin_get_iface(struct urom_plugin_iface *iface);
```

- Get the RDMO plugin version which will be used to verify that the host and DPU plugin versions are compatible:

```
doca_error_t urom_plugin_get_version(uint64_t *version);
```

6.24.6.3 Command Line Flags

6.24.6.3.1 General Flags

Short Flag	Long Flag	Description
`-h`	`--help`	Prints a help synopsis and exits
`-v`	`--version`	Prints program version information and exits
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.24.6.3.2 Program Flags

Short Flag	Long Flag	Description
`d`	`device`	DOCA UROM IB device name
`s`	`server-name`	RDMO server name
`m`	`mode`	RDMO application mode [server, client]

6.24.6.4 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.24.7 Application Code Flow

1. Parse application argument.
    a. Initialize arg parser resources and register DOCA general parameters.

```
doca_argp_init();
```

    b. Register UROM RDMO application parameters.

```
register_urom_rdmo_params();
```

    c. Parse the arguments.

```
doca_argp_start();
```

2. Run main logic:
    - If the application mode is server:
        i. Create UROM objects and spawn UROM worker on the BlueField.
        ii. Initialize UCP with features: **UCP_FEATURE_AM**, **UCP_FEATURE_EXPORTED_MEMH**.
        iii. Create a UCP worker and query the worker address
        iv. Initialize the RDMO worker client with the command **UROM_WORKER_CMD_RDMO_CLIENT_INIT**.
        v. Send UROM RDMO worker address to the initiator via OOB channel and receive the intiator's UCP worker address
        vi. Create a UCP memory handle and register it with the RDMO server using the command **UROM_WORKER_CMD_RDMO_MR_REG**. Receive an R-key in return.
        vii. Send the RDMO key to the initiator
        viii. Create an RDMO RQ by passing the initiator's UCP worker address to the UROM command **UROM_WORKER_CMD_RDMO_RQ_CREATE**.
        ix. Wait till the RDMO append operation is done and next validate the memory data.
        x. Wait till the RDMO scatter operation is done and next validate the memory data.
        xi. Destroy the UCP resources.
        xii. Destroy UROM RDMO worker and UROM objects.
    - If the application mode is client:
        i. Create UCP worker using UCX API directly.
        ii. Receive the UROM RDMO worker address via OOB channel and send the initiator's UCP worker address.
        iii. Create a UCP endpoint using the RDMO worker address.
        iv. Install an Active Message handler on the endpoint to receive RDMO responses.
        v. Send an RDMO requests via UCP Active Message protocol with the header pointing to the serialized RDMO and Op headers, and data pointing to the payload. The request parameter flag: **UCP_AM_SEND_FLAG_REPLY** will be set to allow the RDMO server to identify the sender.
        vi. Once the RDMO operations are done, Destroy UCP resources.

3. Arg parser destroy.

```
doca_argp_destroy();
```

6.24.8 References

- **/opt/mellanox/doca/applications/urom_rdmo/**

6.25 DOCA YARA Inspection Application Guide

This guide provides YARA inspection implementation on top of NVIDIA® BlueField® DPU.

6.25.1 Introduction

YARA inspection monitors all processes in the host system for specific YARA rules using the DOCA App Shield library.

This security capability helps identify malware detection patterns in host processes from an independent and trusted DPU. This is an innovative Intrusion Detection System (IDS) as it is designed to run independently on the DPU's Arm cores without hindering the host.

This DOCA App Shield based application provides the capability to read, analyze, and authenticate the host (bare metal/VM) memory directly from the DPU.

Using the library, this application scans host processes and looks for pre-defined YARA rules. After every scan iteration, the application indicates if any of the rules matched. Once there is a match, the application reports which rules were detected in which process. The reports are both printed to the console and exported to the DOCA Telemetry Service (DTS) using inter-process communication (IPC).

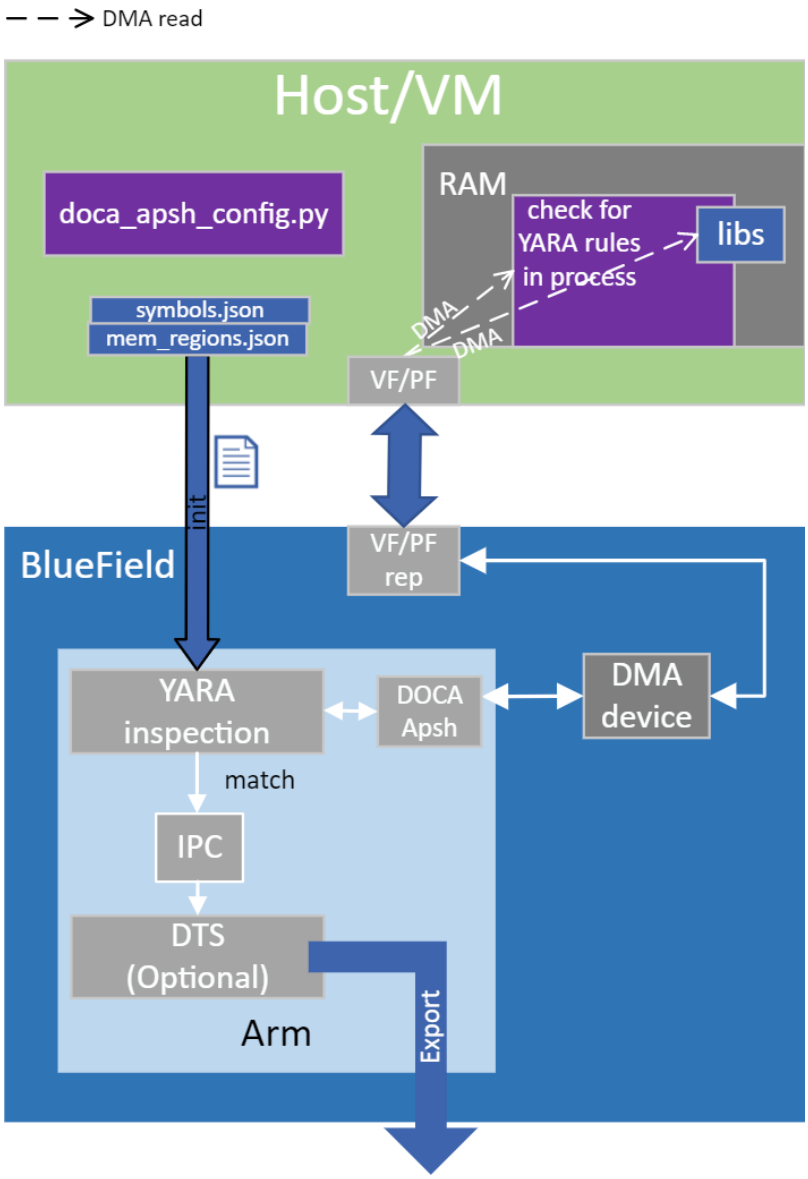
This guide describes how to build YARA inspection using the DOCA App Shield library which leverages DPU abilities such as hardware-based DMA, integrity, and more.

> ⚠ As the DOCA App Shield library only supports the YARA API for Windows hosts, this application can only be used to inspect Windows hosts.

6.25.2 System Design

The host's involvement is limited to generating the required ZIP and JSON files to pass to the DPU. This is done before the app is triggered, when the host is still in a "safe" state.

Generating the needed files can be done by running DOCA App Shield's **doca_apsh_config.py** tool on the host. See DOCA App Shield for more info.

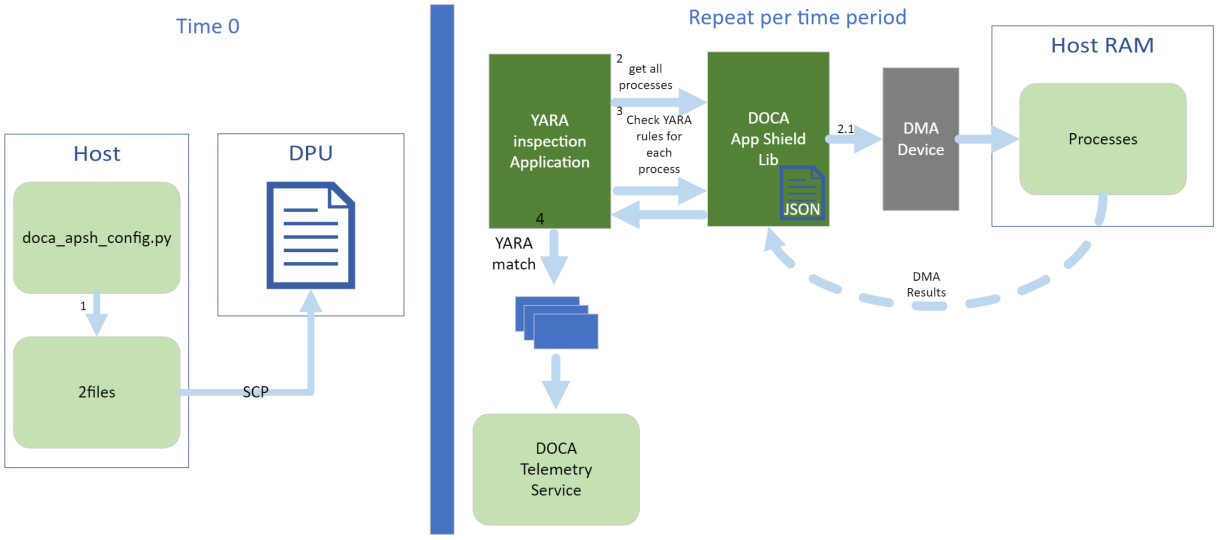


6.25.3 Application Architecture

The user creates the ZIP and JSON files using the DOCA tool **doca_apsh_config.py** and copies them to the DPU.

The application can report YARA rules detection to the:

- File
- Terminal
- DTS



1. The files are generated by running **doca_apsh_config.py** on the host against the process at time zero.
2. The following steps recur at regular time intervals:
    a. The YARA inspection app requests a list of all apps from the DOCA App Shield library.
    b. The app loops over all processes and checks for YARA rules match using the DOCA App Shield library.
    c. If YARA rules are found (1 or more), the YARA attestation app reports results with a timestamp and details about the process and rules to:
        - Local telemetry files – a folder and files representing the data a real DTS would have received

          ⚠ These files are used for the purpose of this example only as normally this data is not exported into user-readable files.

        - DOCA log
        - DTS IPC interface (even if no DTS is active)
3. The App Shield agent exits on first YARA rule detection.

6.25.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA App Shield
- 2024-10-09_07-10-18_DOCA Telemetry

Refer to their respective programming guide for more information.

6.25.5 Limitations

- The application is only available on Ubuntu 22.04 environments

- The application only supports the inspection of Windows hosts

6.25.6 Compiling the Application

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

The installation of DOCA's reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The sources of the application can be found under the application's directory: **/opt/mellanox/doca/applications/yara_inspection/**.

6.25.6.1 Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_yara_inspection is created under **/tmp/build/yara_inspection/**.

6.25.6.2 Compiling Only the Current Application

To directly build only the YARA inspection application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_yara_inspection=true
ninja -C /tmp/build
```

doca_yara_inspection is created under **/tmp/build/yara_inspection/**.

Alternatively, one can set the desired flags in the **meson_options.txt** file instead of providing them in the compilation command line:

1. Edit the following flags in **/opt/mellanox/doca/applications/meson_options.txt**:
   - Set **enable_all_applications** to **false**
   - Set **enable_yara_inspection** to **true**
2. Run the following compilation commands:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

doca_yara_inspection is created under **/tmp/build/yara_inspection/**.

6.25.7 Running the Application

6.25.7.1 Prerequisites

1. Configure the BlueField's firmware
   a. On the BlueField system, configure the PF base address register and NVME emulation. Run:

   ```
   dpu> mlxconfig -d /dev/mst/mt41686_pciconf0 s PF_BAR2_SIZE=2 PF_BAR2_ENABLE=1
    NVME_EMULATION_ENABLE=1
   ```

   b. Perform a BlueField system reboot for the **mlxconfig** settings to take effect.
   c. This configuration can be verified using the following command:

   ```
   dpu> mlxconfig -d /dev/mst/mt41686_pciconf0 q | grep -E "NVME|BAR"
   ```

2. Download target system (host/VM) symbols.
   - For Ubuntu:

   ```
   host> sudo tee /etc/apt/sources.list.d/ddebs.list << EOF
   deb http://ddebs.ubuntu.com/ $(lsb_release -cs) main restricted universe multiverse
   deb http://ddebs.ubuntu.com/ $(lsb_release -cs)-updates main restricted universe multiverse
   deb http://ddebs.ubuntu.com/ $(lsb_release -cs)-proposed main restricted universe multiverse
   EOF
   host> sudo apt install ubuntu-dbgsym-keyring
   host> sudo apt-get update
   host> sudo apt-get install linux-image-$(uname -r)-dbgsym
   ```

   - For CentOS:

   ```
   host> yum install --enablerepo=base-debuginfo kernel-devel-$(uname -r) kernel-debuginfo-$(uname -r)
   kernel-debuginfo-common-$(uname -m)-$(uname -r)
   ```

   - No action is needed for Windows
3. Perform IOMMU passthrough. This stage is only needed on some of the cases where IOMMU is not enabled by default (e.g., when the host is using an AMD CPU).

   Skip this step if you are not sure whether you need it. Return to it only if DMA fails with a message in **dmesg** similar to the following:

   ```
   host> dmesg
   [ 3839.822897] mlx5_core 0000:81:00.0: AMD-Vi: Event logged [IO_PAGE_FAULT domain=0x0047
    address=0x2a0aff8 flags=0x0000]
   ```

   - Locate your OS's `grub` file (most likely **/boot/grub/grub.conf**, **/boot/grub2/grub.cfg**, or **/etc/default/grub**) and open it for editing. Run:

```
host> vim /etc/default/grub
```

- Search for the line defining **GRUB_CMDLINE_LINUX_DEFAULT** and add the argument **iommu=pt**. For example:

```
GRUB_CMDLINE_LINUX_DEFAULT="iommu=pt <intel/amd>_iommu=on"
```

- Run:

> ⚠ Prior to performing a power cycle, make sure to do a graceful shutdown.

  - For Ubuntu:

```
host> sudo update-grub
host> ipmitool power cycle
```

  - For CentOS:

```
host> grub2-mkconfig -o /boot/grub2/grub.cfg
host> ipmitool power cycle
```

  - For Windows targets: Turn off Hyper-V capability.

4. The DOCA App Shield library uses hugepages for DMA buffers. Therefore, the user must allocate 42 huge pages.
   a. Run:

```
dpu> nr_huge=$(cat /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages)
     nr_huge=$((42+$nr_huge))
     echo $nr_huge | sudo tee -a /sys/devices/system/node/node0/hugepages/hugepages-2048kB/
nr_hugepages
```

   b. Create the ZIP and JSON files. Run:

```
target-system> cd /opt/mellanox/doca/tools/
target-system> python3 doca_apsh_config.py <pid-of-process-to-monitor> --os <windows/linux> --path
<path to dwarf2json executable  or pdbparse-to-json.py>
target-system> cp /opt/mellanox/doca/tools/*.* <shared-folder-with-baremetal>
dpu> scp <shared-folder-with-baremetal>/* <path-to-app-shield-binary>
```

   If the target system does not have DOCA installed, the script can be copied from the BlueField.
   The required **dwaf2json** and **pdbparse-to-json.py** are not provided with DOCA.

> ⚠ If the kernel and process **.exe** have not changed, there no need to redo this step.

6.25.7.2 Application Execution

The YARA inspection application is provided in source form. Therefore, a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_yara_inspection [DOCA Flags] [Program Flags]

DOCA Flags:
```

```
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -m, --memr <path>                 System memory regions map
  -f, --vuid                        VUID of the System device
  -d, --dma                         DMA device name
  -o, --osym <path>                 System OS symbol map path
  -t, --time <seconds>              Scan time interval in seconds
```

This usage printout can be printed to the command line using the **-h** (or **--help**) options:

```
./doca_yara_inspection -h
```

For additional information, refer to section "Command Line Flags".

2. CLI example for running the application on the BlueField:

```
./doca_yara_inspection -m mem_regions.json -o symbols.json -f MT2125X03335MLNXS0D0F0VF1 -d mlx5_0 -t 3
```

All used identifiers (**-f** and **-d** flags) should match the identifier of the desired devices.

6.25.7.3 Command Line Flags

6.25.7.3.1 General Flags

Short Flag	Long Flag	Description
-h	**--help**	Prints a help synopsis and exits
-v	**--version**	Prints program version information and exits
-l	**--log-level**	Sets the numeric log level for the application: • **10** – DISABLE • **20** – CRITICAL • **30** – ERROR • **40** – WARNING • **50** – INFO • **60** – DEBUG • **70** – TRACE (requires compilation with TRACE support)
N/A	**--sdk-log-level**	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	**--log-filter**	Filters logs from specific modules (comma-separated list)
-j	**--json**	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.25.7.3.2 Program Flags

Short Flag	Long Flag	Description
m	memr	Path to the pre-generated mem_regions.json file transferred from the host

f	pcif	System PCIe function vendor unique identifier (VUID) of the VF/PF exposed to the target system. Used for DMA operations.
		To obtain this argument, run:
		`target-system> lspci -vv | grep "\[VU\] Vendor specific:"`
		Example output:
		`[VU] Vendor specific: MT2125X03335MLNXS0D0F0`
		`[VU] Vendor specific: MT2125X03335MLNXS0D0F1`
		Two VUIDs are printed for each DPU connected to the target system. The first is of the DPU on pf0 and the second is of the DPU on port pf1.
		⚠ Running this command on the DPU outputs VUIDs with an additional "EC" string in the middle. You must remove the "EC" to arrive at the correct VUID.
		The VUID of a VF allocated on PF0/1 is the VUID of the PF with an additional suffix, VF<vf-number>, where vf-number is the VF index +1.
		For example, for the output in the example above:
		• PF0 VUID = MT2125X03335MLNXS0D0F0
		• PF1 VUID = MT2125X03335MLNXS0D0F1
		• VUID of VF0 on PF0 = MT2125X03335MLNXS0D0F0VF1
		VUIDs are persistent even on reset.

d	dma	DMA device name to use
o	osym	Path to the pre-generated symbols.json file transferred from the host
t	time	Number of seconds to sleep between scans

6.25.7.4 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.25.8 Application Code Flow

1. Parse application argument.
    a. Initialize arg parser resources and register DOCA general parameters.

```
doca_argp_init();
```

b. Register application parameters.

```
register_apsh_params();
```

c. Parse the arguments.

```
doca_argp_start();
```

2. Initialize DOCA App Shield lib context.
   a. Create lib context.

```
doca_apsh_create();
```

   b. Set DMA device for lib.

```
open_doca_device_with_ibdev_name();
doca_apsh_dma_dev_set();
```

   c. Start the context

```
doca_apsh_start();
apsh_system_init();
```

3. Initialize DOCA App Shield lib system context handler.
   a. Get the representor of the remote PCIe function exposed to the system.

```
open_doca_device_rep_with_vuid();
```

   b. Create and start the system context handler.

```
doca_apsh_system_create();
doca_apsh_sys_os_symbol_map_set();
doca_apsh_sys_mem_region_set();
doca_apsh_sys_dev_set();
doca_apsh_sys_os_type_set();
doca_apsh_system_start();
```

4. Telemetry initialization.

```
telemetry_start();
```

   a. Initialize a new telemetry schema.
   b. Register YARA type event.
   c. Set up output to file (in addition to default IPC).
   d. Start the telemetry schema.
   e. Initialize and start a new DTS source with the **`gethostname()`** name as source ID.

5. Loop until YARA rule is matched.
   a. Get all processes from the host.

```
doca_apsh_processes_get();
```

   b. Check for YARA rule identification and send a DTS event if there is a match.

```
doca_apsh_yara_get();
if (yara_matches_size != 0) {
```

```
    /* event fill logic
    doca_telemetry_exporter_source_report();
DOCA_LOG_INFO();
sleep();
```

6. Telemetry destroy.

```
telemetry_destroy();
```

7. YARA inspection clean-up.

```
doca_apsh_system_destroy();
doca_apsh_destroy();
doca_dev_close();
doca_dev_rep_close();
```

8. Arg parser destroy.

```
doca_argp_destroy();
```

6.25.9 References

- **/opt/mellanox/doca/applications/yara_inspection/**

6.26 DOCA Pipeline Language Runtime Controller Gateway SHM Application Guide

This document describes the usage of the NVIDIA DOCA Pipeline Language (DPL) Runtime Controller Gateway SHM sample application.

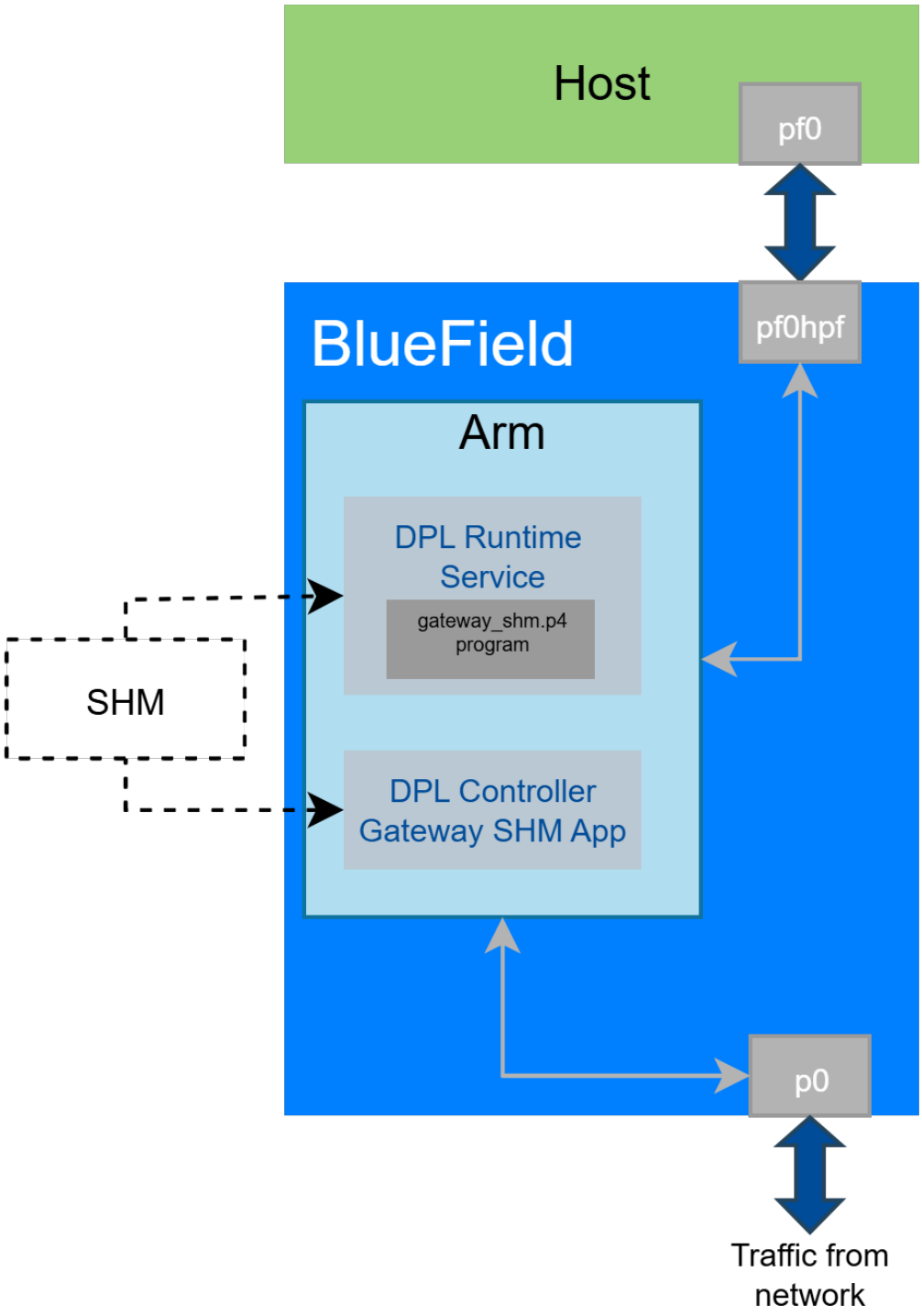
6.26.1 Introduction

This sample application leverages the capabilities of DOCA Pipeline Languages (DPL) Services combined with the DPL Runtime Controller SDK to implement a VXLAN gateway application . It provides encapsulation/decapsulation logic and interactive packet/counter management.

6.26.2 System Design

The following diagram illustrates the connections between the DPL Runtime Controller-based application and the DPL Runtime Service (DPL daemon). The DPL daemon is loaded with a DPL **gateway_shm.p4** program that implements the gateway logic.

The controller communicates with the DPL daemon through gRPC or Shared Memory (SHM) interfaces to insert, query, and delete HW steering rules. Traffic arriving from the wire or the VFs is processed according to the gateway program and passed to the relevant port based on the loaded program.



6.26.3 Architecture

The application is based on two major components that work together to define and manage the forwarding state:

6.26.3.1 DPL Program (gateway_shm.p4)

A DPL program loaded onto the DPL Runtime Service:

gateway_shm.p4

```
/*
 * SPDX-FileCopyrightText: Copyright (c) 2024-2025 NVIDIA CORPORATION & AFFILIATES. All rights reserved.
 * SPDX-License-Identifier: LicenseRef-NvidiaProprietary
 *
 * NVIDIA CORPORATION, its affiliates and licensors retain all intellectual
 * property and proprietary rights in and to this material, related
 * documentation and any modifications thereto. Any use, reproduction,
 * disclosure or distribution of this material and related documentation
 * without an express license agreement from NVIDIA CORPORATION or
 * its affiliates is strictly prohibited.
 */

#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>

/*
 * VxLAN tunnel gateway
 * This application allows the user to have a customized tunnel gateway that can stitch VxLAN
 * packets across tenant domains. End point traffic destined to local bare metal hosts can
 * be decapsulated and forwarded, while gateway traffic can be decapsulated, then
 * encapsulated back to the wire. This program can be easily extended to be a gateway across
 * different tunnel types as well.
 */

/*
* Table sizes.
*/
const bit<32> DECAP_TABLE_SIZE = 32768;
const bit<32> ENCAP_TABLE_SIZE = 32768;

/* The directionality is based on network to host
 * The user will configure the DPL port IDs in the DPL RT configuration
 */
const bit<32> WIRE_PORT = 32w0;

struct headers_t {
    NV_FIXED_HEADERS
}

parser packet_parser(packet_in packet, out headers_t headers) {
    NV_FIXED_PARSER(packet, headers)
 }

/**
* This control performs the overlay policy including L2 encap with VxLAN
*/
control overlay_encap(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) encap_counter;

    action deny() {
        encap_counter.count();
        nv_drop();
    }

    action to_port(nv_logical_port_t port) {
        encap_counter.count();
        nv_send_to_port(port);
    }

    action vxlan_v4_encap(nv_mac_addr_t underlay_src_mac, nv_mac_addr_t underlay_dst_mac,
                nv_ipv4_addr_t underlay_sip, nv_ipv4_addr_t underlay_dip, bit<24> vni, nv_logical_port_t port) {
        nv_set_vxlan_v4_underlay(headers, underlay_dst_mac, underlay_src_mac, underlay_sip, underlay_dip, vni);
        encap_counter.count();
        nv_send_to_port(port);
    }

    table encap_v4_table {
        key = {
            headers.ipv4.dst_addr : exact;
        }
        actions = {
            vxlan_v4_encap;
            to_port;
            deny;
        }
        size = ENCAP_TABLE_SIZE;
        default_action = deny;
        direct_counter = encap_counter;
        nv_high_update_rate = true;
    }

    apply {
```

```
        if (headers.ipv4.isValid()) {
            encap_v4_table.apply();
        }
    }
}


/**
* This control is for packets from wire to host (RX)
* and includes policy for L2 decap
*/
control underlay_decap(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) decap_counter;

    action deny() {
        decap_counter.count();
        nv_drop();
    }

    action decap() {
        decap_counter.count();
        nv_l2_decap(headers);
    }

    action to_port(nv_logical_port_t port) {
        nv_send_to_port(port);
    }

    action decap_to_port(nv_logical_port_t port) {
        decap();
        to_port(port);
    }

    table decap_v4_table {
        key = {
            headers.vxlan.vni : exact;
        }
        actions = {
            decap;
            to_port;
            decap_to_port;
            deny;
            NoAction;
        }
        size = DECAP_TABLE_SIZE;
        direct_counter = decap_counter;
        default_action = NoAction;
        nv_high_update_rate = true;
    }

    apply {
        if (headers.vxlan.isValid()) {
            decap_v4_table.apply();
        }
    }
}

control gateway(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    overlay_encap() over;
    underlay_decap() under;

    /* user should add entries that correspond to the wire ports
     * A hit means this is an RX packet, miss means a TX packet
    */
    table direction_table {
        key = {
            std_meta.ingress_port : exact;
        }
        actions = {
            NoAction;
        }
        default_action = NoAction;
        const entries = {
            (WIRE_PORT) : NoAction();
        }
    }

    apply {
        if (direction_table.apply().hit) {
            under.apply(headers, std_meta, user_meta, pkt_out_meta);
        }
        else {
            over.apply(headers, std_meta, user_meta, pkt_out_meta);
        }
    }
}

NvDocaPipeline(
    packet_parser(),
    gateway()
) main;
```

This P4 application implements a basic VXLAN termination and origination function for IPv4 traffic. Its primary goal is to differentiate between incoming packets from the underlay network (Rx/Decapsulation) and packets originating from a host (Tx/Encapsulation), applying the necessary L2 overlay policies in each direction.

The program logic is separated into three distinct Control Blocks: `gateway`, `underlay_decap`, and `overlay_encap`.

- `gateway`: Responsible for directing packets into the relevant control block (`underlay_decap` or `overlay_encap`) by matching on the ingress port.
- `underlay_decap`: Responsible for L2 decapsulation of packets from wire to host (Rx).
- `overlay_encap`: Responsible for the overlay policy, including L2 VXLAN encapsulation of packets from host to wire (Tx).

6.26.3.1.1 P4 Tables

Table Name	Control Block	Match Field	Actions	Purpose
`direction_table`	`gateway`	ingress_port	`NoAction` (default)	Determines the processing direction (Rx or Tx)
`decap_v4_table`	`underlay_decap`	VxLAN.vni	`decap`, `to_port`, `decap_to_port`, `deny`, `NoAction` (default)	Core policy table for decapsulation; identifies the tenant context
`encap_v4_table`	`overlay_encap`	IPv4.dst_addr	`vxlan_v4_encap`, `to_port`, `deny` (default)	Core policy table for encapsulation; determines the tunnel endpoint and VNI

We don't have a way to export this macro.

6.26.3.1.2 Direct Counters

Counter Name	Tied to Table	Actions Tracked	Function
`decap_counter`	`decap_v4_table`	`decap`, `deny`	Counts successfully decapsulated packets and denied packets
`encap_counter`	`encap_v4_table`	`vxlan_v4_encap`, `deny`	Counts successfully encapsulated packets and denied packets

6.26.3.2 Control Application (gateway.entries.json)

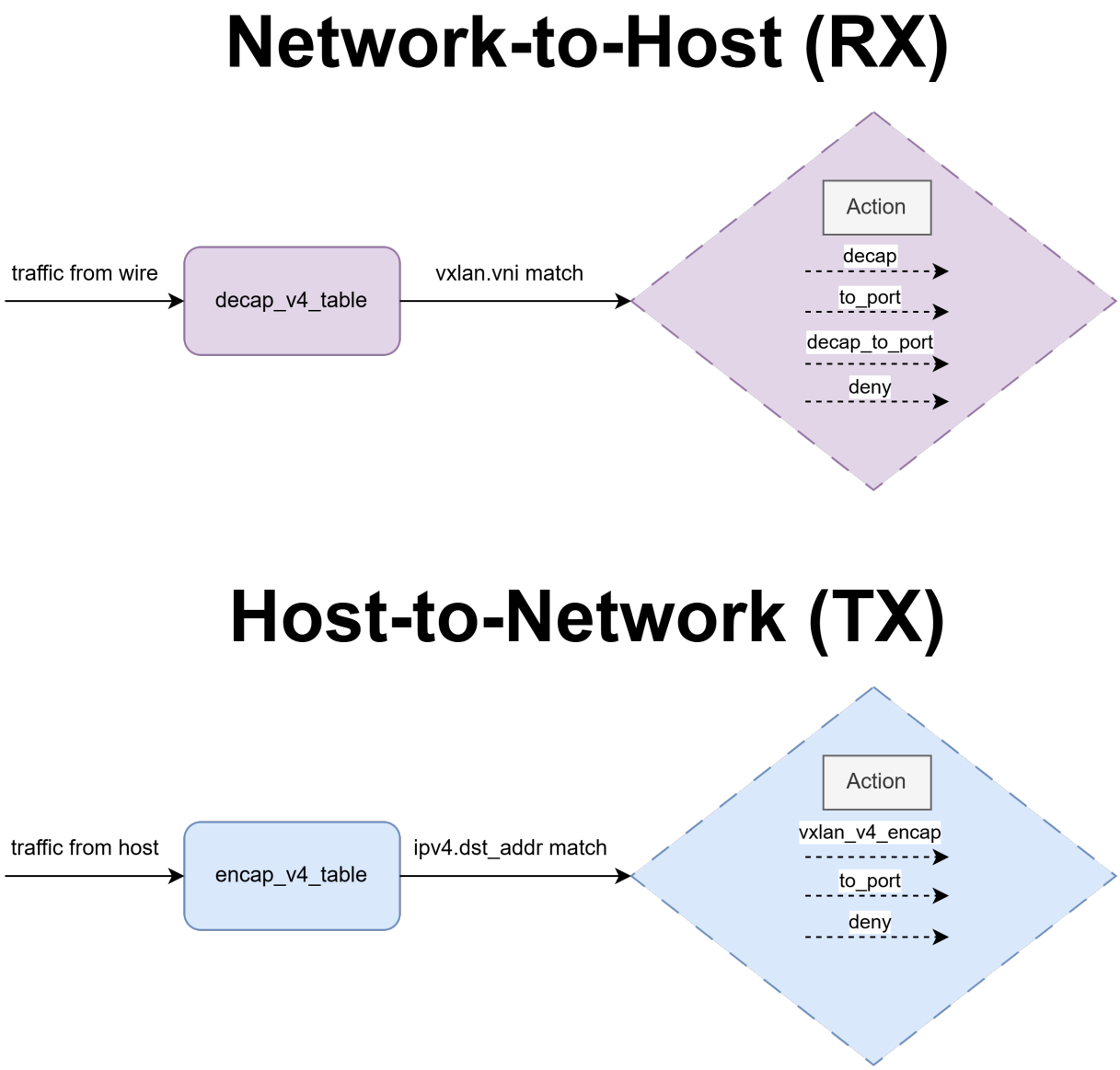
A control application manages the daemon's HW steering rules from a JSON input file that describes the desired rules.

gateway.entries.json

```
{
    "doctype" : "gateway_shm.p4",
    "tables": {
    "encap_v4_table": {
        "entries": [
            {
                "match": {
                    "headers.ipv4.dst_addr": "6.6.6.4"
                },
                "action": "vxlan_v4_encap_encap_v4_table",
                "params": {
                    "underlay_src_mac": "3C:6D:66:11:11:11",
                    "underlay_dst_mac": "ff:ff:ff:ff:ff:ff",
                    "underlay_sip": "6.6.6.3",
                    "underlay_dip": "6.6.6.2",
                    "vni": "1",
                    "port": "0"
                }
            },
            {
                "match": {
                    "headers.ipv4.dst_addr": "6.6.6.5"
                },
                "action": "vxlan_v4_encap_encap_v4_table",
                "params": {
                    "underlay_src_mac": "3C:6D:66:11:11:11",
                    "underlay_dst_mac": "ff:ff:ff:ff:ff:ff",
                    "underlay_sip": "6.6.6.3",
                    "underlay_dip": "6.6.6.2",
                    "vni": "1",
                    "port": "0"
                }
            }
        ]
    },
    "decap_v4_table": {
        "entries": [
            {
                "match": {
                    "headers.vxlan.vni": "1"
                },
                "action": "decap_to_port_decap_v4_table",
                "params": {
                    "port": "1"
                }
            },
            {
                "match": {
                    "headers.vxlan.vni": "2"
                },
                "action": "deny_decap_v4_table"
            }
        ]
    }
    }
}
```

6.26.3.3 Pipeline

The following diagram shows a schematic of the **gateway_shm.p4** program. The program defines dynamic **decap_v4_table** and **encap_v4_table** tables with no pre-defined rules. The DPL Runtime Controller uses **gateway.entries.json** to insert custom rules with the desired match values and actions into these tables.



6.26.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DPL Runtime Controller SDK

Refer to their respective programming guide for more information.

6.26.5 Compiling the Application

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for the application is located in the following directory:

```
/opt/dpl_rt_controller/samples/gateway_shm/
```

6.26.5.1 Prerequisites

- The application relies on the **dpl_rt_controller** library.
- The application relies on the **json-c** open source, requiring the following package to be installed:

  ```
  sudo apt install libjson-c-dev
  ```

- DPL Development Container installed on the host. See the DPL Installation Guide for more details.

6.26.5.2 Compiling the Application

To build the Gateway application using the dpl-dev container and the DPL compiler:

1. On the x86 host, compile the DPL **gateway_shm.p4** program:

   ```
   dplp4c.sh --target doca --odir /tmp/gateway_shm_out gateway_shm.p4
   ```

   Because this application relies on the Shared Memory (SHM) Interface, it is mandatory for the application to run directly on the DPU . Therefore, the compiler's output must be copied from the x86 host to the DPU's file system (e.g., **/tmp/gateway_shm_out**) before compiling the application

2. On the DPU Arm, compile the Gateway "C" application:

   ```
   cd /opt/dpl_rt_controller/samples/gateway_shm
   meson /tmp/gateway_shm -Dsample_programs_out=/tmp/gateway_shm_out
   ninja -C /tmp/gateway_shm
   ```

The **dpl_sample_gateway_shm** application is created in **/tmp/gateway_shm** .

6.26.6 Running the Application

> ⚠ The application is provided in source form and requires compilation before execution . For details, refer to section "Compiling the Application".

6.26.6.1 Prerequisites

- The DPL Runtime Service must be ready to run on the BlueField (Arm side)

6.26.6.2 Application Execution

1. Start the DPL Runtime Service as detailed in the DPL Container Deployment.
2. Run the gateway application. Usage:

```
Application usage instructions
```

```
./dpl_sample_gateway_shm <device_id> <p4info path> <program path> <json_entries_path>
```

> ⚠ The device id (first argument) should match the ID of the DPL device as configured at **/etc/dpl_rt_service/devices.d/** .

Example:

```
sudo /tmp/gateway_shm/dpl_sample_gateway_shm 1000 /tmp/gateway_shm_out/gateway_shm.p4info.txt /tmp/
gateway_shm_out/gateway_shm.dplconfig gateway.entries.json
```

6.26.7 Application Code Flow

1. gateway_main.cc:
   Parses the received arguments and executes **doca_error_t gateway(uint32_t device_id, const char *p4info_path, const char *blob_path, const char *json_entries_path)** , responsible for the main logic.
2. gateway_sample.cc:
   Implements **doca_error_t gateway()** and runs the main logic:
   a. Load the compiled program and p4info using gRPC:

```
DPL_P4RT_Controller::Controller ctrl = DPL_P4RT_Controller::Controller(device_id,
"localhost:9559");
ctrl.LoadProgram(p4info_path, blob_path);
```

   b. Connect to the DPL device:

```
dpl_rt_controller_connect(&connect_attr, &device);
```

   c. Insert the entries from **gateway.entries.json** using the SHM interface:

```
doca_error_t insert_entries_from_json(device, json_entries_path, entries) {
```

```
    ...
    // Process encap_v4_table/decap_v4_table entries
    for (int i = 0; i < entries_count; i++) {
        json_object *entry_obj = json_object_array_get_idx(entries_array, i);
        struct dpl_rt_controller_entry *dpl_entry = nullptr;
        dpl_shm_gateway_shm_gateway_over_encap_v4_table_entry_t *table_entry = nullptr;
        dpl_shm_gateway_shm_gateway_over_encap_v4_table_alloc_entry_mem(device, &dpl_entry,
&table_entry);
        ...
        /* Parse match fields, actions and parameters to construct the table_entry */
        ...
        // Insert the entry using the SHM API
        dpl_shm_gateway_shm_insert_entry(dpl_entry, nullptr);
        ...
    }
}
```

d. The program enters an interactive loop:

```
while (!quit) {
    std::cout << "Press a key (e=entries, c=counters, q=quit): " << std::flush;
    ...
    // Display all entries on 'E'/'e' key press
    display_all_entries(entries);
    ...
    // Read counter data on 'C'/'c' key press
    read_entry_counters(device, entries);
    ...
}
```

e. The system is now configured. Test packets can be sent (e.g., via **scapy**):

```
sendp(Ether(src=\"00:11:11:11:11:11\", dst=\"00:22:22:22:22:22\") / IP(src=\"192.168.1.1\",
dst=\"192.168.1.2\"), iface=\"eth2\", count=50)
```

f. On 'E'/'e' key press, all table entries are displayed:

```
void display_all_entries(entries) {
    ...
    // Iterate over entries per table ("encap_v4_table"/"decap_v4_table")
    for (const auto& table_pair : entries) {
        const std::string& table_name = table_pair.first;
        const std::vector<struct dpl_rt_controller_entry*>& table_entries = table_pair.second;
        ...
        // Iterate over entry handles
        for (size_t i = 0; i < table_entries.size(); i++) {
            void *shm_entry = dpl_rt_controller_table_entry_get_shm(table_entries[i]);
            ...
            /* Extract & display entry data using the controller interface */
            ...
        }
    }
}
```

g. On 'C'/'c' key press, counter data for each entry is read:

```
```

> The application displays counter values for each entry because **gateway_shm.p4** uses **direct_counters**. However, counters are only actively updated by the **decap**, **deny**, and **vxlan_v4_encap** actions. Therefore, only entries using these actions will show updated values.

h. On quit:

i. Cleanup entries:

```
doca_error_t cleanup_entries(entries) {
    ...
    // Iterate over entries per table
    for (const auto& table_pair : entries) {
        ...
        // Iterate over entry handles
        for (size_t i = 0; i < table_entries.size(); i++) {
            // Delete entry using the controller interface
            dpl_rt_controller_table_entry_delete(table_entries[i]);
            ...
        }
    }
```

```
}
```

ii. Disconnect from device and detach:

```
dpl_rt_controller_disconnect(device);
dpl_rt_controller_detach();
```

6.26.8 References

- DPL Runtime Service
- DPL Runtime Controller SDK
- DOCA Target Architecture
- P4 Language Support in DPL

6.27 DOCA DPU GPU Remote Offload Application Guide

This guide provides a reference implementation for developing a GPU offload application on top of NVIDIA® BlueField® DPUs and NVIDIA GPUs using the DOCA Comch and DOCA GPUNetIO libraries.

6.27.1 Introduction

The DOCA DPU GPU Remote Offload applications serve as a reference implementation, demonstrating how to develop a GPU offload application using DOCA Comch and DOCA GPUNetIO on the NVIDIA BlueField platform with NVIDIA GPUs.

These applications are simplified for clarity. A real-world solution would require more complexity tailored to a specific use case.

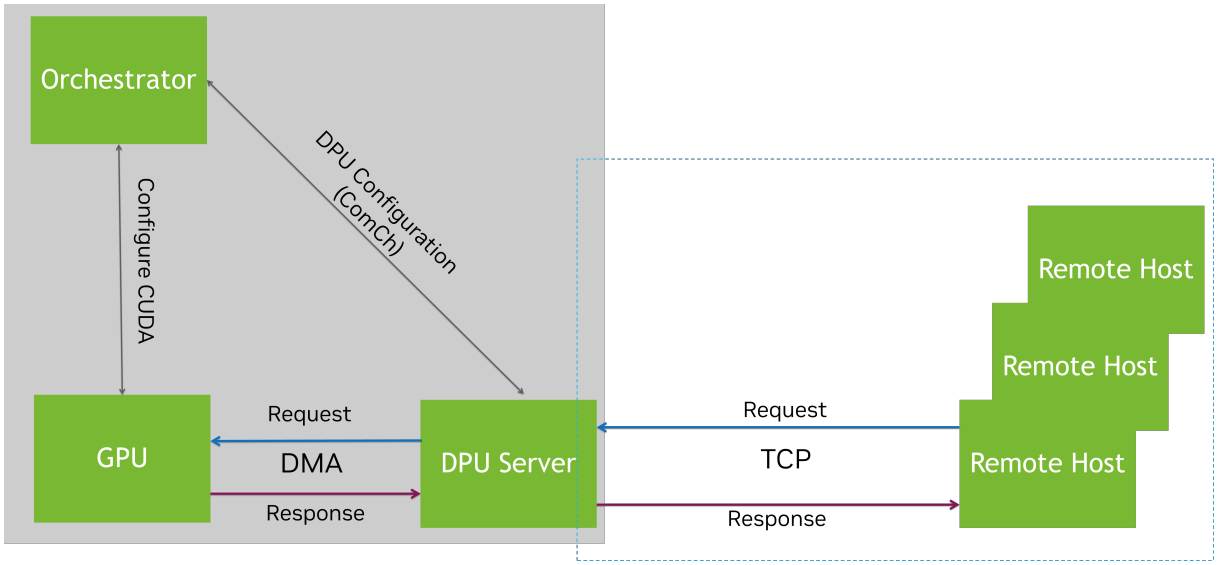
The general architecture involves:

- A server application running on the DPU.
- An orchestrator application running on the host CPU, which launches a CUDA Kernel.

The server receives TCP requests from remote clients and forwards them to the GPU via DOCA Comch. The GPU processes the message (in this reference, by reversing the byte order) and returns a response to the server. The server then forwards this response to the remote clients over TCP.

6.27.2 Applications

DOCA DPU GPU Remote Offload consists of three separate applications that work together to offload work from a DPU to a host-side GPU.



6.27.2.1 doca_dpu_gpu_remote_offload_server

This application runs on the BlueField DPU and acts as both a DOCA Comch server and a TCP server.

- DOCA Comch: Used to communicate with the Orchestrator and GPU.
- TCP: Used to communicate with remote clients for requests and responses

6.27.2.2 doca_dpu_gpu_remote_offload_orchestrator

This application configures and launches the CUDA Kernel for processing. It connects to the server via DOCA Comch, exchanges configuration information, and then launches the kernel . The Orchestrator application continues to run, monitoring the Comch connection and CUDA Kernel, and handles clean shutdown.

6.27.2.3 doca_dpu_gpu_remote_offload_client

This application sends requests to and receives responses from the server. It validates the responses and displays throughput figures.

6.27.3 Architecture

6.27.3.1 doca_dpu_gpu_remote_offload_server

The server application executes in two main phases: a Control Phase and a TCP Server Phase.

6.27.3.1.1 Control Phase

This phase begins when the application is launched. It configures the DOCA Comch server, waits for the orchestrator application to connect, and spawns threads to handle incoming messages. Once producers and consumers are connected, it creates a TCP listening socket.

A typical flow is:

1. Open the required DOCA device.
2. Create the DOCA Comch Server.
3. Wait for the Orchestrator application to connect.
4. Prepare the appropriate number of Socket threads. Each thread will:
    - Create DOCA Comch Producers and Consumers.
    - Wait for Producers and Consumers to be connected.
    - Create and pool **doca_comch_producer_send_tasks**.
    - Create and submit **doca_comch_consumer_post_recv_tasks**.
5. Create a TCP socket to listen for connections.
6. Start the TCP Server phase.

6.27.3.1.2 TCP Server Phase

In this phase, the application listens on the configured TCP port for incoming connections. When a client connects, its thread forwards messages to the GPU via DOCA Comch. It continues sending until the socket has no more data or the **send_task** pool is empty. It then waits for a response from the GPU, which is forwarded back to the remote client.

A typical per-thread flow loop is:

1. Poll the DOCA PE
    - If a **doca_comch_producer_send_task** completion is received, add the task back to the task pool.
    - If a **doca_comch_consumer_post_recv_task** completion is received:
        - Extract the response data.
        - Write the response to the TCP socket.
2. Poll TCP socket
    - If data can be read from the socket:
        - Read data from the socket.
        - Get the next available **doca_comch_consumer_post_recv** task (this should be **doca_comch_producer_send_task** based on context).
        - Copy message contents into the send task's data buffer.
        - Submit the send task.

In this phase, the application also listens for DOCA Comch control messages from the orchestrator or a CTRL-C signal to ensure a clean shutdown.

6.27.3.2 doca_dpu_gpu_remote_offload_orchestrator

This application executes in two phases: a Control Phase and a GPU Processing Phase.

6.27.3.2.1 Control Phase

This phase configures the DOCA Comch client and connects to the server. It creates producers and consumers, allocates GPU memory, waits for connections, and then launches the CUDA Kernel.

1. Open the required DOCA device.
2. Open the required GPU device.
3. Create the DOCA Comch client.
4. Connect to the DOCA Comch server.
5. Allocate the appropriate GPU memory.
6. Prepare the appropriate number of Producers and Consumers.
7. Launch the CUDA Kernel.

6.27.3.2.2 GPU Processing Phase

Once the CUDA Kernel is launched, the CPU portion of the application remains running. It monitors the kernel's execution and listens for DOCA Comch control shutdown messages (from the server or a user CTRL-C signal). If a shutdown is detected, it stops the kernel and cleans up memory.

The GPU processing portion initially submits multiple **post_recv** buffers. It then enters a loop, polling for messages. When a message is received, it processes it (reverses the bytes), sends the response back to the server, and resubmits the buffer. This continues until a fatal error or a global stop flag is set by the CPU.

A typical CUDA thread loop is:

1. Poll for **post_recv** messages
    - If a message is received:
        - Extract the data.
        - Verify the message is a client request.
        - Reverse the order of the bytes in the data.
        - Record the buffer in the **inflight_messages** array.
        - Submit the response to the server using **doca_dev_gpu_comch_producer_send**.
2. Poll for producer send message completions
    - If a send completion is indicated:
        - Use **user_msg_id** to determine which buffer was sent.
        - Submit this buffer to receive a new message with **doca_dev_gpu_comch_consumer_post_recv**.

6.27.3.3 doca_dpu_gpu_remote_offload_client

This application is simpler than the other two in the use case. Its purpose is to initiate TCP connections to the **doca_dpu_gpu_remote_offload_server**, using one connection per thread.

Each thread sends a specified number of request messages while recording throughput. Upon receiving a response, the client validates its content against the expected response; if they do not match, the application exits with an error.

If all requests are sent and successfully validated, the application outputs final statistics for the run, including:

- Run length
- Operation count
- Transmit and receive data rates
- IO rate

6.27.4 DOCA Libraries

6.27.4.1 Server

The **doca_dpu_gpu_remote_offload_server** application leverages the following DOCA libraries:

- DOCA Comch

6.27.4.2 Orchestrator

The **doca_dpu_gpu_remote_offload_server** application leverages the following DOCA libraries:

- DOCA Comch
- DOCA GPUNetIO

6.27.4.3 Client

The **doca_dpu_gpu_remote_offload_client** application does not leverage any DOCA libraries, however it does utilize DOCA Arg Parser.

6.27.5 Compiling the Applications

The **doca_dpu_gpu_remote_offload_orchestrator** application requires CUDA version 13. Compilation will fail if an older version is used.

The **doca_dpu_gpu_remote_offload_server** application will only compile on the DPU, and the **doca_dpu_gpu_remote_offload_orchestrator** will only compile on x86 hosts. The **doca_dpu_gpu_remote_offload_client** application compiles on all architectures.

6.27.5.1 Compiling All Applications

To build all DOCA applications together:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

The applications are created under **/tmp/build/dpu_gpu_remote_offload/** .

6.27.5.2 Compiling Only DPU GPU Remote Offload Applications

To directly build only the storage applications:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_dpu_gpu_remote_offload=true
ninja -C /tmp/build
```

Alternatively, you can edit **/opt/mellanox/doca/applications/meson_options.txt**:

1. Set **enable_all_applications** to **false**.
2. Set **enable_dpu_gpu_remote_offload** to **true**.
3. Run the compilation commands:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

The applications are created under **/tmp/build/dpu_gpu_remote_offload/**.

6.27.6 Running the Applications

The DOCA DPU GPU Remote Offload applications are provided in source form and must be compiled before execution.

6.27.6.1 Server

The server application must run on the NVIDIA BlueField DPU and must be started before the orchestrator or client.

6.27.6.1.1 Application Usage Instructions

```
Usage: doca_dpu_gpu_remote_offload_server [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -d, --device-id <DEV ID>          Device ID (mandatory).
  -r, --representor-id <REPRESENTOR ID> Representor ID (mandatory).
  -c, --comch-channel-name <DEV ID> Comch channel name (optional).
  -p, --server-listen-port <PORT>   Server listen port (mandatory).
  --cpu <CPU>                       CPU to use when executing data path operations (mandatory). May be repeated for
 more cores
  --max-concurrent-messages <NUM_MESSAGES> Set the maximum number of concurrent messages that can be processed per
thread (optional).
  --max-message-length <LENGTH>     Set the maximum length of a message that can be processed (exclusive of header)
(optional).
```

6.27.6.1.2 CLI Example of the Application

```
./doca_dpu_gpu_remote_offload_server --device-id 03:00.0 --representor-id d8:00.0 \
    -p 12345 --max-concurrent-messages 128 --cpu 1 --cpu 2
```

The DOCA Comch device PCIe address (**03:00.0**) and the representor PCIe address (**d8:00.0**) must match the addresses of the desired devices.

6.27.6.1.3 Command Line Flags

6.27.6.1.3.1 General Flags

Short Flag	Long Flag	Description
-h	**--help**	Prints a help synopsis and exits
-v	**--version**	Prints program version information and exits
-l	**--log-level**	Sets the numeric log level for the application: • **10** – DISABLE • **20** – CRITICAL • **30** – ERROR • **40** – WARNING • **50** – INFO • **60** – DEBUG • **70** – TRACE (requires compilation with TRACE support)
N/A	**--sdk-log-level**	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	**--log-filter**	Filters logs from specific modules (comma-separated list)
-j	**--json**	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.27.6.1.3.2 Program Flags

Short Flag	Long Flag	Description
d	**device-id**	Comm Channel DOCA device PCIe address. This is a mandatory flag

Short Flag	Long Flag	Description
r	representor-id	Comm Channel DOCA device representor PCIe address. ⚠ This is a mandatory flag
c	comm-channel-name	A custom name for the DOCA Comch connection, this must be the same on both server and orchestrator.
c	server-listen-port	The TCP port the server will use to listen for client connections.
N/A	cpu	CPU to use when executing data path operations, this option can be repeated to use multiple CPU cores in execution. ⚠ This is a mandatory flag
N/A	max-concurrent-messages	The maximum number of concurrent messages that can be processed by each thread.
N/A	max-message-length	The maximum length or message, in bytes, that can be supported.

6.27.6.2 Orchestrator

⚠ The orchestrator application runs on the host CPU and must be started after the server application.

6.27.6.2.1 Application Usage Instructions

```
Usage: doca_dpu_gpu_remote_offload_orchestrator [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -d, --device-id <DEV ID>          Device ID (mandatory).
  -g, --gpu-pci-addr <PCI ADDRESS>  GPU PCIe Address (mandatory).
  -c, --comch-channel-name <DEV ID> Comch channel name (optional).
  -t, --num-gpu-threads <NUM>       Number of GPU threads to use when executing data path operations (mandatory).
Must be the same as core count on server.
  --max-concurrent-messages <NUM_MESSAGES> Set the maximum number of concurrent messages that can be processed per
thread (optional).
  --max-message-length <LENGTH>     Set the maximum length of a message that can be processed (exclusive of header)
(optional).
```

6.27.6.2.2 CLI Example of the Application

```
./doca_dpu_gpu_remote_offload_orchestrator --device-id d8:00.0 --gpu-pci-addr b5:00.0 \
    --max-concurrent-messages 128 --num-gpu-threads 2
```

The device PCIe address (`d8:00.0`) and GPU PCIe address (`b5:00.0`) must match your devices.

The number of GPU threads should be the same as the total number of CPU cores provided to the server.

6.27.6.2.3 Command Line Flags

6.27.6.2.3.1 General Flags

Short Flag	Long Flag	Description
`-h`	`--help`	Prints a help synopsis and exits
`-v`	`--version`	Prints program version information and exits
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.27.6.2.3.2 Program Flags

Short Flag	Long Flag	Description
`d`	`device-id`	Comm Channel DOCA device PCIe address. This is a mandatory flag

Short Flag	Long Flag	Description
g	gpu-pci-addr	The PCIe address of the GPU to process messages ⚠ This is a mandatory flag
c	comm-channel-name	A custom name for the DOCA Comch connection, this must be the same on both server and orchestrator.
t	num-gpu-threads	The number of GPU threads to use in processing messages. This number should be the same as the total number of CPU cores assigned to the server ⚠ This is a mandatory flag
N/A	max-concurrent-messages	The maximum number of concurrent messages that can be processed by each thread.
N/A	max-message-length	The maximum length or message, in bytes, that can be supported.

6.27.6.3 Client

⚠ The client must be started after the server and orchestrator have established their Comch connection.

6.27.6.3.1 Application Usage Instructions

```
Usage: doca_dpu_gpu_remote_offload_client [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -s, --server-ip-address <IP ADDR> Server IP address (mandatory).
  -p, --server-ip-port <IP PORT>    Server IP port (mandatory).
  -t, --thread-count <THREAD_COUNT> Thread count (mandatory).
  -i, --iteration-count <ITERATION_COUNT> Iteration count (mandatory).
  -m, --message-string <MESSAGE>    message string (mandatory).
  -e, --expected-response <EXPECTED RESPONSE> Expected response (mandatory).
  --max-concurrent-messages <NUM_MESSAGES> Set the maximum number of concurrent messages that can be processed per
thread (optional).
```

6.27.6.3.2 CLI Example of the Application

```
./doca_dpu_gpu_remote_offload_client -s 172.17.0.1 -p 12345 -t 1 -i 1 -m "ABCD" -e "DCBA"
```

The GPU message response will contain the bytes in the request in reverse order.

6.27.6.3.3 Command Line Flags

6.27.6.3.3.1 General Flags

Short Flag	Long Flag	Description
`-h`	`--help`	Prints a help synopsis and exits
`-v`	`--version`	Prints program version information and exits
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.27.6.3.3.2 Program Flags

Short Flag	Long Flag	Description
`s`	`server-ip-address`	The IP address of the `doca_dpu_gpu_remote_offload_server`. This is a mandatory flag
`p`	`server-ip-port`	The TCP port assigned to the `doca_dpu_gpu_remote_offload_server`. This is a mandatory flag

Short Flag	Long Flag	Description
t	thread-count	The number of threads to use in execution. Each thread will send iteration-count number of messages. ⚠ This is a mandatory flag
i	iteration-count	The number requests to be sent from each execution thread. ⚠ This is a mandatory flag
m	message-string	The message to be sent to the server ⚠ This is a mandatory flag
i	expected-response	The expected response that should be received back from the server. ℹ This should be the bytes of the initial message in reverse order. ⚠ This is a mandatory flag
N/A	max-concurrent-messages	The maximum number of concurrent messages that can be processed by each thread.

6.28 DOCA Timescale Translation Application Guide

This guide provides an example of a Timescale Translation implementation on top of NVIDIA® BlueField® DPU.

6.28.1 Introduction

The DOCA Timescale Translation reference application demonstrates how to trigger and correlate events across processors operating on different clock domains within a BlueField DPU environment.

The application triggers events on the x86 host, the BlueField Arm cores, and the Data Path Accelerator (DPA) subsystem. It then leverages the DOCA Clock cross-timestamping functionality (available in DOCA Core) to correlate these events against the NVIDIA® ConnectX® real-time or free-running clock. This allows the application to determine the precise chronological ordering of events occurring across the different subsystems.

⚠ The DOCA Timescale Translation Application serves as a reference for using the DOCA Clock cross-timestamping APIs within the DOCA Core library.

It demonstrates how timestamps from independent system clocks can be compared using a shared reference clock.

However, it is not intended for correlating sequences of events that rely on independent timestamps.

For legacy reasons the implementation of the DOCA Timescale Translation application is referred to DOCA Time Sync

The application translates events timestamped by different clocks in the system to the ConnectX clock timeline. It will automatically detect whether the ConnectX device has the real-time clock or the free-running clock enabled.

To manually verify if the real-time clock is enabled on the device (if disabled, it defaults to free-running mode), use the following command, replacing **<device>** with your specific PCIe address or MST device:

```
sudo mlxconfig -d <device> --enable_verbosity q | grep REAL_TIME_CLOCK_ENABLE
```

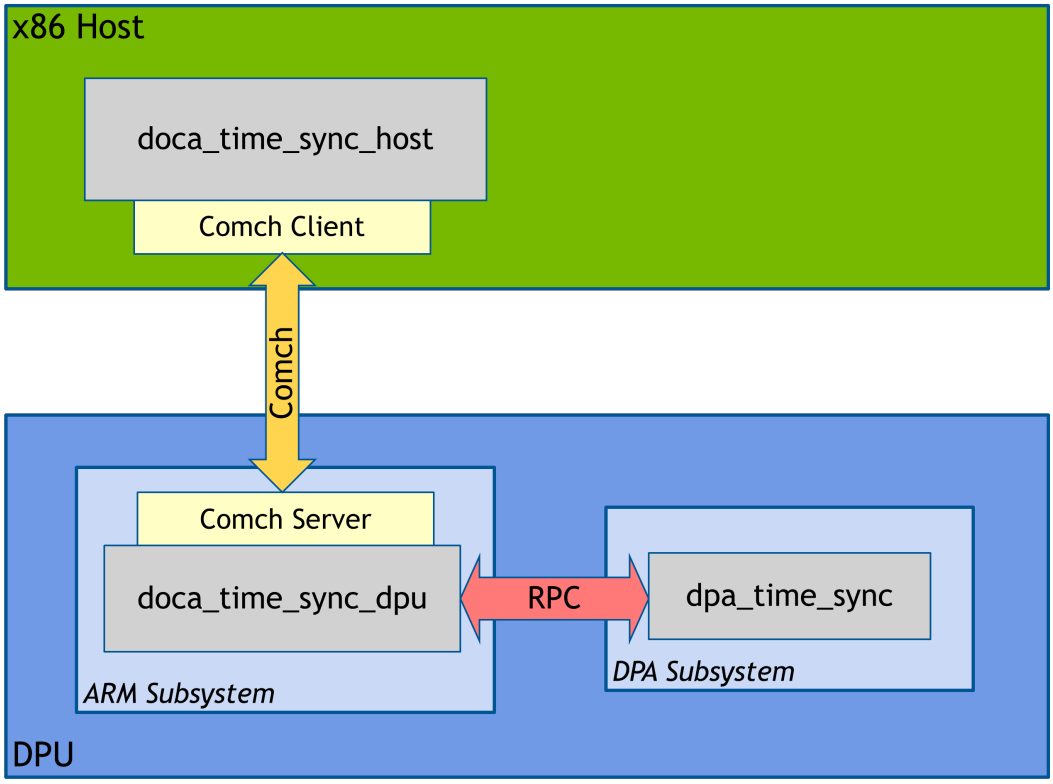
6.28.2 System Design

The DOCA Timescale Translation architecture consists of two distinct applications:

- DPU application (**doca_time_sync_dpu**) – Runs on the BlueField DPU Arm cores.
- Host application (**doca_time_sync_host**) – Runs on the x86 host system.

Operational workflow:

1. The DPU application must be started first. It establishes a DOCA Comch server to listen for incoming connections.
2. The Host application connects to the Comch server on the DPU.
3. Event triggering:
    a. Host to Arm: The Host application sends a message via the Comch connection to the DPU Arm cores, triggering the first event.
    b. Arm to DPA: The DPU application loads a DOCA DPA kernel and communicates with it to trigger a second event on the DPA subsystem.
4. All event timestamps are relayed back to the Host application, where they are correlated and ordered based on the common DOCA Clock.



6.28.3 Application Architecture

The DOCA Timescale Translation application generates a sequence of four events across three distinct processors, inserting variable delays between each step to simulate real-world processing latency.

6.28.3.1 Event Sequence

The application tracks the following four events:

1. x86 Host: Package and send a request message to the DPU
2. BlueField Arm Cores: Receive the message from the Host
3. DPA Subsystem: Execute a Remote Procedure Call (RPC) triggered by the Arm cores
4. x86 Host: Receive the response message from the DPU

6.28.3.2 Clock Correlation

Each event records a timestamp using its processor's local clock:

- x86 Host & Arm Cores: Use the system RTC.
- DPA Subsystem: Uses its internal local timer.

To order these events chronologically, all timestamps are correlated against a single common reference clock: the ConnectX NIC clock embedded in the BlueField DPU.

6.28.3.2.1 Host and Arm Core

Both the x86 and DPU applications utilize the DOCA Clock cross-timestamping library (part of DOCA Core). This library captures the local clock time and the common NIC clock time simultaneously to establish a precise correlation.

6.28.3.2.2 DPA Subsystem

The DPA kernel can only capture time using its local timer. To translate this local DPA timestamp to the common NIC clock, the Host application performs a retrospective calculation:

1. Use DOCA Clock to capture the current relationship between the NIC clock and the DPA timer.
2. Determine the duration (in seconds/nanoseconds) between the current DPA time and the recorded event time.
3. Subtract this delta from the current NIC time to derive the precise NIC time when the DPA event occurred.

6.28.3.3 Output Logging

The Host application aggregates all data and outputs it to a log file (**time_sync.log**). Each entry includes:

- Sync time: The calculated time on the common NIC clock.
- Local time: The raw timestamp from the processor's local clock.
- Accuracy: The margin of error for the correlation.
- Event description: A label identifying the specific event step.

6.28.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Comch
- DOCA DPA

Refer to their respective programming guide for more information.

6.28.5 Compiling the Application

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for the application is located in the following directory:

```
/opt/mellanox/doca/applications/time_sync/
```

6.28.5.1 Compiling All Applications

All DOCA applications are defined under a single Meson project. By default, the build process compiles all of them.

MPI is used for the compilation of this application. Make sure that MPI is installed on your setup (**openmpi** is provided as part of the installation of DOCA, as part of the **doca-all** and **doca-ofed** meta-packages).

Compiling the application requires updating the **LD_LIBRARY_PATH** and **PATH** environment variable to include MPI. For example, if **openmpi** is installed under **/usr/mpi/gcc/openmpi-4.1.7rc1**, then updating the environment variables should be like the following

```
export PATH=/usr/mpi/gcc/openmpi-4.1.7rc1/bin:${PATH}
export LD_LIBRARY_PATH=/usr/mpi/gcc/openmpi-4.1.7rc1/lib:${LD_LIBRARY_PATH}
```

To build all applications:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

The build system automatically detects the platform architecture.

- On x86 Host, it generates **doca_time_sync_host**
- On BlueField DPU, it generates **doca_time_sync_dpu**

The binary is created in **/tmp/build/time_sync/**.

6.28.5.2 Compiling Only the Current Application

To reduce build time, you can configure Meson to build only the Timescale Translation application.

Regardless of the method used, the binary (**doca_time_sync_host** or **doca_time_sync_dpu**) is created in **/tmp/build/time_sync/**.

6.28.5.2.1 Option 1: Command Line Configuration

Run the following commands to disable all applications and explicitly enable Timescale Translation:

```
cd /opt/mellanox/doca/applications/
```

```
meson /tmp/build -Denable_all_applications=false -Denable_time_sync=true
ninja -C /tmp/build
```

6.28.5.2.2 Option 2: Configuration File

edit the configuration file directly:

1. Edit `/opt/mellanox/doca/applications/meson_options.txt`.
   - Set `enable_all_applications` to `false`
   - Set `enable_time_sync` to `true`
2. Run the standard compilation commands:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

6.28.6 Running the Application

6.28.6.1 Application Execution

The Timescale Translation application is distributed as source code and must be compiled before execution.

6.28.6.1.1 Running on x86 Host

- Usage syntax:

```
Usage: doca_time_sync_host [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -p, --pci-addr                    DOCA device PCI address
  -d, --delay                       Delay (msecs) to insert between event triggers (optional)
```

- Example execution:

```
sudo ./doca_time_sync_host -p 3b:00.0 -d 1000
```

> **Root Privileges**
>
> The application requires `sudo` (root privileges) to access cross-timestamping system calls.

> **PCIe Addresses**
>
> Ensure `3b:00.0` matches your specific device's PCIe address.

6.28.6.1.2 Running on BlueField DPU

- Usage syntax:

```
Usage: doca_time_sync_dpu [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -p, --pci-addr                    DOCA device PCI address
  -r, --repr-addr                   DOCA device representor PCI address (optional)
```

- Example execution:

```
sudo ./doca_time_sync_dpu -p 03:00.0 -r 3b:00.0
```

Root Privileges

The application requires **sudo** (root privileges) to access cross-timestamping system calls.

PCIe Addresses

Ensure the device address (**03:00.0**) and representor address (**3b:00.0**) match your system configuration.

6.28.6.2 Command Line Flags

6.28.6.2.1 General Flags

Short Flag	Long Flag	Description
-h	**--help**	Prints a help synopsis and exits
-v	**--version**	Prints program version information and exits
-l	**--log-level**	Sets the numeric log level for the application: • **10** – DISABLE • **20** – CRITICAL • **30** – ERROR • **40** – WARNING • **50** – INFO • **60** – DEBUG • **70** – TRACE (requires compilation with TRACE support)
N/A	**--sdk-log-level**	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	**--log-filter**	Filters logs from specific modules (comma-separated list)

Short Flag	Long Flag	Description
-j	--json	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.28.6.2.2 Host Program Flags

Short Flag	Long Flag	Description
p	pci-addr	PCIe address of device to connect DOCA Comch client to, and to cross-timestamp against. This is a mandatory flag.
d	delay	Value in milliseconds of a delay to insert between the triggering of events. The parameter is optional. A default of 1 second (1000 msecs) will be used if excluded.

6.28.6.2.3 DPU Program Flags

Short Flag	Long Flag	Description
p	pci-addr	PCIe address of device to setup DOCA Comch server on, and to cross-timestamp against. This is a mandatory flag.
r	repr-addr	Representor address of the DOCA Comch device to use. This is an optional flag. If excluded, the first found representor associated with PCIe address will be used.

6.28.6.3 Troubleshooting

Refer to the *NVIDIA BlueField Platform Software Troubleshooting Guide* for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.28.7 Application Code Flow

6.28.7.1 Common

1. Parse application argument.
    a. Initialize arg parser resources and register DOCA general parameters.

```
doca_argp_init();
```

    b. Register Timescale Translation application parameters.

```
time_sync_common_reg_params();
```

    c. Parse the arguments.

```
doca_argp_start();
```

2. Open DOCA devices for use in the application:

```
time_sync_common_open_dev_with_caps();

// DPU only
time_sync_common_open_repr();
```

    a. Parse PCIe address for associated DOCA device
    b. Verify the selected device has the required capabilities
        i. Comch support
        ii. DPA timer support
        iii. DOCA DPA support (DPU only)
        iv. Repr Support (DPU only)
    c. Open valid device
    d. On DPU, parse and open repr device
3. Create a DOCA Clock Context:

```
time_sync_common_create_clock();
```

4. Run Host or DPU-specific code.
5. Destroy DOCA Clock Context:

```
time_sync_common_destroy_clock();
```

6. Close DOCA devices:

```
time_sync_common_close_devs();
```

7. Destroy Arg Parser:

```
doca_argp_destroy();
```

6.28.7.2 Host (x86) App

1. Initialize Comch client.

```
time_sync_host_init_comch_client();
```

   a. Create progress engine
   b. Create DOCA Comch Client context
   c. Configure taskpool/callbacks for sending and receiving messages
   d. Start Comch Client
2. Run main loop.

```
time_sync_host_main_loop();
```

   a. Progress until Client is fully connected to Comch Server on DPU
   b. Get event time (on host and NIC) before sending a message
   c. Create and send a message to DPU containing the input delay time in milliseconds
   d. Wait to receive a response from the DPU containing DPU and DPA event times
   e. Get event time (on host and NIC) of message receive
   f. Convert received DPA time to NIC time using cross-timestamping functions
   g. Log the local and common clock time of all events to 'time_sync.log'
3. Clean up Comch Client

```
time_sync_host_close_comch_client();
```

   a. Stop the DOCA Comch Client
   b. Progress until connection is fully shut down and context is IDLE
   c. Destroy Client context and progress engine

6.28.7.3 DPU App

1. Load DPA application.

```
time_sync_dpu_load_dpa_app();
```

   a. Create a new DOCA DPA context
   b. Add an app to the context (app with given name is compiled alongside DPU app - source in /opt/mellanox/doca/applications/time_sync/dpu_device/time_sync_dev.c)
   c. Start the DPA context
2. Initiate Comch Server.

```
time_sync_dpu_init_comch_server();
```

   a. Create progress engine
   b. Create DOCA Comch Server context
   c. Configure taskpool/callbacks for sending and receiving messages
   d. Configure callback for connection events
   e. Start Comch Server
3. Run main loop.

```
time_sync_dpu_main_loop();
```

   a. Progress until an x86 client has established a connection to the Comch Server
   b. Receive a message from Client

i. Extract the delay from request message
ii. Sleep for 'delay' milliseconds
iii. Get event time (on ARM and NIC) of message receive
iv. Sleep for 'delay' milliseconds
v. Trigger a remote procedure call to the loaded DPA app which will return its local time from running kernel
vi. Sleep for 'delay' milliseconds
vii. package event times from ARM/DPA and send response message back to host

c. Progress until x86 has finished and closed Client Comch connection

4. Clean up Comch Server

```
time_sync_dpu_close_comch_server();
```

a. Stop the DOCA Comch Server
b. Progress until context is IDLE
c. Destroy Server context and progress engine

5. Clean up DPA application

```
time_sync_dpu_unload_dpa_app();
```

a. Stop the DOCA DPA context
b. Destroy DPA context

6.28.8 References

- **/opt/mellanox/doca/applications/time_sync/**

6.29 DOCA Virtio-net Emulation Application Guide

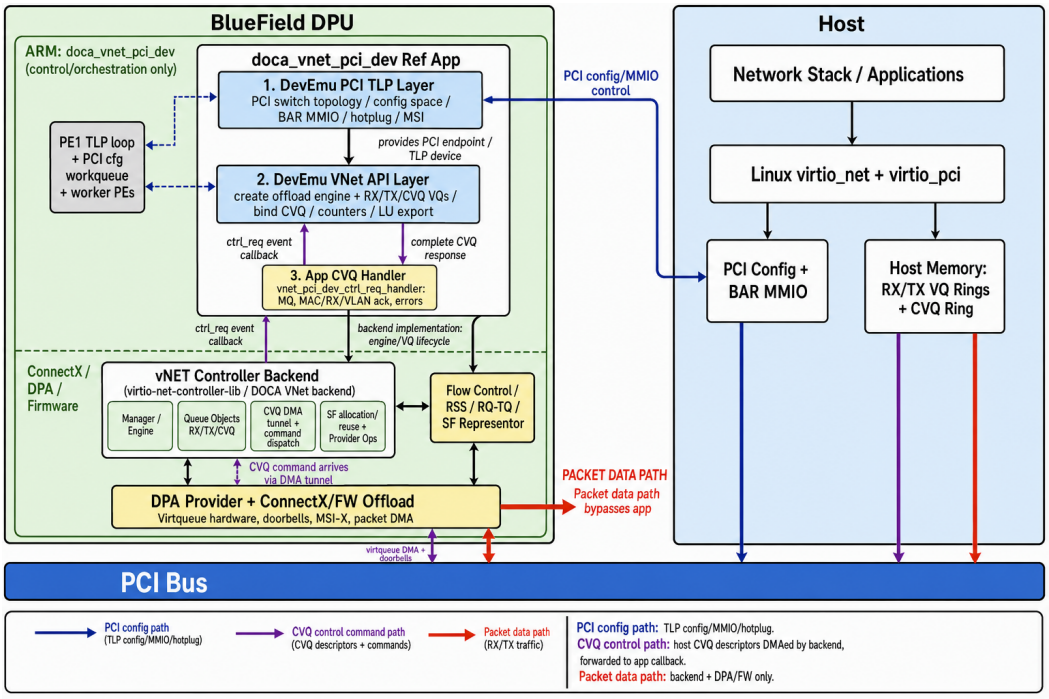
This guide describes the DOCA VNet PCI Device reference application in **doca/applications/vnet_pci_dev**.

6.29.1 1. Introduction

The DOCA VirtIO-net PCI Device Application emulates one or more **virtio-net** PCIe endpoints on NVIDIA BlueField DPUs. The application uses DOCA DevEmu PCI TLP APIs to emulate PCIe configuration/MMIO behavior and DOCA DevEmu VNet APIs to provide the hardware-backed VirtIO network datapath.

The host sees standard **virtio-net** PCI devices behind a software PCI switch topology. The DPU application owns PCI enumeration behavior, VirtIO PCI BAR emulation, MSI/MSI-X handling, hotplug signaling, queue lifecycle, control virtqueue handling, SR-IOV, and optional live update handover.

6.29.1.1 1.1. System Design



6.29.1.1.1 1.1.1. Host Side (Virtio-net Client Driver)

The host runs the standard Linux **virtio_net** driver.

Host responsibilities include:

- Enumerating the emulated PCI switch and VirtIO-net endpoints.
- Negotiating VirtIO feature bits.
- Programming VirtIO common config registers.
- Allocating and configuring RX/TX virtqueues.
- Configuring MSI-X vectors.
- Sending queue notifications through the notify BAR.
- Sending control virtqueue commands, especially multi-queue commands such as **VIRTIO_NET_CTRL_MQ_VQ_PAIRS_SET**.

The host network interface appears as a normal Linux NIC and can be inspected with tools such as **ip link**, **ethtool**, and **lspci**.

6.29.1.1.2 1.1.2. BlueField DPU (DOCA Virtio-net Emulation Application)

The DPU runs doca vnet_pci_dev.

The application:

- Creates a DOCA VNet PCI TLP device type.

- Creates a software PCI topology with one upstream port, several downstream ports (one downstream port per endpoint), and one VirtIO-net endpoint per downstream port.
- Handles PCIe Config Read/Write and MMIO Read/Write TLPs.
- Emulates VirtIO PCI modern capabilities and network device configuration.
- Creates one DOCA VNet offload engine per endpoint.
- Creates RX/TX virtqueue objects and an optional control virtqueue.
- Uses a PCI config workqueue and worker progress engines to keep the main TLP progress loop responsive.
- Supports static endpoint creation and runtime hotplug/unplug.
- Supports active/standby live update through `--lu-mode`.

6.29.1.1.3 1.1.3. Data Path Provider

Unlike the vblk refapp, the vnet refapp does not expose a `--provider DPA|DPU` flag.

The packet datapath is provided by DOCA DevEmu VNet offload and the underlying NIC/firmware path after the VirtIO queues and offload engine are configured. The application itself is mostly a PCI/VirtIO control-plane and lifecycle orchestrator. It does not implement a userspace packet copy loop for RX/TX data queues.

6.29.1.2 1.2. End-to-End Flow

```
device_status (host writes)   RESET(0)-> ACK(1)-> DRIVER(3)-> FEATURES_OK(0xB)-> DRIVER_OK(0xF)-> OPERATIONAL
                                                            |                     |
                              (DPU, async on FEATURES_OK)   |   (DPU, async on DRIVER_OK)
                              engine start; create RX/TX VQs;   |   apply VQ shadow + start VQs;
                              create CVQ IO ctx (if MQ)     |   bind CVQ; enable engine; stats
 host writes 0  -> device_status held at NEEDS_RESET(0x40) until async controller cleanup completes -> 0
```

1. The DPU application starts and creates the PCI TLP type, TLP channel, software PCI topology, and VirtIO-net device model.
2. In static mode, endpoints are created immediately. In hotplug mode, endpoints are created after `plug <idx>`.
3. The host enumerates the PCI topology and discovers VirtIO-net endpoints.
4. The host `virtio_net` driver reads features and device configuration such as MAC, MTU, link status, speed, duplex, and maximum queue pairs.
5. The driver writes VirtIO status bits: `ACKNOWLEDGE`, `DRIVER`, `FEATURES_OK`, and `DRIVER_OK`.
6. On `FEATURES_OK`, the DPU prepares the offload engine, creates VQ objects, and creates the CVQ IO context if multi-queue is negotiated.
7. On `DRIVER_OK`, the DPU applies queue configuration, starts VQs, binds CVQ, and enables the offload engine.
8. Packet RX/TX proceeds through the DOCA VNet offload datapath.
9. Runtime CVQ commands can adjust the number of active queue pairs.
10. Reset, hotplug, unplug, shutdown, and live update paths tear down or transfer state while preserving PCI/VirtIO correctness.

6.29.2 2. Application Architecture

6.29.2.1 2.1. Core Components

Key files:

- **vnet_pci_dev.c** : **main()** , logging, signal handling, arguments registration, top-level config.
- **vnet_pci_dev_core.c** : application orchestration, topology setup, controller lifecycle, VirtIO status callback, hotplug, MQ handling, progress loop.
- **vnet_pci_device.c** : PCI config workqueue, TLP channel callback, config-space emulation, MMIO emulation, VirtIO PCI device model.
- **vnet_pci_device.h** : public interfaces, BAR layout, queue limits, **struct vnet_pci_device** .
- **vnet_pci_dev_core.h** : **struct vnet_pci_dev_config** , **struct vnet_pci_dev_controller** , feature defaults.
- **pci_spec_tlp.h** : PCI/TLP constants, topology macros, **struct tlp_context** , **struct pci_device_config** .
- **vnet_virtio_types.h** : VirtIO-net structures, status bits, feature bits, CVQ command definitions.
- **vnet_pci_dev_lu.c** and **vnet_pci_dev_lu.h** : live update active/standby handover.

Important data structures:

- **struct tlp_context** : global TLP/PCI state, DOCA device, PCI type, TLP channel, PE1, topology, BDF map, ACG queue, VirtIO devices, VNet controllers, and runtime parameters.
- **struct vnet_pci_device** : one per endpoint; holds VirtIO common config, network config, feature state, VQ shadow array, reset generation, and status callback.
- **struct vnet_pci_dev_controller** : one per endpoint; owns the DOCA VNet offload engine, RX/TX VQ arrays, CVQ, VNet IO context, worker PE, counters/stats, and atomic lifecycle state.
- PCI config workqueue: moves slow operations away from the main TLP progress loop.
- PE1: main progress engine for TLP handling.
- PE2: per-controller worker progress engines used by the workqueue and CVQ IO contexts.

6.29.2.1.1 2.1.1. Virtio-net Device Model

The app models a modern VirtIO-net PCI endpoint.

Each endpoint exposes:

- VirtIO PCI vendor/device IDs.
- PCI class code for network controller.
- PCIe capability.
- MSI-X capability.
- VirtIO common config capability.
- VirtIO notify capability.
- VirtIO ISR capability.
- VirtIO device config capability.
- VirtIO PCI config access capability.

The network device config contains:

- MAC address.
- Link status.
- Maximum virtqueue pairs.
- MTU.
- Speed.
- Duplex mode.
- RSS-related size/hash fields.

The base MAC comes from `--mac-addr`. Per-endpoint MACs are generated by incrementing the last octet by the endpoint index.

6.29.2.1.2 2.1.2. Virtio Queues

```
RX VQ = qp*2   TX VQ = qp*2+1   CVQ = max_qp*2   Total VQs = 2*max_qp+1
default max_qp=8   hard max=127   initial active qp=1   queue size default 1024, 16..4096, power-of-two
```

Queue model:

- RX VQ index: `queue_pair_index * 2`.
- TX VQ index: `queue_pair_index * 2 + 1`.
- CVQ index: `max_queue_pairs * 2`.
- Total VQs: `max_queue_pairs * 2 + 1`.

Defaults and limits:

- Default max queue pairs: `8`.
- Maximum queue pairs: `127`.
- Initial active queue pairs: `1`.
- Default queue size: `1024`.
- Queue size range: `16-4096`.
- Queue size must be a power of two.

The application maintains a shadow copy of host-programmed VQ config in `struct vnet_pci_device`. Worker threads snapshot this shadow state before applying it to DOCA VQs, avoiding races with reset.

6.29.2.1.3 2.1.3. PCIe TLP Emulation

The application handles PCIe TLPs directly.

TLP handling flow:

1. `vnet_pci_dev_event_cb()` receives a TLP channel request.
2. ACG requests are cached for later MSI Memory Write TLPs.
3. PCI event requests are completed directly.
4. Normal requests are classified as config read/write, memory read/write, or unsupported.
5. The target device is resolved through BDF lookup or BAR address lookup.

6. Endpoint TLPs are protected with an endpoint read lock (to avoid racing hot-unplug destruction).
7. Config-space and MMIO handlers generate the correct completions.
8. Invalid/unmapped requests return Unsupported Request through the dummy device path.

The app emulates both Type 0 endpoint config space and Type 1 bridge config space.

6.29.2.1.4 2.1.4. PCI Switch Topology

The topology is:

```
USP -+- DSP[0] - EP[0]      USP: upstream port (Type-1)   DSP: downstream port per EP (Type-1 + HP slot)
     +- DSP[N] - EP[N]      EP: virtio-net (Type-0)       DUMMY: Unsupported Request for invalid BDFs
```

DUMMY device for unsupported BDF routing

Components:

- **USP** : upstream switch port, root of the software switch.
- **DSP** : downstream switch port, one per endpoint, with PCIe hotplug slot capability.
- **EP** : VirtIO-net endpoint.
- **DUMMY** : returns Unsupported Request for invalid probes.

The host assigns bus/device/function numbers during enumeration. The app caches BDF mappings and updates them on bus renumbering.

6.29.2.1.5 2.1.5. SR-IOV

Each PF can expose up to 64 VFs (each VF an independent virtio-net device, usable on the host or passed to a VM).
One SF backs each VF (its offload-engine representor must be OVS-bridged for RX).
ARI places VFs at functions 1..N of the PF bus (**first_vf_offset=1** , **vf_stride=1**), so >7 VFs/PF work;
VF Device ID **0x1041** .
VF queue count mirrors the PF **-q** .
Per-VF MSI-X num = per-VF doorbells = **2 * max_qp + 2** .
Enabled by **--total-vfs** .

6.29.2.1.6 2.1.6. Threading & latency model (core invariant)

```
PE1 (main thread): services host config/MMIO TLP completions, < 1 ms, MUST NOT BLOCK
    | submit heavy op
    v
PCI config workqueue (preallocated 1024-item pool)  ->  worker on PE2 (one per controller):
    engine start/enable | VQ create/config | CVQ IO-ctx | controller cleanup |
    hotplug create/destroy | MQ start (<=2 concurrent) | diagnostics/stats
```

Any operation that can block on firmware/DOCA transitions is moved off PE1, or the host's non-posted reads stall (Linux probe/reboot soft-lock).
Reset safety: per-device reset generations, cancel flags, VQ-shadow snapshots, bounded waits.

6.29.2.2 2.2. Initialization Process

The initialization process is:

1. Parse command line arguments.
2. Allocate and initialize **`tlp_context`**.
3. Open the DOCA device by IB device name or PCI address.
4. Create PE1 for TLP progress.
5. Configure optional CPU affinity.
6. Initialize the PCI config workqueue.
7. Verify VNet PCI TLP support.
8. Create the VNet PCI TLP type.
9. Attach the DOCA device to the PCI type.
10. Configure MSI-X count and doorbell count.
11. Start the PCI type.
12. Query and validate the predefined VirtIO-net BAR layout.
13. Create and configure the TLP channel.
14. Connect the TLP channel to PE1 and start it.
15. Initialize software PCI topology.
16. Initialize DOCA VNet subsystem.
17. Create VirtIO-net device models.
18. Create endpoints immediately in static mode, or defer endpoint creation in hotplug mode.
19. Allocate per-endpoint transaction regions.
20. Create one worker PE per controller.
21. Enter the progress loop.

6.29.2.3 2.3. Reset and Shutdown

Reset is triggered when the host writes **`device_status = 0`**.

The app:

- Marks the device as canceling in-progress worker operations.
- Increments the per-device reset generation.
- Reinitializes VirtIO queue shadow state.
- Holds **`device_status`** at **`NEEDS_RESET`** while controller cleanup is pending.
- Disables/stops/destroys VQs.
- Stops and destroys the CVQ IO context when needed.
- Keeps the offload engine object alive for normal driver rebind.
- Destroys the offload engine only during endpoint destruction or final shutdown.

Shutdown:

- Stops accepting new workqueue items.
- Waits for outstanding work and MQ-start threads.
- Destroys VirtIO device models.
- Destroys endpoint controllers, TLP devices, and representors.
- Flushes ACG credits.
- Stops and destroys the TLP channel.

- Stops/destroys PCI type and DOCA VNet subsystem.
- Destroys worker PEs and PE 1.
- Frees `tlp_context`.

6.29.3 3. DOCA Libraries

The application uses:

- DOCA DevEmu PCI TLP: PCIe TLP channel, PCI type, TLP devices, ACG, MSI/MSI-X support.
- DOCA DevEmu VNet: VirtIO-net offload engine, RX/TX VQs, CVQ, counters.
- DOCA DevEmu VirtIO: generic VirtIO offload engine and VQ lifecycle APIs.
- DOCA PE: progress engines for TLP and worker-side asynchronous operations.
- DOCA Arg Parser: command-line parsing.
- DOCA Log: application and SDK logging.
- `libibverbs`: used by build and live update/device handling paths.
- Optional `libbsd`: used for `strlcpy` when available.

6.29.4 4. Dependencies

Required:

- BlueField platform with DOCA DevEmu VNet PCI TLP support.
- DOCA SDK including DevEmu support.
- `libibverbs`.
- Firmware/device support for VirtIO-net BAR layout.
- PCI switch/TLP emulation enabled in firmware.

The app checks that the VNet TLP type is supported and that the TLP channel exposes exactly one NV switch TLP DSP. If not, initialization fails and asks for firmware configuration adjustment.

Unlike the vblk refapp, this code does not create application-managed DMA buffer pools or hugepage-backed data buffers for packet data. Packet movement is handled by DOCA VNet offload.

6.29.5 5. Compiling the Application

6.29.5.1 5.1. Compiling All Applications

```
cd /opt/mellanox/doca/applications/
meson setup /tmp/build && ninja -C /tmp/build
```

6.29.5.2 5.2. Compiling Only the Current Application

```
cd /opt/mellanox/doca/applications/
meson setup /tmp/build -Denable_all_applications=false -Denable_vnet_pci_dev=true && ninja -C /tmp/build
```

Alternatively, edit `applications/meson_options.txt`:

- Set `enable_all_applications` to `false`.

- Set **enable_vnet_pci_dev** to **true**.

Then run the normal meson && ninja build.

6.29.5.3 5.3. Build Outputs

The build produces one executable:

doca_vnet_pci_dev

Typical output path:

/tmp/build/vnet_pci_dev/doca_vnet_pci_dev

Live update is implemented inside the same executable through **--lu-mode**; there are no separate ***_lu** and ***_emu** binaries in the current vnet app.

6.29.6 6. Running the Application

6.29.6.1 6.1. Prerequisites

A typical setup must enable TLP and PCI switch emulation in firmware. The code expects one NV switch TLP downstream port.

Example firmware configuration pattern:

```
mlxconfig -y -d /dev/mst/mt41692_pciconf0 reset
mlxconfig -y -d /dev/mst/mt41692_pciconf0 set TLP_EMULATION_NUM_PF=32
mlxconfig -y -d /dev/mst/mt41692_pciconf0 set TLP_EMULATION_ENABLE=1
mlxconfig -y -d /dev/mst/mt41692_pciconf0 set PCI_SWITCH_EMULATION_ENABLE=1
mlxconfig -y -d /dev/mst/mt41692_pciconf0 set PCI_SWITCH_EMULATION_NUM_TLP_PORT=1
mlxconfig -y -d /dev/mst/mt41692_pciconf0 set PCI_SWITCH_EMULATION_NUM_PORT=2
```

SR-IOV supported up to 256 VFs, thus need set 320 TOTAL_SF (256 + 32 + 32).

```
mlxconfig -y -d /dev/mst/mt41692_pciconf0 s PF_BAR2_ENABLE=0 PER_PF_NUM_SF=1 PF_SF_BAR_SIZE=8 PF_TOTAL_SF=320
```

Cold-reboot is required after firmware configuration changes.

6.29.6.2 6.2. Application Usage

Usage: doca_vnet_pci_dev [DOCA Flags] [Program Flags]

Common DOCA flags include:

- **-h, --help**
- **-v, --version**
- **-l, --log-level**
- **--sdk-log-level**
- **--log-filter**
- **-j, --json**

Program flags are described in the Command Line Flags section below.

6.29.6.2.1 6.2.1. Example Command Line Execution

Run with defaults in hotplug mode:

```
./doca_vnet_pci_dev [-H 1]
```

Run in static mode with four endpoints and 16 max queue pairs:

```
./doca_vnet_pci_dev -n 4 -q 16 -H 0
```

Run with custom MAC, MTU, and speed:

```
./doca_vnet_pci_dev -m 52:54:00:12:34:56 -t 9000 -s 100000
```

Run with explicit CPU affinity:

```
./doca_vnet_pci_dev --tlp-core 8 --worker-core 9 --mq-core 10
```

Run with SR-IOV (4 PF x 4 VF x 4 QP):

```
./doca_vnet_pci_dev -H 0 -n 4 --total-vfs 4 -q 4
```

Run in live update active mode(-H 0 for static mode, -H 1 for hotplug mode):

```
./doca_vnet_pci_dev -H 0 --lu-mode active
```

Run a standby process for live update:

```
./doca_vnet_pci_dev -H 0 --lu-mode standby
```

Run with MRG_RXBUF and GUEST_CSUM features enabled (driver need to negotiate)

```
./doca_vnet_pci_dev -H 0 -t 9000 --mrg-rxbuf --guest-csum
```

6.29.6.2.2 6.2.2. Runtime Commands

Available through stdin:

- **plug <DSP_IDX>** : create and hotplug an endpoint into a DSP slot. Hotplug mode only.
- **unplug <DSP_IDX>** : request hot-unplug of the endpoint in a DSP slot. Hotplug mode only.
- **speed <EP_IDX> <Mbps>** : update the reported link speed and raise a config-change MSI-X notification.

Examples:

```
plug 0
unplug 0
```

```
speed 0 25000
```

6.29.6.2.3 6.2.3. Verifying the Device (Host Side)

After startup or hotplug:

```
lspci | grep -i virtio
ip link
```

Inspect queue support:

```
ethtool -l <interface>
```

Change active queue pairs:

```
ethtool -L <interface> combined 4
```

Inspect link information:

```
ethtool <interface>
```

6.29.6.3 6.3. Command Line Flags

6.29.6.3.1 6.3.1. General Flags

The app uses DOCA Arg Parser, so standard DOCA flags are available:

- **-h, --help** : print usage.
- **-v, --version** : print version.
- **-l, --log-level** : set application log level.
- **--sdk-log-level** : set SDK log level.
- **--log-filter** : filter logs by module.
- **-j, --json** : parse flags from JSON.

6.29.6.3.2 6.3.2. Program Flags

- **-p, --pci-addr** : DOCA device PCI address. Default: **0000:03:00.0** .
- **-d, --ibdev-name** : IB device name. Overrides **--pci-addr** .
- **-m, --mac-addr** : base MAC address. Default: **52:54:00:12:34:56** .
- **-t, --mtu** : MTU, range **68-9000** . Default: **1500** .
- **-s, --speed** : link speed in Mbps. Default: **100000** .
- **-D, --duplex** : duplex mode, **0** half or **1** full. Default: **1** .
- **-q, --max-queue-pairs** : maximum queue pairs, range **1-127** . Default: **8** .
- **-z, --queue-size** : queue size, power of two, range **16-4096** . Default: **1024** .
- **-n, --num-ep** : number of endpoints, range **1-32** . Default: **1** .
- **-H, --hotplug-mode** : **1** hotplug, **0** static. Default: **1** .
- **--tlp-core** : CPU core for the main TLP progress thread, or **-1** for auto.

- `--worker-core` : CPU core for the PCI config worker thread, or `-1` for auto.
- `--mq-core` : CPU core for MQ start threads, or `-1` for auto.
- `--lu-mode` : live update mode: `none` , `active` , or `standby` . Default: `none` .

Validation rules:

- `--tlp-core` must not equal `--worker-core` when both are explicitly set.
- `--mq-core` must not equal `--tlp-core` when both are explicitly set.
- `--queue-size` must be a power of two.
- `--mac-addr` last octet must leave enough room for per-endpoint incrementing.

6.29.6.3.3 6.3.3. Optional Features Flags

- `--guest-csum` Driver handles packets with partial checksum (VIRTIO_NET_F_GUEST_CSUM)
- `--guest-tso4` Driver can receive TSOv4 (VIRTIO_NET_F_GUEST_TSO4)
- `--guest-tso6` Driver can receive TSOv6 (VIRTIO_NET_F_GUEST_TSO6)
- `--guest-ecn` Driver can receive TSO with ECN (VIRTIO_NET_F_GUEST_ECN)
- `--guest-ufo` Driver can receive UFO (VIRTIO_NET_F_GUEST_UFO)
- `--host-ecn` Device can receive TSO with ECN (VIRTIO_NET_F_HOST_ECN)
- `--host-ufo` Device can receive UFO (VIRTIO_NET_F_HOST_UFO)
- `--mrg-rxbuf` Driver can merge receive buffers (VIRTIO_NET_F_MRG_RXBUF)

6.29.6.4 6.4. Troubleshooting

Common issues:

- VNet TLP type unsupported: the device or firmware does not expose VirtIO-net emulation support.
- Wrong TLP port count: the app expects exactly one NV switch TLP DSP.
- Host does not see devices: verify firmware settings, run PCI rescan, and confirm the app is in static mode or that `plug <idx>` was issued.
- Hotplug command rejected: host hotplug interrupt may not be enabled yet, or a previous unplug is still in progress.
- Queue startup takes time with high `--max-queue-pairs` : MQ expansion is intentionally deferred and backgrounded.
- RSS/hash CVQ commands fail: the app does not implement those CVQ commands.
- Host stalls during development changes: keep slow DOCA operations off PE1; the existing workqueue design is there to avoid PCIe completion timeouts.

6.29.7 7. Application Code Flow

6.29.7.1 7.1. Framework Initialization

6.29.7.1.1 7.1.1. Initialization Flow

1. `main()` initializes logging.

2. `doca_argp_init()` initializes argument parsing.
3. `register_vnet_pci_dev_params()` registers application-specific flags.
4. `doca_argp_start()` parses CLI/JSON input.
5. Signal handlers are installed for `SIGINT` and `SIGTERM`.
6. `vnet_pci_dev_run()` receives the parsed config.
7. `init_tlp_context()` allocates topology state.
8. `find_doca_device()` opens the DOCA device by IB name or PCI address.
9. `init_progress_engine()` creates PE1.
10. `vnet_pci_device_configure_affinity()` records affinity preferences.
11. `init_virtio_network_device()` initializes PCI/TLP, topology, VNet, VirtIO device state, and endpoint creation.
12. Worker PEs are created and attached to the PCI config workqueue.
13. `run_progress_loop()` starts processing TLPs, runtime commands, MSI retries, and LU triggers.

6.29.7.1.2 7.1.2. Key Points

- PE1 is reserved for TLP handling and must remain responsive.
- Heavy operations are submitted to the PCI config workqueue.
- Worker PEs are created per controller so multi-endpoint operations can progress independently.
- In LU standby mode, device and channel restore alter the normal open/start flow.

6.29.7.2 7.2. Resource Management

6.29.7.2.1 7.2.1. Resource Management Flow

Main resources:

- `tlp_context`
- DOCA device handle
- PE1 main progress engine
- VNet PCI TLP type
- TLP channel
- ACG credit queue
- software PCI device configs
- BDF map
- VirtIO device models
- VNet controller array
- per-controller worker PEs
- per-controller offload engine
- RX/TX/CVQ objects
- VNet IO context for CVQ
- stats/counters handles
- per-endpoint transaction region memory
- PCI config workqueue and work-item pool

Resource creation order:

1. Allocate `tlp_context`.
2. Initialize locks, maps, endpoint arrays, and controller arrays.
3. Open DOCA device.
4. Create PE1.
5. Initialize PCI config workqueue.
6. Create/start PCI type.
7. Create TLP channel.
8. Initialize topology and VNet subsystem.
9. Create VirtIO device models.
10. Create endpoints if static mode.
11. Allocate transaction regions.
12. Create worker PEs.
13. Create offload/VQ/IO resources lazily as the host reaches VirtIO states.

6.29.7.2.2 7.2.2. Key Points

- The workqueue uses a preallocated pool of 1024 work items to avoid hot-path allocation.
- MQ start concurrency is limited to 2 to avoid memory pressure and firmware command saturation.
- Reset generation fields prevent stale async work from affecting a post-reset device.
- Endpoint read/write locks protect TLP handlers from hotplug destruction races.
- Diagnostics run on the worker side, not PE1.

6.29.7.3 7.3. PCI Switch Topology and Device Configuration

6.29.7.3.1 7.3.1. PCI Switch Topology Flow

1. `init_device_topology()` initializes all bridge and endpoint config structures.
2. USP is configured as a Type 1 bridge with upstream-port PCIe capability.
3. Each DSP is configured as a Type 1 bridge with downstream-port and hotplug slot capabilities.
4. Each endpoint is configured as a Type 0 network device.
5. The dummy device is configured for unsupported request handling.
6. In static mode, `create_all_devices()` creates endpoint representors and TLP devices.
7. In hotplug mode, `create_device()` is called later from the hotplug work item.
8. Host enumeration assigns bus numbers and BDFs.
9. BDF mappings are cached and updated if the host renumbers buses.

Endpoint creation flow:

1. Create or reattach a representor.
2. Query representor VHCA ID.
3. Create the TLP device.
4. Start the TLP device.
5. Create the VNet controller and offload engine.
6. Mark endpoint present.

6.29.7.3.2 7.3.2. Key Points

- DSP count equals endpoint count.
- Each DSP supports PCIe hotplug slot behavior.
- Hotplug signaling uses attention button and slot status bits.
- MSI for hotplug is sent as a Memory Write TLP using ACG credits.
- If ACG credits are exhausted, MSI retry is scheduled.
- Static mode sets slots to powered-on and link-active at startup.

6.29.7.4 7.4. Virtio-net Controller

6.29.7.4.1 7.4.1. Controller Initialization Flow

Controller creation happens in `vnet_pci_dev_vnet_controller_create()`:

1. Resolve endpoint PF index.
2. Associate controller with the per-endpoint VirtIO device.
3. Convert TLP device to PCI endpoint.
4. Create a VNet offload engine, or import one during LU standby.
5. Configure MTU.
6. Configure MAC address.
7. Configure total number of queues.
8. Initialize controller atomics and deferred MQ state.
9. Allocate RX and TX VQ pointer arrays.
10. Query/log SF representor information.
11. Wait for host `FEATURES_OK` before starting hardware preparation.

On `FEATURES_OK`:

1. Validate driver features vector.
2. Determine if `VIRTIO_NET_F_CTRL_VQ` and `VIRTIO_NET_F_MQ` were negotiated.
3. Set initial active queue pairs.
4. Submit offload engine start.
5. Submit VQ object creation.
6. Submit CVQ IO context creation.

On `DRIVER_OK`:

1. Submit start-and-enable work.
2. Enable optional features if negotiated.
3. Snapshot VQ shadow config.
4. Configure RX/TX/CVQ VQs.
5. Start VQs.
6. Bind CVQ to IO context.
7. Enable VirtIO offload engine.
8. Create stats/counter resources.

6.29.7.4.2 7.4.2. Virtio Queue Lifecycle State Machine

Conceptual state flow:

```
UNCREATED --FEATURES_OK--> CREATED --host cfg--> CONFIGURED --start--> STARTED --enable--> ENABLED
   ^                                                                                       |
 DESTROYED <- STOPPED <- DISABLED <----------------- reset / group-disable -----------------+
```

Lifecycle by phase:

- **UNCREATED**: no DOCA VQ object exists.
- **CREATED**: RX/TX/CVQ object has been created after **FEATURES_OK**.
- **CONFIGURED**: host-provided queue size, MSI-X vector, descriptor address, driver address, and device address have been applied.
- **STARTED**: **doca_devemu_virtio_vq_start()** has completed.
- **ENABLED**: offload engine or VQ group enable has made the queue operational.
- **DISABLED**: VQ group or engine disable has stopped host-visible processing.
- **STOPPED**: VQ stop completed.
- **DESTROYED**: VQ object destroyed.

6.29.7.4.3 7.4.3. Key Points

- RX/TX VQs are created for the configured maximum queue pairs.
- CVQ is created only when MQ is negotiated.
- Data queues may be deferred during multi-endpoint startup so enumeration can settle.
- Increasing queue pairs completes the CVQ request immediately, then starts extra queues in background.
- Decreasing queue pairs disables and stops extra queues synchronously.
- Worker readers use snapshots of VQ shadow config to avoid reset races.

6.29.7.5 7.5. I/O Request Processing

6.29.7.5.1 7.5.1. I/O Request Processing Flow

For packet data:

1. Host **virtio_net** driver posts RX buffers and TX descriptors.
2. Host notifies queues through the VirtIO notify region.
3. DOCA VNet offload processes packet datapath using the configured VQs.
4. The application does not copy packet payloads in userspace.
5. Completion and interrupt behavior are handled through the DOCA VNet/VirtIO offload path.

For CVQ control requests:

1. Host submits a control command on CVQ.
2. DOCA VNet forwards the request to the registered IO context.
3. Worker PE progresses the IO context.
4. **vnet_pci_dev_ctrl_req_handler()** receives **cls** and **cmd**.
5. MQ pair-set commands are handled by the app.

6. MAC/RX/VLAN classes are acknowledged as backend-handled.
7. Unsupported classes/commands return **VIRTIO_NET_ERR**.
8. The request is completed with **doca_devemu_vnet_ctrl_req_complete()**.

6.29.7.5.2 7.5.2. Key Points

- RX/TX data path is hardware/offload driven.
- CVQ is the main application-visible I/O request path.
- MQ expansion avoids blocking the host by completing the CVQ command before long VQ startup.
- RSS/hash CVQ commands are not implemented by the app.
- The app's most important responsibility is correct lifecycle sequencing, not packet processing.

6.29.7.6 7.6. Device Lifecycle Management

6.29.7.6.1 7.6.1. Device Lifecycle State Machine

Endpoint-level conceptual state:

```
Endpoint: INIT -> MODEL_CREATED -> ENDPOINT_CREATED -> FEATURES_OK_PREP -> DRIVER_OK_ENABLE -> OPERATIONAL ->
RESETTING -> CLEANED_FOR_REBIND -> (rebind) | DESTROYED
```

Hotplug state:

```
Hotplug : ABSENT --plug--> PRESENT(link down) --Power-ON--> DLActive --> OPERATIONAL
             ^                                                             | unplug
          DESTROYED <- delayed destroy <- Power-OFF (or 30s timeout) <- UNPLUG_PENDING
```

6.29.7.6.2 7.6.2. Hotplug Management Flow

Plug:

1. Runtime command **plug <idx>** is parsed.
2. Workqueue receives hotplug work.
3. The endpoint representor, TLP device, and VNet controller are created.
4. DSP slot status sets attention button, presence detect changed, and presence detect state.
5. MSI is sent if ACG credit and host MSI setup are available.
6. Host **pciehp** notices the slot event.
7. Host powers on the slot.
8. TLP handler observes Slot Control Power ON and sets **DLActive**.
9. Host enumerates and probes the VirtIO-net endpoint.

Unplug:

1. Runtime command **unplug <idx>** is parsed.
2. Workqueue marks endpoint **pending_unplug**.
3. DSP slot status raises attention button.
4. MSI is sent or retried.

5. Host **pciehp** unbinds the driver and powers off the slot.
6. TLP handler observes Slot Control Power OFF.
7. Delayed destroy work is submitted.
8. Controller, TLP device, and representor are destroyed.
9. Pending flags are cleared.

If the host never powers off the slot, the main loop forces removal after **UNPLUG_TIMEOUT_SEC**, currently 30 seconds.

6.29.7.6.3 7.6.3. Key Points

- Hotplug creation is intentionally done on the workqueue because endpoint creation can take hundreds of milliseconds.
- PE1 must keep progressing TLPs during hotplug.
- Plug and unplug use PCIe hotplug slot semantics rather than ad hoc host signaling.
- Unplug waits for host power-off before destroying resources.
- A timeout safety net handles failed or stuck host removal.

6.29.7.7 7.7. SR-IOV

```
app init (per PF): pool_alloc_range -> create_pf_rep_ex(TotalVFs, stride=1, offset=1, vf_msix, vf_db)
                   -> PF tlp_dev -> SR-IOV cap shadow -> discover & open VF reps (templates only)

host: echo N > sriov_numvfs -> VF-Enable(VFE) 0->1 -> vfe_fan_out: per slot vnet_sriov_start_vf_slot (VF tlp_dev)

guest VF: FEATURES_OK -> vf_engine_create+start + vf_vqs_create + vf_io_context
          DRIVER_OK   -> vf_start_and_enable (start VQs, bind CVQ user_data=slot, enable engine)

per-VF FLR (DevCtl bit15): handle_vf_flr (coalesce) -> reset_vf_slot (tlp_dev stop->start; rep stays)

host: echo 0 -> VFE 1->0 -> stop_vf_slot (engine destroy + tlp_dev stop)
```

Limits: **--total-vfs ≤ 64/PF** (ARI ceiling 255); device-wide VF-VQ budget validated at init; VF cfg BARs read 0.
Mutually exclusive with LU.
Pass a VF to a VM via **vfio-pci** (**1af4:1041**) + QEMU/libvirt.

SR-IOV Support — one emulated PF exposes up to 64 independent virtio-net VFs (own cfg space, MSI-X, doorbells, engine, SF datapath); host-driven or VM-passthrough.
Off by default; --total-vfs <1..64>.
Split: vnet_sriov.c (pool/rep/tlp_dev), vnet_pci_device.c (cap + VFE fan-out + per-VF TLP/FLR), vnet_pci_dev_core.c (per-VF datapath).

6.29.7.7.1 7.7.1. 1 Model & rationale

SF-per-VF (must OVS-bridge each VF rep for RX); ARI functions 1..N (FirstVFOffset=1, VFStride=1), VF Device ID 0x1041; two-stage lazy bring-up (VFE creates tlp_dev only; engine deferred to guest FEATURES_OK, enabled at DRIVER_OK); deterministic per-VF MAC (base_low16 + num_ep + vf_index).

6.29.7.7.2 7.7.2. 2 Resources & validation

per VF: 1 GVMI, 2q+2 MSI-X/doorbells, 1 SF+rep, VNET_TOTAL_VQS(q)=2q+1 VQs. Table of vnet_sriov_validate_caps() checks → exact rejection strings (per-PF 64, ARI 255, device-wide budget 4352, aggregate 256, SDK pool/stride/msix). Fallback to non-rep_ex when pool_size==0.

6.29.7.7.3 7.7.3. 3 VF lifecycle

6-phase diagram (A init → B host VFE fan-out → C guest FEATURES_OK/DRIVER_OK → D per-VF FLR → E host disable → F exit), per-VF sub-state machine, a guard-atomics table (active, flr_in_progress, engine_started, vqs_created, engine_enabled, io_ctx_started/cvq_bound, num_active_qps), and the subtle fan-out semantics (VFE starts all TotalVFs; NumVFs write is shadow-only).

6.29.7.7.4 7.7.4. Per-VF FLR

A Function Level Reset resets a single VF in isolation, leaving its siblings and the parent PF running. The guest sets the Initiate-FLR bit in the VF's PCIe Device Control register; the TLP thread then runs **vnet_pci_device_handle_vf_flr()**, which:

1. atomically sets **flr_in_progress**, coalescing a duplicate FLR that arrives in the window (logged **FLR coalesced**);
2. resets the VF's config-space shadow synchronously, so the host reads a clean, post-reset config space immediately; and
3. queues **VF_FLR_RESET**, whose worker runs **vnet_sriov_reset_vf_slot()** = **tlp_dev** stop then start (the representor stays open), after which the guest re-probes through Phase C.

The worker always clears **flr_in_progress** on completion, even on error, so a VF never gets stuck coalescing. A PF-level FLR is handled separately by the PF's own virtio reset state machine.

6.29.7.7.5 7.7.5. SR-IOV PCIe Capability (Host View)

The PF presents a standard PCIe SR-IOV Extended Capability at config-space offset **0x100**, chained **0x100** (SR-IOV) → **0x140** (ARI) → **0x148** (ACS); ARI enables VF function numbers above 7, and ACS gives per-VF IOMMU isolation for passthrough. Registers are synthesized on demand from **struct vnet_sriov_pf_state** (**sriov_cap_read**) and host writes are decoded back into it (**sriov_cap_decode_write**); the layout follows PCIe Base Spec (**struct sriov_ext_cap_layout**).

DW	Field(s)	Value / behavior
0	Capability header	SR-IOV cap ID, version, next-cap offset → ARI (0x140)
1	SR-IOV Capabilities	ARI Capable Hierarchy Preserved (bit 1); VF Migration not supported
2	Control | Status	Control RW bits VFE (0x1), VFM (0x2), VF-MigInt (0x4), VF-MSE (0x8), ARI (0x10); Status VF-Migration is W1C (bit 0), rest RO
3	InitialVFs | TotalVFs	both = total_vf

DW	Field(s)	Value / behavior
4	NumVFs | FunctionDependencyLink	NumVFs host-written, clamped to [0, total_vf]
5	FirstVFOffset | VFStride	1 | 1
6	VF Device ID	0x1041
7	Supported Page Sizes	0x553 = {4 KB, 8 KB, 64 KB, 256 KB, 1 MB, 4 MB}
8	System Page Size	host-written (RW), validated against DW7
9..14	VF BARs	64-bit prefetchable aperture for the VF MMIO windows
15	VF Migration State Array Offset	0 (no migration)

Setting VF-Enable triggers the fan-out described above.
A VF's own Type-0 BARs read 0 by design: a VF addresses its MMIO through the SR-IOV VF BAR aperture (DW9..14), as the specification requires.

6.29.7.7.6 7.7.6. VF datapath & VM passthrough

Because each VF has its own SF representor, its data-path must be bridged on the DPU before it carries traffic.
Add each VF representor to the OVS bridge (the per-VF engine-create log prints the **ifindex** to use):

```
ovs-vsctl add-port ovsbr1 <vf_representor>
```

To pass a VF through to a VM, bind it on the host to **vfio-pci** (IOMMU enabled) and attach it to the guest; the VF enumerates as PCI ID **1af4:1041**:

```
QEMU:    -device vfio-pci,host=<VF BDF>
libvirt: <hostdev mode='subsystem' type='pci' managed='yes'> ... </hostdev>
```

Traffic path: **guest -> VF -> DPU SF/representor -> OVS -> uplink**.

6.29.7.7.7 7.7.7. Constraints & expected logs

- Stating that the host must set **sriov_numvfs = 0** (and shut down any VM holding a VF) before rebooting, and that rebooting with VFE set leaves stale VF state on the DPU that can disrupt the next enumeration.
- Mutually exclusive with **--lu-mode**: live update transfers PF state only, so the app refuses to start with both enabled.
- Each VF's queue count mirrors the PF **--max-queue-pairs**.
- A PF cannot be hot-unplugged while VF-Enable is set or any VF datapath is live; the unplug is rejected (**... (VFE=1, K live VF datapath(s)); disable it on the host first**).

A healthy single-PF, single-VF bring-up produces this log sequence (match on text, not line numbers):

```
SR-IOV: dynamic SR-IOV supported (max_total_vfs=.., pool_size=..)
SR-IOV: pool reserved range [0, N)
SR-IOV: PF representor created with VF range [0, N) (vf_num_msix=18, vf_num_db=18)
SR-IOV: opened all N VF reps for this PF
SR-IOV VFE: 0 -> 1 (NumVFs=N, TotalVFs=N)
SR-IOV: VF[vf_index=g] tlp_dev started
SR-IOV VFE 0->1 fan-out: N/N slots started (0 failed)
SR-IOV: VF[slot=s, vf_index=g] FEATURES_OK - queueing engine bring-up (mq=1)
SR-IOV: VF[vf_index=g] offload engine created (idle, 17 VQs, MTU=1500)
SR-IOV: VF[vf_index=g] engine representor: vhca_id=0x.... ifindex=.. (add to OVS bridge for RX)
SR-IOV: VF[vf_index=g] created 8 queue pair(s) + CVQ
SR-IOV: VF[vf_index=g] CVQ IO context started
SR-IOV: VF[slot=s, vf_index=g] DRIVER_OK - queueing engine enable
SR-IOV: VF[vf_index=g] datapath operational (engine enabled, 8 queue pair(s))
```

6.29.7.8 7.8. Cleanup and Shutdown

6.29.7.8.1 7.8.1. Cleanup and Shutdown Flow

Shutdown begins when **SIGINT** or **SIGTERM** sets **force_quit**.

The main progress loop exits when:

- Quit is requested and no hot-unplug is pending.
- Hot-unplug pending has completed.
- Hot-unplug pending times out.
- LU handover is requested by **SIGUSR1**.

6.29.7.8.2 7.8.2. Phase 1 – Worker and Device Cleanup

The current vnet app does not have vblk-style per-IO worker threads named like **vblk_io_ctx_thread()**. Its equivalent first phase is workqueue and controller-side cleanup.

Steps:

1. Disable worker-side diagnostics.
2. Shut down PCI config workqueue.
3. Join outstanding MQ start threads.
4. Stop accepting new work.
5. Destroy VirtIO device models.
6. For each endpoint, destroy controller resources.
7. Disable VQs while engine is enabled.
8. Disable engine.
9. Stop VQs.
10. Destroy VQs.
11. Stop and destroy CVQ IO context.
12. Stop offload engine.
13. Destroy offload engine when required.
14. Destroy TLP devices and representors.
15. Clear endpoint present state.

6.29.7.8.3 7.8.3. Phase 2 – Reverse-Order Resource Cleanup

After workqueue shutdown:

1. Clear worker PE references from the workqueue.
2. Destroy per-controller worker PEs.
3. Flush pending ACG credits.
4. Stop the TLP channel context.
5. Drain PE1 until the TLP channel reaches idle or timeout.
6. Destroy the TLP channel.
7. Stop and destroy the PCI type.
8. Tear down DOCA VNet.
9. Remove the DOCA device from VNet.
10. Destroy PE1.
11. Free transaction regions, BDF maps, topology arrays, controller arrays, and VirtIO device arrays.
12. Close LU connections if any.
13. Destroy ARGP resources in **main()**.

6.29.7.8.4 7.8.4. Key Points

- Cleanup order follows ownership dependencies.
- Workqueue shutdown happens before freeing objects referenced by work items.
- Worker PEs are destroyed only after the workqueue exits.
- TLP channel is stopped and drained before destruction.
- LU cleanup differs because the active process transfers traffic ownership before releasing local handles.

6.29.7.9 7.9. Live Upgrade (LU) Mode

The vnet refapp implements live update in the same binary through **--lu-mode**.

```
A(active) <- B(standby) connects -> A exports state + cmd FD -> B reconstructs device/EP/engines
from export blobs, replays VQ cfg (engines IDLE) -> B DEV_READY -> A DEV_GO -> B enables engines
-> B DEV_ACK -> TLP channel handover -> B active (chainable)
```

Modes:

- **none**: normal operation.
- **active**: current traffic owner.
- **standby**: process that restores state from active and takes over.

LU uses:

- Unix domain socket: **/tmp/vnet_live_update.sock**.
- Lock file: **/tmp/vnet_live_update.lock**.
- Shared memory: **/vnet_live_update**.
- TLP channel shared memory directory: **/dev/shm/vnet_lu_channel**.
- **SIGUSR1** to trigger active handover.

High-level LU flow:

1. Active process runs and serves traffic.
2. Standby process connects to active.
3. Active sends command FD and exports device state.
4. Standby opens shared memory and reconstructs DOCA device state.
5. Standby recreates endpoint/controller/offload objects from export blobs.
6. Standby replays VirtIO config and VQ state without enabling engines.
7. Standby signals **DEV_READY**.
8. Active sends final **DEV_GO**.
9. Standby enables engines, one or more in parallel.
10. Standby sends **DEV_ACK**.
11. Channel LU transfers the TLP channel.
12. Standby becomes active and can support another chained LU.

Static LU state includes:

- endpoint VHCA ID
- MAC/MTU
- device features
- max queue pairs
- active queue pairs
- queue size
- MQ negotiation state
- per-VQ descriptor/driver/device addresses
- MSI-X vectors
- offload engine export blobs
- TLP channel configuration

Hotplug LU has additions:

- Number of populated slots
- Slot-presence bitmap
- Base MAC used to reconstruct per-slot MAC addresses

6.29.7.10 7.10. Crash Recovery

The current vnet refapp has live update state transfer but does not implement a separate persistent crash recovery mechanism comparable to a full crash-resume feature.

Important distinction:

- LU is coordinated and stateful: active and standby cooperate.
- Crash recovery would require reconstructing after unexpected termination without a live handover.
- The current code has robust cleanup, reset handling, timeout handling, and LU import/export paths, but no standalone crash recovery workflow is exposed as an application mode.

6.29.8 8. Virtio-net Operations Support Summary

Supported by the refapp/offload stack:

- RX/TX packet queues: supported through DOCA DevEmu VNet offload after VQs are configured and engine is enabled.
- Control virtqueue: supported when **VIRTIO_NET_F_CTRL_VQ** and **VIRTIO_NET_F_MQ** are negotiated.
- Multi-queue pair set: implemented for **VIRTIO_NET_CTRL_MQ_VQ_PAIRS_SET**.
- Runtime link speed update: supported through app command **speed <idx> <Mbps>** and config-change MSI-X.
- Hotplug/unplug: supported through PCIe hotplug slot emulation.

Partially supported or backend-acknowledged:

- MAC control class: acknowledged as backend-handled in the app callback.
- RX control class: acknowledged as backend-handled in the app callback.
- VLAN control class: acknowledged as backend-handled in the app callback.

Not implemented in the app callback:

- RSS configuration command.
- Hash configuration command.
- Unknown MQ commands.
- Unknown control classes.

6.29.9 9. Virtio Feature Bits Support Summary

Advertised by default:

- **VIRTIO_NET_F_MAC**: device reports MAC address.
- **VIRTIO_NET_F_STATUS**: link status is available.
- **VIRTIO_NET_F_CSUM**: checksum offload feature advertised.
- **VIRTIO_NET_F_MTU**: MTU is reported.
- **VIRTIO_NET_F_SPEED_DUPLEX**: speed and duplex are reported.
- **VIRTIO_NET_F_HOST_TSO4**: host can send TSOv4 packets to device.
- **VIRTIO_NET_F_HOST_TSO6**: host can send TSOv6 packets to device.

Always added by device initialization:

- **VIRTIO_F_VERSION_1**: modern VirtIO compliance.
- **VIRTIO_F_ACCESS_PLATFORM**: platform/IOMMU access semantics.
- **VIRTIO_NET_F_CTRL_VQ**: control virtqueue available.
- **VIRTIO_NET_F_MQ**: multi-queue support.

Defined but not part of the default advertised feature set unless code (SDK) is extended:
(some of the features are offered to enabled using CLI option)

- **VIRTIO_NET_F_GUEST_CSUM**
- **VIRTIO_NET_F_CTRL_GUEST_OFFLOADS**

- `VIRTIO_NET_F_GUEST_TSO4`
- `VIRTIO_NET_F_GUEST_TSO6`
- `VIRTIO_NET_F_GUEST_ECN`
- `VIRTIO_NET_F_GUEST_UFO`
- `VIRTIO_NET_F_HOST_ECN`
- `VIRTIO_NET_F_HOST_UFO`
- `VIRTIO_NET_F_MRG_RXBUF`
- `VIRTIO_NET_F_CTRL_RX`
- `VIRTIO_NET_F_CTRL_VLAN`

6.29.10 10. Developer Notes

The main engineering theme in this refapp is latency isolation. PE1 must continuously progress PCI TLP completions, so any operation that can block on firmware or DOCA state transitions is moved to the PCI config workqueue or detached MQ-start threads.

The second key theme is reset safety. The app uses reset generations, cancellation flags, VQ shadow snapshots, and short bounded waits so stale worker operations cannot corrupt a newly reset device.

The third key theme is host compatibility. Hotplug follows PCIe slot semantics, VirtIO status transitions follow the expected driver lifecycle, and CVQ MQ commands are completed quickly to avoid Linux probe and soft-lockup issues.

6.30 DOCA Virtio-blk Emulation Application Guide

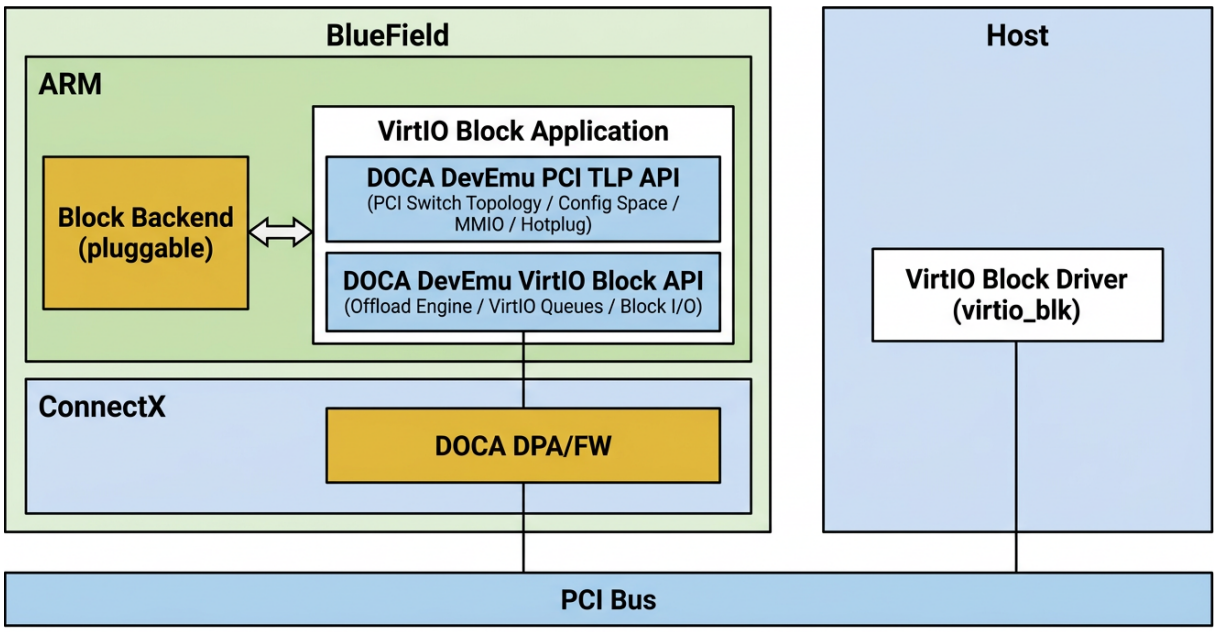
This guide describes the DOCA Virtio-blk Emulation Application, which emulates virtio-blk PCIe functions on NVIDIA® BlueField® DPUs. The application leverages DOCA DevEmu PCI TLP APIs to construct a full software-defined PCI switch topology, presenting one or more standard virtio-blk devices to the host. The data path is hardware-accelerated on the DPU's DPA (Data Path Accelerator) or DPU Arm cores.

> ⚠ This feature is currently supported at alpha level.

6.30.1 Introduction

The DOCA Virtio Block (**`virtio-blk`**) Emulation Application demonstrates how to use the DOCA DevEmu PCI TLP and DOCA DevEmu Virtio-blk libraries to emulate virtio-blk PCIe functions with full PCIe configuration space emulation, multi-queue support, hotplug capabilities, and DMA-based data transfers. The emulated block devices appear as standard **`virtio-blk`** devices to the host operating system.

6.30.1.1 System Design



6.30.1.1.1 Host Side (Virtio-blk Client Driver)

- Runs a standard `virtio_blk` client driver in the host operating system.
- Host applications issue block I/O operations (e.g., read, write, flush, get device ID).
- The `virtio_blk` driver translates these into virtio-blk protocol I/O requests submitted through virtio queues.

6.30.1.1.2 BlueField DPU (DOCA Virtio-blk Emulation Application)

- Runs the DOCA Virtio-blk Emulation Application.
- Constructs a software PCI switch topology (USP + DSPs + Endpoints) visible to the host.
- Handles PCIe TLP (Transaction Layer Packets) for configuration space emulation and MMIO processing, and configures doorbell and MSI-X regions for hardware-accelerated notification.
- Decodes virtio-blk I/O requests (`VBLK_T_IN` , `VBLK_T_OUT` , `VBLK_T_GET_ID` , etc.) from the host via virtio queues.
- Performs DMA data transfers between host and DPU memory using `doca_dma` memcpy operations.

6.30.1.1.3 Data Path Provider (DPA or DPU)

The application supports two data path providers:

- DPA – Offloads virtio queue processing to the DPU's Data Path Accelerator for hardware-accelerated, low-latency operation.

- DPU (default) – Processes virtio queues on the DPU Arm cores for maximum flexibility and debugging.

6.30.1.2 End-to-End Flow

1. Host application initiates a block I/O operation (e.g., disk read/write).
2. The `virtio_blk` driver submits the request through a virtio queue.
3. The DPU's offload engine detects the new request and dispatches it to an IO context thread.
4. The IO context thread processes the request, performing DMA data transfers as needed.
5. Upon completion, the DPU sends the response back to the host through the virtio queue.

This architecture offloads block device emulation from the host to the DPU, enabling hardware-accelerated storage device virtualization without requiring physical block devices on the host.

6.30.2 Application Architecture

6.30.2.1 Core Components

6.30.2.1.1 Virtio-blk Device Model

Virtio-blk is a para-virtualized block storage device defined in the Virtio Specification. It enables high-performance block device emulation using the virtio framework. Key parameters include:

- `capacity` – Device capacity in 512-byte sectors.
- `seg_max` – Maximum number of scatter-gather segments per request.
- `num_queues` – Number of virtio queues for parallel I/O processing.
- `size_max` – Maximum size of any single segment.

6.30.2.1.2 Virtio Queues

Virtio-blk uses one or more request queues to communicate between host and device:

- Request queues – The host submits block I/O requests (read, write, flush, get ID, etc.) through descriptor chains.
- Multi-queue support – Up to 255 queues can be configured for parallel I/O processing across multiple CPU cores.
- Queue lifecycle – Each queue transitions through states: `DESTROYED → STARTING → BINDING → RUNNING → STOPPING → UNBINDING → DESTROYED`.

6.30.2.1.3 PCIe TLP Emulation

Unlike higher-level DevEmu APIs, this application operates at the PCIe Transaction Layer Packet (TLP) level:

- Configuration space emulation – The application intercepts and responds to PCIe Configuration Read/Write TLPs, emulating the full PCI configuration space for each device in the topology.

- MMIO emulation – Memory-mapped I/O requests (BAR accesses) are handled to emulate virtio PCI common configuration, device configuration, ISR, and MSI-X tables. Doorbell regions are configured in the BAR layout but processed by hardware/DPA at runtime.
- Software PCI switch – A complete PCI switch topology is constructed in software, consisting of an Upstream Port (USP), multiple Downstream Ports (DSPs), and Endpoints (EPs).

6.30.2.1.4 PCI Switch Topology

The application constructs the following topology:

```
USP (idx=0)
      DSP[0] (idx=1)     EP[0]  (virtio-blk device)
      DSP[1] (idx=2)     EP[1]  (virtio-blk device)
      ...
      DSP[N] (idx=N+1)   EP[N]  (virtio-blk device)
```

Component	Description
USP	Upstream Switch Port, the root of the PCI switch.
DSP	Downstream Switch Port, one per endpoint, supporting hotplug capabilities.
EP	Virtio-blk endpoint, appearing as a **virtio-blk** PCI device to the host.
DUMMY	A catch-all device for unmatched BDF routing. When the host probes a BDF that does not map to any USP/DSP/EP, the TLP handler returns Unsupported Request (UR) via this device, which is standard PCIe enumeration behavior for non-existent devices.

6.30.2.2 Initialization Process

The virtio-blk initialization process involves:

1. Creating the PCI type with **doca_devemu_vblk_pci_tlp_type_create()**.
2. Configuring PCI capabilities (Express, MSI-X) for endpoints; VPD and PM are configured for bridge devices.
3. Setting up the TLP channel for PCIe transaction handling.
4. Constructing the PCI switch topology with USP, DSPs, and EPs.
5. Creating virtio-blk offload engines and binding them to endpoints.
6. Starting worker threads for TLP processing and I/O handling.

6.30.2.3 Reset and Shutdown

Proper reset and shutdown procedures ensure device integrity and resource cleanup:

Procedure	Description
Virtio reset	Triggered by the host writing **device_status = 0**. The application tears down all active virtio queues, waits for in-flight operations to complete, and reinitializes the device.

Procedure	Description
Shutdown	Gracefully terminates all threads, stops all contexts, destroys offload engines, and releases hardware resources.

6.30.3 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA DevEmu PCI TLP – PCIe TLP-level device emulation
- DOCA DevEmu Virtio-blk – virtio Block offload engine, I/O contexts, and request handling
- DOCA DMA – DMA memcpy for host-DPU data transfers
- DOCA Arg Parser – Command-line argument parsing

For additional information about the used DOCA libraries, please refer to the respective programming guides.

6.30.4 Dependencies

The following software components are required to build and run the DOCA Virtio-blk Emulation Application:

- BlueField-3 DPU
- DOCA SDK with DevEmu, DMA, and DPA support
- Huge page allocation configured on the DPU

> In some setups, the following dependencies may be missing and must be installed before compiling the application:
>
> - `libjson-c-dev`
> - `libjsoncpp-dev`

6.30.5 Compiling the Application

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are distributed with their source code and build instructions. This allows you to compile the applications as-is or modify the source code and rebuild custom versions.

> For more information on DOCA applications, development workflows, and build guidance, see the DOCA Reference Applications documentation page.

The source code for the application is located at:

```
/opt/mellanox/doca/applications/vblk_pci_dev/
```

6.30.5.1 Compiling All Applications

All DOCA reference applications are part of a unified meson project. By default, building the project compiles all available applications.

To build all applications:

```
cd /opt/mellanox/doca/applications/
meson setup /tmp/build
ninja -C /tmp/build
```

doca_vblk_pci_dev is created under **/tmp/build/vblk_pci_dev/**.

6.30.5.2 Compiling Only the Current Application

To build only the virtio-blk application, use the following command to disable all other applications:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_vblk_pci_dev=true
ninja -C /tmp/build
```

doca_vblk_pci_dev is created under **/tmp/build/vblk_pci_dev/**.

Alternatively, you can configure build flags directly in the **meson_options.txt** file rather than passing them as command-line arguments:

1. Edit the following file **/opt/mellanox/doca/applications/meson_options.txt**.
2. Modify the flags:
   - Set **enable_all_applications** to **false**
   - Set **enable_vblk_pci_dev** to **true**
3. Run the standard build commands:

   ```
   cd /opt/mellanox/doca/applications/
   meson /tmp/build
   ninja -C /tmp/build
   ```

6.30.5.3 Build Outputs

The build produces the following executables:

Executable	Description
doca_vblk_pci_dev	Single-process virtio-blk device (TLP + I/O in one process)
doca_vblk_pci_dev_lu	Live Upgrade TLP manager process (manages TLP and spawns EMU)

Executable	Description
doca_vblk_pci_dev_emu	Live Upgrade EMU worker process (offload engine and I/O)

6.30.6 Running the Application

6.30.6.1 Prerequisites

1. Enable the firmware configuration with virtio-blk emulation PF. A cold reboot is required after updating the firmware configuration:

```
[dpu]
mlxconfig -y -d /dev/mst/mt41692_pciconf0 reset
mlxconfig -y -d /dev/mst/mt41692_pciconf0 set TLP_EMULATION_NUM_PF=32
mlxconfig -y -d /dev/mst/mt41692_pciconf0 set TLP_EMULATION_ENABLE=1
mlxconfig -y -d /dev/mst/mt41692_pciconf0 set PCI_SWITCH_EMULATION_ENABLE=1
mlxconfig -y -d /dev/mst/mt41692_pciconf0 set PCI_SWITCH_EMULATION_NUM_TLP_PORT=1
mlxconfig -y -d /dev/mst/mt41692_pciconf0 set PCI_SWITCH_EMULATION_NUM_PORT=2
```

2. This application requires huge page allocation. Ensure huge pages are configured on the DPU:

```
echo 3072 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

6.30.6.2 Application Execution

This application is provided in source form; therefore, compilation is required before execution. Refer to the "Compiling the Application" section for build instructions.

6.30.6.2.1 Application Usage

```
Usage: doca_vblk_pci_dev [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version information
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  -d, --emulation-manager <mlx5 device name> mlx5 device that manages tlp emulation devices
  -q, --num-queues <1-255>          Number of virtio queues per endpoint, must be >= IO context count (default:
255)
  --io-ctx-mask <mask>              IO contexts CPU mask, up to 15 cores; TLP core must not be included (default:
0x00FF = cores 0-7)
  --tlp-core-idx <core>             TLP core, must not be in io-ctx-mask (default: 15)
  --offload-engine-core-idx <core>  Offload engine core, must be in io-ctx-mask (default: 0)
  --stats-ios-period <ios-period>   Stats are queried after every <ios-period> requests are handled. The default
 value of 0 means stats are not queried
  --seg-max <seg_max>               seg_max PCI register value, should be [1, max{queue_size - 2, 128}] (default:
126)
  --indirect                        enable use indirect descriptor feature, default: disabled
  -n, --num-ep <1-32>               Number of endpoints to create (1-32, default: 1), some host BIOS does not
support large number of endpoints. Suggest to use number <= 20
  -H, --hotplug-mode <0|1>          Set hotplug mode: 1 for hotplug mode (default), 0 for static mode. In hotplug
mode, enter commands to control devices:
  plug <DSP_IDX>   - Plug device to DSP slot
  unplug <DSP_IDX> - Unplug device from DSP slot
Example: plug 0
  --total-vfs                       SR-IOV TotalVFs per PF (1..7, no ARI). Any value >= 1 enables SR-IOV; omit to
disable. Per-VF MSI-X/doorbells are derived from the queue count
  --provider <datapath provider name> Set data path provider, {DPA or DPU, default: DPU}
```

Use the **-h** or **--help** flag to print this usage information at runtime:

```
./doca_vblk_pci_dev -h
```

For additional information, refer to the "Command Line Flags" section.

6.30.6.2.2 Example Command Line Execution

Run the application on BlueField with default settings (8 IO cores, TLP on core 15, hotplug mode):

```
./doca_vblk_pci_dev -d mlx5_0
```

Run the application with enabling indirect_desc feature

```
./doca_vblk_pci_dev -d mlx5_0 --indirect
```

Run the application with querying stats counter (such as host and device avail_idx & used_idx. The example code querying the stats in IO path which hit the performance)

```
./doca_vblk_pci_dev -d mlx5_0 --indirect --stats-ios-period 1024
```

Run with custom configuration (4 endpoints, 64 queues per endpoint, static mode):

```
./doca_vblk_pci_dev -d mlx5_0 -n 4 -q 64 -H 0 --io-ctx-mask 0x0f --tlp-core-idx 8
```

Run with DPU data path provider:

```
./doca_vblk_pci_dev -d mlx5_0 --provider DPU
```

Run with SRIOV enabled

```
./doca_vblk_pci_dev -d mlx5_0 -n 1 --total-vfs 2
```

6.30.6.2.3 Runtime Commands

The application accepts interactive commands via stdin during operation:

Command	Description	Availability
cap <GB>	Set block device capacity in GB (e.g., **cap 1** sets capacity to 1 GB)	All modes
plug <DSP_IDX>	Plug a virtio-blk device into the specified DSP slot	Hotplug mode only
unplug <DSP_IDX>	Unplug the virtio-blk device from the specified DSP slot	Hotplug mode only

6.30.6.2.4 Verifying the Device (Host Side)

After staring the application, then boot up the host, the virtio-blk driver will probe the device on the host, the virtio block device should appear:

```
# Verify the device is visible
lspci | grep -i virtio

# Check block device
lsblk
```

6.30.6.3 Command Line Flags

6.30.6.3.1 General Flags

Short Flag	Long Flag	Description
`-h`	`--help`	Prints a help synopsis and exits
`-v`	`--version`	Prints program version information and exits
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.30.6.3.2 Program Flags

Short Flag	Long Flag / JSON Key	Description	Default
`d`	`emulation-manager`	Mandatory: mlx5 device that manages TLP emulation devices	–
`q`	`num-queues`	Number of virtio queues per endpoint [1–255]. Must be ≥ the number of IO contexts.	255

Short Flag	Long Flag / JSON Key	Description	Default
`n`	`num-ep`	Number of endpoints to create [1–32]. Some host BIOS may not support large numbers; recommended ≤ 20.	1
`H`	`hotplug-mode`	Hotplug mode: 1 = hotplug (dynamic plug/unplug via stdin), 0 = static (all EPs created at startup).	1 (hotplug)
N/A	`io-ctx-mask`	IO contexts CPU mask, up to 15 cores. The TLP core must not be included.	0x00FF (cores 0–7)
N/A	`tlp-core-idx`	Dedicated CPU core for TLP processing. Must not be in `io-ctx-mask`.	15
N/A	`offload-engine-core-idx`	CPU core for the offload engine. Must be included in `io-ctx-mask`.	0
N/A	`seg-max`	`seg_max` PCI register value, should be [1, max{queue_size - 2, 128}]. Controls the maximum number of scatter-gather segments per request.	126
N/A	`indirect`	Enable the virtio indirect descriptor feature (`VIRTIO_F_INDIRECT_DESC`).	disabled
N/A	`stats-ios-period`	Stats are queried after every N requests. 0 = disabled.	0 (disabled)
N/A	`provider`	Data path provider: `DPA` (hardware-accelerated) or `DPU` (Arm cores).	DPU
N/A	total-vfs	SR-IOV TotalVFs per PF (1..7, no ARI). Any value >= 1 enables SR-IOV; omit to disable	0

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

The default `num-queues` is 255, which is designed for the default single endpoint (`num-ep=1`). When increasing the number of endpoints, reduce `num-queues` accordingly (e.g., `-q 8`) to decrease memory usage, as each queue consumes DMA task pool entries and offload engine resources per endpoint.

6.30.6.4 Troubleshooting

Please refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue you may encounter with the installation or execution of the DOCA applications.

Known issues:
It doesn't support fast frequently bind/ubind operation due to known linux kernel issue which could result in kernel task hung on old kernel.
On old kernel, there's ABBA deadlock issue between the unbind/ `del_gendisk` path and a concurrent sysfs read from udev. It's fixed in upstream linux kernel 6.0 commit 6f8191fd.

6.30.7 Application Code Flow

This section provides a comprehensive guide to implementing a virtio-blk device emulation application using the DOCA library. The DOCA Virtio-blk Emulation Reference Application demonstrates how to leverage hardware acceleration on a Data Processing Unit (DPU) to emulate fully functional block storage devices, appearing to the host as standard **virtio-blk** PCIe devices.

The implementation provides hardware-accelerated block device emulation, utilizing DPU resources for high-performance I/O processing. The multi-threaded architecture, with explicit CPU core affinity, ensures optimal resource utilization and minimizes contention between worker threads. Asynchronous processing throughout the pipeline maintains high throughput by avoiding blocking operations that could degrade system performance.

The design supports multiple endpoints (up to 32), each with up to 255 virtio queues, enabling massive parallel I/O processing. Both static and hotplug deployment modes are supported: static mode creates all endpoints at startup, while hotplug mode allows dynamic addition and removal of endpoints at runtime.

This modular approach allows developers to understand each phase of the application flow independently while appreciating how they integrate into a complete virtio-blk emulation solution using DOCA libraries and hardware acceleration.

6.30.7.1 Framework Initialization

This section outlines the essential APIs and data structures required to initialize the DOCA virtio-blk emulation framework before creating and managing block devices.

6.30.7.1.1 Initialization Flow

1. Create logging backends
    - Create the application log backend using **doca_log_backend_create_standard()**, which handles logs produced by the application via **DOCA_LOG_*** macros.
    - Create a separate SDK internal log backend using **doca_log_backend_create_with_file_sdk(stderr, &sdk_log)**, which captures logs produced inside the DOCA SDK libraries.
    - Set the SDK log level to WARNING using **doca_log_backend_set_sdk_level()** to suppress verbose SDK internals.
2. Parse application arguments
    - Initialize the argument parser framework using **doca_argp_init()**.
    - Register application-specific parameters via **register_vblk_pci_dev_params()**.
    - Store parsed values in **struct vblk_pci_dev_config**.
3. Start argument parsing
    - Parse command-line arguments using **doca_argp_start()**.
    - Validate arguments and populate the configuration structure with parsed values.
4. Set up signal handlers
    - Register **SIGINT** and **SIGTERM** handlers to set the global **force_quit** flag for graceful shutdown.
5. Validate configuration

- Verify that `num_queues` is within [1, 255] and is ≥ the number of IO contexts.
- Ensure `tlp_core_idx` is not included in `io_ctx_mask`.
- Ensure `offload_engine_core_idx` is included in `io_ctx_mask`.
- Build the `io_ctx_cores` array from the CPU mask.

6. Set data path provider
   - Configure the data path provider (DPA or DPU) using `doca_devemu_vblk_set_datapath_on_dpa()`.
7. Initialize virtio-blk subsystem
   - Open the DOCA device by IB device name using `open_doca_device_with_ibdev_name()`.
   - Add the device for virtio-blk emulation using `doca_devemu_vblk_add_dev()`.
   - Set per-request user data size using `doca_devemu_vblk_set_vblk_req_user_data_size()`, sized to `struct vblk_io_request`.
   - Configure `seg_max` using `vblk_ctrl_manager_set_max_seg_max()`.
   - If `--indirect` is enabled, verify device capability with `doca_devemu_vblk_cap_is_indir_descs_supported()`.
   - Initialize the virtio-blk framework using `doca_devemu_vblk_init()`.

6.30.7.1.2 Key Points

- Configuration validation – `vblk_app_cfg_validate()` enforces several constraints before the application proceeds:
  - `num_queues` must be in [1, 255] and must be ≥ the number of IO context cores (derived from `io_ctx_mask`).
  - `tlp_core_idx` must not overlap with `io_ctx_mask` – the TLP thread and IO threads run on separate cores.
  - `offload_engine_core_idx` must be included in `io_ctx_mask` – the offload engine shares a core with one IO context.
  - `num_ep` must not exceed `MAX_NUM_EP` (32).
- Capability check before init – `vblk_ctrl_manager_set_max_seg_max()` queries the device's hardware-supported `seg_max` via `doca_devemu_vblk_cap_get_max_seg_max()` and fails if the configured value exceeds hardware capability. Similarly, indirect descriptor support is verified via `doca_devemu_vblk_cap_is_indir_descs_supported()` before enabling the feature.
- Error recovery in `vblk_init()` – If any step after `doca_devemu_vblk_add_dev()` fails, the device is removed via `doca_devemu_vblk_rm_dev()` before returning the error, preventing resource leaks.
- Default configuration – The application ships with sensible defaults: 8 IO cores (mask `0x00FF`, cores 0–7), TLP on core 15, offload engine on core 0, 255 queues/EP, 1 EP, hotplug enabled, DPA datapath. These defaults target a BlueField-3 with 16 Arm cores.
- `force_quit` signal flag – `SIGINT` / `SIGTERM` set a global `volatile bool force_quit` flag. All worker threads poll this flag in their progress loops to coordinate graceful shutdown without requiring inter-thread synchronization primitives.

6.30.7.2 Resource Management

This section describes the setup and initialization of core resources required for virtio-blk operation after the framework initialization is complete. These resources include progress engines, memory pools, worker threads, and thread-local configurations necessary for high-performance block device emulation.

6.30.7.2.1 Resource Management Flow

1. Initialize application resources
    - Allocate PE (Progress Engine) context arrays for IO threads using `vblk_pci_dev_resources_init()`.
    - Allocate per-endpoint controller arrays (`struct vblk_ctrl`) and hotplug state arrays (`struct vblk_ep_hotplug_state`).
    - Allocate per-IO-context thread configuration array (`struct vblk_io_ctx_cfg`), which holds each IO thread's bound core index, progress engine, memory pool set, and other per-thread state.
2. Create progress engines
    - For each IO context thread, create a progress engine using `doca_pe_create()`.
    - Create a dedicated progress engine for the TLP thread.
    - Progress engine handles are stored in `struct vblk_pci_dev_pe_context`.
3. Set up shared memory pools
    - Create per-IO-context shared memory pool sets using `vblk_mpool_set_create()`.
    - Each pool set contains multiple tiers of buffer sizes for efficient allocation:

Buffer Size	Count	Use Case
4 KB	1024	Small I/O requests
8 KB	1024	Standard I/O requests
64 KB	512	Medium I/O requests
256 KB	512	Large I/O requests
512 KB	256	Very large I/O requests

    - Memory pools use `doca_mmap` for memory registration and `doca_buf_pool` for buffer management.
    - Huge pages are utilized for large contiguous memory allocations.
4. Create TLP thread
    - Spawn a dedicated TLP processing thread, bound to `tlp_core_idx`.
    - The TLP thread handles all PCIe TLP events: configuration reads/writes, MMIO, and MSI-X. Doorbell notifications are handled by hardware/DPA.
5. Create IO worker threads
    - For each IO context core (determined by `io_ctx_mask`), spawn a worker thread using `pthread_create()`.
    - Bind each thread to its designated CPU core using `pthread_setaffinity_np()`.
    - Thread handles are stored in `struct vblk_pci_dev_pe_context`.
6. Configure thread roles

- TLP thread – Handles PCIe TLP processing, PCI configuration space emulation, hotplug events, and stdin commands.
- IO threads – Process virtio-blk I/O requests and manage DMA transfers.
- Offload Engine thread – The IO thread whose core matches `offload_engine_core_idx` additionally runs the virtio offload engine for queue lifecycle management (create, bind, enable, disable, unbind, destroy).

7. Wait for thread readiness
   - After spawning TLP and IO threads, the main thread (which called `main()` → `vblk_pci_dev_run()` → `progress_contexts_start()`) synchronously spins on two barriers using atomic state variables:
     - `app_wait_io_ready()` – Spins (`sched_yield()` loop) until all IO threads signal ready. Each IO thread increments the atomic counter `vblk_app_io_ready` after completing its per-thread initialization; when the count reaches `num_io_ctx`, the state transitions to `VBLK_APP_IO_READY`.
     - `app_wait_ctrl_enabled()` – Spins until the offload engine thread signals that the virtio controller is enabled (state transitions to `VBLK_APP_CTRL_ENABLED`).
   - If any thread reports an error via the atomic `vblk_app_err` flag, or `force_quit` is set, the wait returns immediately with an error.
   - After all threads are ready, the main thread calls `vblk_app_join_io_and_tlp_threads()` to `pthread_join` all worker threads, blocking until shutdown. The main thread itself does not participate in TLP or IO processing.
8. If it needs to query virtio queue statistics info
   - Create doca_devemu_virtio_queue_dbg_state with calling doca_devemu_virtio_offload_engine_queue_dbg_state_create_list.
   - Issue the request to request for queue statistics info with calling doca_devemu_virtio_queue_dbg_state_populate_list.
   - Query whether the request has been completed with calling doca_devemu_virtio_queue_dbg_state_is_populated.
   - After the querying has been completed, call the corresponding API to get the statistics info

API name	purpose
doca_devemu_virtio_queue_dbg_state_get_id	queue idx
doca_devemu_virtio_queue_dbg_state_get_enabled	queue enabled/disabled
doca_devemu_virtio_queue_dbg_state_get_size	queue_size
doca_devemu_virtio_queue_dbg_state_get_inflights	num of inflight IOs
doca_devemu_virtio_queue_dbg_state_get_hw_avail_idx	device side avail_idx
doca_devemu_virtio_queue_dbg_state_get_driver_avail_idx	host driver avail_idx
doca_devemu_virtio_queue_dbg_state_get_hw_used_idx	device side used_idx
doca_devemu_virtio_queue_dbg_state_get_driver_used_idx	host driver used_idx

6.30.7.2.2 Key Points

- Thread affinity – Each worker thread must be bound to a dedicated CPU core to ensure optimal performance and minimize scheduling overhead.
- Offload engine role – One IO thread (on the offload engine core) is responsible for virtio queue lifecycle operations, while other IO threads focus solely on I/O processing.
- Memory pool sizing – Memory pool sizes and tiers should be configured based on anticipated workload and expected I/O sizes. The multi-tier design allows efficient buffer allocation across varying request sizes.
- Progress engines – Each worker thread requires its own progress engine to independently handle asynchronous operations and maintain high throughput.
- Resource isolation – Each IO context has its own dedicated memory pool set and DMA context, created independently to avoid cross-thread contention. `shared_mpools_create()` allocates one `vblk_mpool_set` per IO context (`res->shared_mpools[ctx_id]`), and `vblk_ctrl_io_ctx_init()` creates a separate `doca_dma` instance per IO context. No locks are needed for buffer allocation or DMA submission within a single thread.
- Error recovery – If resource creation fails at any stage, the application must perform cleanup of already-initialized resources to avoid leaks and maintain system stability.
- Scalability – Resource allocation logic scales with the number of configured CPU cores, allowing the application to adapt to various deployment environments.

6.30.7.3 PCI Switch Topology and Device Configuration

This section describes the process of constructing the software PCI switch topology, configuring PCI device types, and creating virtio-blk device instances. Unlike higher-level DevEmu approaches, this application operates at the TLP level, providing full control over the PCIe configuration space and device behavior.

6.30.7.3.1 PCI Switch Topology Flow

The following steps correspond to the call chain in `vblk_pci_init()` :

1. Build switch topology
    - Allocate device configuration structures for all devices: USP + DSPs + EPs + DUMMY.
    - Initialize USP (Upstream Switch Port) configuration space:
        - Type 1 header (bridge), class code `0x060400` .
        - PCIe Express capability as Upstream Port.
        - PM and VPD capabilities via `init_usp_capabilities()` .
    - Initialize DSP (Downstream Switch Port) configuration spaces:
        - Type 1 header (bridge) with hotplug slot capabilities.
        - PCIe Express capability as Downstream Port.
        - Hotplug slot capabilities including power control and surprise removal.
        - PM and VPD capabilities via `init_dsp_capabilities()` .
    - Initialize EP (Endpoint) configuration spaces:
        - Type 0 header (endpoint), virtio vendor ID `0x1AF4` .
        - MSI-X capability, PCIe Express capability as Endpoint.

2. Set up TLP channel
    - Create a TLP channel using **doca_devemu_pci_tlp_channel_create()**.
    - Register TLP request event handlers with **doca_devemu_pci_tlp_channel_event_req_register()**.
    - Enable ACG (Asynchronous Credit Grant) on the channel with **doca_devemu_pci_tlp_channel_set_acg_enabled()**.
    - Start the TLP channel context using **doca_ctx_start()**.
3. Initialize ACG queue (hotplug mode only)
    - Pre-allocate ACG request slots for sending MSI TLPs to the host during hotplug events.
4. Create PCI TLP type and configure capabilities
    - Create the virtio-blk PCI TLP type using **doca_devemu_vblk_pci_tlp_type_create()**.
    - Attach the DOCA device using **doca_devemu_pci_type_set_dev()**.
    - Register endpoint PCI capabilities using **doca_devemu_pci_tlp_type_set_pci_cap_conf()**:
        - PCI Express – PCIe Capability structure (offset 0x60, 60 bytes).
        - MSI-X – Message Signaled Interrupts eXtended (offset 0x9C, 12 bytes).
    - Configure MSI-X and doorbell counts: **doca_devemu_pci_type_set_num_msix()**, **doca_devemu_pci_type_set_num_db()**.
    - Start the PCI type with **doca_devemu_pci_type_start()**.
5. Configure BAR layout and virtio capabilities
    - Set default virtio PCI capability structures (common config, notify, ISR, device config, MSI-X).
    - Set BAR offsets based on runtime PCI type info by querying four region info lists:
        - **doca_devemu_pci_type_create_transaction_region_info_list()** – MMIO transaction regions (common config, ISR, device config).
        - **doca_devemu_pci_type_create_db_region_by_data_info_list()** – Doorbell regions.
        - **doca_devemu_pci_type_create_msix_table_region_info_list()** – MSI-X Table region.
        - **doca_devemu_pci_type_create_msix_pba_region_info_list()** – MSI-X PBA region.
    - A single BAR (BAR 0, 32KB) is used with the following layout:

Region	Offset	Length	Description
PCI Config	0x0000	0x100	Virtio PCI Common Configuration
ISR Config	0x0100	0x1	Interrupt Status Register
Device Config	0x0200	0x100	Virtio-blk Device Configuration
MSI-X Table	0x2000	0x1000	MSI-X Table entries
MSI-X PBA	0x3000	0x1000	MSI-X Pending Bit Array
Doorbells	0x4000	0x1000	Virtio queue notification doorbells

6. Create virtio device instances (runtime)
    - For each endpoint, create a representor with **doca_devemu_pci_type_create_rep()**.
    - Create TLP device instances using **doca_devemu_pci_tlp_dev_create()**.
    - Start TLP device with **doca_devemu_pci_tlp_dev_start()**.

- In static mode, all endpoints are created at startup; in hotplug mode, endpoints are created on **plug** command via the controller initialization path.

7. BDF (Bus/Device/Function) mapping (runtime, dynamic)
   - BDF addresses are assigned dynamically as the host enumerates the PCI bus via Config Read/Write TLPs.
   - A hash map (**bdf_entries**) provides fast BDF-to-device lookup.
   - **find_device_by_bdf()** walks the topology (USP → DSPs → EPs) on first access and caches the mapping.

6.30.7.3.2 Key Points

- TLP-level emulation – Full PCIe configuration space is emulated in software, providing complete control over device behavior and topology.
- Multi-device topology – The software PCI switch supports up to 32 endpoints, each presented as an independent virtio-blk device.
- Hotplug support – DSP configurations include hotplug slot capabilities, enabling runtime device addition and removal.
- ACG – Enables the application to asynchronously send TLPs (e.g., MSI memory writes for hotplug) to the host without waiting for a host-initiated request.
- BDF dynamic assignment – Device BDF addresses are assigned during host PCI enumeration and tracked dynamically.
- Capability configuration – Endpoint capabilities (Express, MSI-X) are registered via **doca_devemu_pci_tlp_type_set_pci_cap_conf()** . Bridge capabilities (PM, VPD) are set directly in **pci_device_caps** structures.

6.30.7.4 Virtio-blk Controller

This section describes the virtio-blk controller (**struct vblk_ctrl**), which manages the lifecycle and I/O processing for each emulated block device endpoint. Each endpoint has one controller that coordinates between the offload engine, I/O contexts, and virtio queues.

6.30.7.4.1 Controller Initialization Flow

Steps 1-2 happen in **vblk_ctrl_init()** (called by the offload engine thread). Steps 3-5 happen per IO thread via **vblk_ctrl_init_io_ctx_on_thread()** . All threads run **vblk_io_ctx_thread()** and synchronize via atomic signal/wait callbacks:

```
OE thread                    Non-OE threads

     vblk_ctrl_init()               wait_ctrl_created() [spin]
     (steps 1-2)
     signal_ctrl_created()          (unblocked)

     init_io_ctx_on_thread()        init_io_ctx_on_thread()
     (steps 3-4, own ctx_id)        (steps 3-4, own ctx_id)

     signal_io_ready()              signal_io_ready()

     wait_io_ready() [spin]         enter progress loop
     (wait all threads ready)
     vblk_ctrl_enable()
     (step 5)
     enter progress loop
```

1. Create PCI endpoint and VQ array

- Allocate per-queue state array (`ctrl->vqs`), and assign queues to IO contexts round-robin via `vblk_qid_to_io_ctx_id()` .
- Build device attributes: device features (`VBLK_F_MQ` , `F_SEG_MAX` , `F_SIZE_MAX` , optionally `F_INDIRECT_DESC`), block device config (`capacity` , `seg_max` , `size_max`), and callbacks (`virtio_ctrl_change_cb` for host state transitions, `virtio_ctrl_tlp_poll_cb` for TLP polling).
- Create the virtio-blk PCI endpoint using `vblk_pci_virtio_dev_create(&pci_attr)` .

2. Create and configure offload engine
   - Instantiate the virtio-blk offload engine using `doca_devemu_vblk_offload_engine_create()` , bound to the PCI endpoint's TLP device.
   - Configure `seg_max` using `doca_devemu_vblk_offload_engine_set_seg_max()` .
   - Cast to virtio offload engine via `doca_devemu_vblk_offload_engine_as_virtio_offload()` .
   - Enable indirect descriptor support (if configured) using `doca_devemu_virtio_offload_engine_set_indir_descs_enabled()` .
   - Set the number of queues using `doca_devemu_virtio_offload_engine_set_num_queues()` .
   - Start the offload engine with `doca_devemu_virtio_offload_engine_start()` .
3. Initialize I/O contexts (per thread)
   - Assign the shared memory pool (or create a per-IO-context pool if none is provided).
   - Create a `doca_devemu_vblk_io` instance (stored as `io_ctx->vq_io_ctx`) from the offload engine using `doca_devemu_vblk_io_create_from_offload_engine()` . This DOCA SDK object receives virtio-blk requests dispatched by the offload engine to this thread.
   - Create a DMA context using `doca_dma_create()` for host-DPU data transfers.
   - Configure DMA memcpy task pool size (`num_queues * VBLK_CTRL_MAX_QUEUE_SIZE`) and callbacks via `doca_dma_task_memcpy_set_conf()` .
   - Disable ordered completions with `doca_dma_set_ordered_completions(0)` to maximize DMA throughput.
   - Register virtio-blk request event handler using `doca_devemu_vblk_io_event_vblk_req_register(vblk_io_handler)` .
4. Connect and start contexts
   - Connect both the `doca_devemu_vblk_io` context and DMA context to the thread's progress engine using `doca_pe_connect_ctx()` .
   - Start DMA context, then start IO context with `doca_ctx_start()` .
   - IO context start may be asynchronous (`DOCA_ERROR_IN_PROGRESS`); the thread spins on `doca_pe_progress()` until the context reaches RUNNING state.
5. Enable offload engine
   - Once all I/O contexts across all threads are created and started, enable the offload engine using `doca_devemu_virtio_offload_engine_enable()` .
   - The device is now ready to process virtio-blk requests from the host.

6.30.7.4.2 Virtio Queue Lifecycle State Machine

Each virtio queue transitions through the following states:

```
DESTROYED    STARTING    BINDING    RUNNING
          UNBINDING    STOPPING
```

State	Description	Managed By
`DESTROYED`	VQ does not exist. Initial state or after `doca_devemu_vblk_req_vq_destroy()` completes.	–
`STARTING`	`doca_devemu_vblk_req_vq_create()` + `doca_devemu_virtio_vq_set_conf()` + `doca_devemu_virtio_vq_start()` pending.	Offload Engine thread
`BINDING`	`doca_devemu_virtio_io_bind_vq()` + `doca_devemu_virtio_vq_enable()` pending.	IO thread
`RUNNING`	VQ fully operational. `doca_devemu_virtio_vq_enable()` completed.	IO thread
`STOPPING`	OE thread calls `doca_devemu_virtio_vq_stop()` then `doca_devemu_vblk_req_vq_destroy()`. If stop fails, destroy is still attempted. If destroy fails, the queue stays in STOPPING and retries on the next progress cycle.	OE thread
`UNBINDING`	`doca_devemu_virtio_vq_disable()` + `doca_devemu_virtio_io_unbind_vq()` pending.	IO thread

Queue state transitions are triggered by:

- Host driver writes `DRIVER_OK` – Queues transition from `DESTROYED` to `STARTING`, then through `BINDING` to `RUNNING`.
- Host driver resets device – Queues transition from `RUNNING` through `STOPPING`, `UNBINDING`, back to `DESTROYED`.

6.30.7.4.3 Key Points

- Per-endpoint controller – Each endpoint has its own `struct vblk_ctrl` managing its offload engine, I/O contexts, and virtio queues.
- Thread-specific I/O contexts – Each IO thread has its own DMA and virtio-blk I/O contexts to avoid contention.
- Atomic state tracking – Virtio queue states are tracked using atomic variables for lock-free cross-thread coordination.

- Poll-based state machine – Queue state transitions are driven by poll counter tracking. A compile-time fallback (**VBLK_POLL_FALLBACK_ENABLE=1**) can be enabled for debugging stuck queues – it forces state polling even when counters are inaccurate, at the cost of increased latency.
- Queue-to-IO-context mapping – Queues are distributed across I/O contexts using round-robin assignment based on **queue_index % num_io_ctx** .

6.30.7.5 I/O Request Processing

This section describes the core operational phase where the application processes incoming virtio-blk requests from the host, executes corresponding block operations, and manages DMA data transfers between the host and DPU.

6.30.7.5.1 I/O Request Processing Flow

1. Receive virtio-blk requests from host
    - The DOCA SDK invokes the registered callback **vblk_io_handler(req, type, sector, req_user_data)** when a virtio-blk request arrives.
    - **type** (request type) and **sector** (LBA) are passed directly as callback parameters; data buffer length and scatter-gather list are obtained via **doca_devemu_vblk_req_get_data_len()** / **doca_devemu_vblk_req_get_data()** in the per-command handlers.
2. Dispatch by request type
    - **vblk_io_handler()** switches on **type** :
        - **VBLK_T_IN** (read) → **vblk_cmd_read()**
        - **VBLK_T_OUT** (write) → **vblk_cmd_write()**
        - **VBLK_T_GET_ID** → **vblk_cmd_get_id()**
        - Unknown types → complete with **VBLK_S_UNSUPP**
3. Allocate DMA buffers
    - Obtain a DMA buffer from the shared memory pool using **vblk_mpool_set_buf_get()** .
    - The buffer size is selected from the appropriate tier based on the request data size.
4. Process by request type

Request Type	Operation	DMA Direction
VBLK_T_IN (Read)	Read data from block device, DMA to host	DPU → Host
VBLK_T_OUT (Write)	DMA from host, write data to block device	Host → DPU
VBLK_T_GET_ID	Return device ID string (20 bytes)	DPU → Host

5. Execute DMA transfers
    - For read operations: Populate the DPU buffer with data, then submit a DMA memcpy task using **doca_dma_task_memcpy_alloc_init()** and **doca_task_submit()** to transfer data to host memory.
    - For write operations: Submit a DMA memcpy task to transfer data from host memory to the DPU buffer, then process the write.
    - DMA completion is handled asynchronously via registered callbacks.

6. Complete requests
    - Upon DMA completion, the registered callback is invoked.
    - Complete the virtio-blk request using **doca_devemu_vblk_req_complete()** with the appropriate status:
        - **VBLK_S_OK** – Success.
        - **VBLK_S_IOERR** – I/O error.
        - **VBLK_S_UNSUPP** – Unsupported operation.
    - Release DMA buffers back to the memory pool.
7. Update statistics
    - If **stats_ios_period** is configured, track request counts.
    - Periodically query queue debug state using **doca_devemu_virtio_queue_dbg_state** for performance monitoring.

6.30.7.5.2 Key Points

- Zero-copy transfers – DMA operations enable efficient data transfers between host and DPU memory without involving the CPU for data movement.
- Asynchronous processing – All DMA operations are handled asynchronously to prevent thread blocking and maintain system responsiveness.
- Multi-tier buffer pools – The multi-tier memory pool design (4 KB to 512 KB) allows efficient buffer allocation matching the actual request data size.
- Error propagation – All errors (DMA failures via **vblk_ctrl_dma_error()**, command failures in **vblk_io_handler()**) are reported to the host as **VBLK_S_IOERR** via **doca_devemu_vblk_req_complete()**.
- Thread isolation – Each IO thread processes requests independently with its own DMA context and memory pool, avoiding shared resource contention.
- Performance monitoring – Optional periodic statistics collection provides visibility into system health and performance.

6.30.7.6 Device Lifecycle Management

This section describes the procedures for managing device state transitions, handling hotplug events, and implementing virtio device reset.

6.30.7.6.1 Device Lifecycle State Machine

The application maintains a state machine for each endpoint:

```
INIT     CTRL_CREATED     IO_READY     CTRL_ENABLED     [Running]
                                                  ERROR
```

State	Description
INIT	Initial state before any resources are created.
CTRL_CREATED	Controller and offload engine are created and started.
IO_READY	All IO context threads have initialized their contexts.

State	Description
`CTRL_ENABLED`	Offload engine is enabled; device is operational.
`ERROR`	An error occurred; `force_quit` is set for shutdown.

6.30.7.6.2 Hotplug Management Flow

1. Receive hotplug command
    - In hotplug mode, the TLP thread reads stdin commands (`plug <DSP_IDX>` / `unplug <DSP_IDX>`).
    - The TLP thread sets `plug_requested` or `unplug_requested` in the per-EP `struct vblk_ep_hotplug_state` .
2. Plug operation (TLP thread → OE thread → IO threads)
    - The OE thread detects `plug_requested` and initiates controller creation (`vblk_ctrl_init()`).
    - Once the controller is created, the OE thread signals IO threads to create their I/O contexts (`need_io_ctx = true`).
    - Each IO thread creates its I/O context on its own thread and signals readiness.
    - When all IO contexts are ready, the OE thread enables the offload engine.
    - The TLP thread triggers PCI hotplug notification via `vblk_pci_trigger_hotplug()` , which sends an MSI to the host to signal presence detect and link active changes.
    - (Host side) The host PCI subsystem enumerates the new device and loads the `virtio_blk` driver. This happens outside the application.
3. Unplug operation (TLP thread → OE thread → all IO threads → OE thread)
    - The TLP thread receives the `unplug` command and enqueues the EP index into `unplug_queue` (FIFO). In its progress loop, the TLP thread dequeues one entry at a time (`unplug_active`) and sets `unplug_requested = true` . This ensures unplugs are serialized (i.e., only one EP is being unplugged at a time).
    - The OE thread detects `unplug_requested` in `oe_check_unplug_requests()` , sets `pending_unplug = true` , and sends an MSI to the host via `vblk_pci_trigger_hotplug(ep, false)` to signal link-down.
    - (Host side) The host responds by powering off the DSP slot. The TLP thread detects this in `bridge_cap_write()` and sets `host_power_off = true` .
    - The OE thread detects `host_power_off` in `oe_check_host_power_off()` and sets `need_io_ctx_shutdown = true` .
    - All IO threads (including OE) detect `need_io_ctx_shutdown` in `io_check_need_io_ctx_shutdown()` and shut down their own I/O contexts, incrementing `io_ctx_shutdown_done` .
    - The OE thread detects all IO contexts are shut down in `oe_check_unplug_complete()` , then performs controller cleanup (`vblk_ctrl_cleanup()`) and destroys the PCI device (`vblk_pci_destroy_device()`).
4. Virtio device reset

- Triggered when the host driver writes `device_status = 0`, detected in `virtio_ctrl_change_cb()` on the TLP thread.
- The TLP thread transitions queues based on their current state:
  - RUNNING / BINDING → UNBINDING
  - STARTING → STOPPING (no VQ object to unbind)
  - UNBINDING / STOPPING → already being torn down, just increment `pending_vqs`
- After state transitions, cached VQ configs are cleared (`memset`) to prevent OE from reading stale data.
- If `pending_vqs > 0`, `device_status` is held at `NEEDS_RESET` (0x40). The host polls until `device_status == 0` before reinitializing, guaranteeing `DRIVER_OK` only arrives after all VQs are destroyed.
- IO threads progress UNBINDING → STOPPING (disable + flush + unbind VQ).
- OE thread progresses STOPPING → DESTROYED (stop + destroy VQ), decrementing `pending_vqs` atomically via CAS in `vblk_ctrl_reset_vq_done()`.
- When the last VQ is destroyed (`pending_vqs` reaches 0), the OE thread directly sets `device_status = 0`, allowing the host to reinitialize.

6.30.7.6.3 Key Points

- Cross-thread coordination – Hotplug operations require careful coordination between the TLP thread, OE thread, and IO threads using atomic flags.
- Host visibility – The device becomes visible to the host after the PCI switch DSP signals presence and link-active status.
- Standard hotplug signaling – The application emulates standard PCIe hotplug signaling (slot status, link status, MSI interrupts via `send_msi_via_memory_write_tlp()`) for host driver compatibility.
- Reset Synchronization — Device reset uses `NEEDS_RESET` (0x40) to hold `device_status` non-zero while VQs are torn down. The OE thread atomically decrements `pending_vqs` via CAS and clears `device_status` to 0 when the last VQ is destroyed, allowing the host to reinitialize safely.
- State atomicity – Per-EP hotplug states use atomic variables for lock-free cross-thread communication.

6.30.7.7 Cleanup and Shutdown

This section describes the procedures for gracefully shutting down the application, freeing allocated resources, and ensuring clean termination.

6.30.7.7.1 Cleanup and Shutdown Flow

Shutdown happens in two phases:

- Phase 1 inside each worker thread (before `pthread_join` returns), and
- Phase 2 in `vblk_pci_dev_run()` after all threads have exited (resources are freed in reverse order of creation)

6.30.7.7.1.1 Phase 1 – Thread-internal cleanup (inside `vblk_io_ctx_thread()`)

All threads exit their `while (!force_quit)` main loop when `SIGINT` / `SIGTERM` sets `force_quit = true`.

1. Each IO thread shuts down its own IO contexts:
    - For each enabled EP, call `vblk_ctrl_shutdown_io_ctx_on_thread()`:
        - For each queue mapped to this `ctx_id` in RUNNING, UNBINDING, or BINDING state: call `vblk_ctrl_queue_stop()` (spins with `doca_pe_progress()` on `DOCA_ERROR_AGAIN`), then `doca_devemu_virtio_io_flush_vq()` + `doca_devemu_virtio_io_unbind_vq()`. Queues in DESTROYED, STOPPING, or STARTING are skipped (handled by OE thread).
        - Stop DMA and IO contexts with `doca_ctx_stop()`, wait for IDLE via `vblk_wait_ctx_idle()`.
2. OE thread (after its own IO ctx shutdown) additionally:
    - `wait_all_io_exited()` – waits for all non-OE IO threads to signal exit.
    - `wait_pci_stopped()` – waits for TLP thread to signal PCI stopped.
    - For each enabled EP, call `vblk_ctrl_cleanup()`:
        - `vblk_ctrl_vqs_destroy()` – iterate all queues and call `vblk_ctrl_vq_stop_destroy()` for each non-NULL VQ object.
        - `vblk_ctrl_io_ctxs_reset()` – reset IO contexts.
        - Disable → stop → destroy offload engine.
        - `vblk_pci_virtio_dev_destroy()` – destroy virtio-blk PCI endpoint.
        - `free(ctrl->vqs)`.
3. TLP thread exits its own `while (!force_quit)` loop:
    - `vblk_pci_stop()` – flush all pending ACG credits (`DOCA_DEVEMU_PCI_TLP_ACG_COMP_OPMODE_FLUSH`), then stop TLP channel context (`doca_ctx_stop()`).
    - Wait for TLP channel to reach IDLE via `vblk_wait_ctx_idle()`.
    - `signal_pci_stopped()`.
4. All IO threads (including OE) call `signal_io_exited()` before returning.

6.30.7.7.1.2 Phase 2 – Reverse-order resource cleanup (in `vblk_pci_dev_run()` after `pthread_join`)

After all threads are joined, resources are freed in reverse order of creation:

1. `vblk_pci_reset()` – defensive cleanup of any residual PCI resources: destroy remaining TLP devices and representors, stop and destroy PCI type, destroy TLP channel, free ACG queue / virtio_devs / BDF map / device configs, zero the TLP context.
2. `vblk_reset()` – tear down virtio-blk framework via `doca_devemu_vblk_teardown()`.
3. `shared_mpools_destroy()` – destroy each per-IO-context memory pool set.
4. `progress_contexts_destroy()` + `doca_libs_close()` – destroy all progress engines, close DOCA device.

5. `vblk_pci_dev_resources_cleanup()` – free PE context arrays, controller arrays, hotplug state arrays.
6. `free(io_cfgs)` – free IO context thread configs.
7. `doca_argp_destroy()` – destroy argument parser (in `main()`).

6.30.7.7.2 Key Points

- Ordered cleanup – Resources are released in reverse order of creation, respecting dependencies.
- Thread synchronization during shutdown – Three synchronization barriers coordinate the shutdown:
    a. OE thread calls `wait_all_io_exited()` to wait for all non-OE IO threads to finish their per-EP IO context shutdown and call `signal_io_exited()`.
    b. OE thread calls `wait_pci_stopped()` to wait for TLP thread to flush ACG, stop TLP channel, and call `signal_pci_stopped()`.
    c. Main thread `pthread_join`s all worker threads (IO + TLP) before entering Phase 2 resource cleanup.
- Graceful queue teardown – Virtio queues are disabled, flushed, and unbound before destroying offload engines.
- Signal handling – `SIGINT` / `SIGTERM` only set `force_quit = true`. Threads exit at the next loop iteration boundary. After exiting the loop, each thread proceeds to stop its DMA and IO contexts via `doca_ctx_stop()` and waits for them to reach IDLE via `vblk_wait_ctx_idle()`.
- Error resilience – Cleanup proceeds even if individual steps encounter errors (cleanup functions use `(void)` to ignore return values and continue).

6.30.7.8 SRIOV mode

6.30.7.8.1 Resource management

Expose and service SR-IOV on the emulated vBLK PF using eager provisioning, lazy activation, and final-only destruction

1. Advertise ACS on every emulated DSP.
2. All VF resources are allocated during PF initialization, including tlp_dev & OE.
3. Host VFE changes merely enable or disable existing datapaths (OE & VQ).

6.30.7.8.1.1 Limitation

1. Doesn't support ARI, so max VFs per PF is 7.
2. Doesn't support SRIOV-enabled PF FLR.
3. Inflight IO is expected when FLR is running with FIO.
4. No graceful exit is guaranteed if teardown with FIO running.
5. VF & PF should share the same DB and MSIX configuration.
6. If VF passthrough to VM, before host reboot, need VM shutdown and VF unpluged.

6.30.7.8.1.2 Capacity and reservation

For P PFs, V VFs per PF, and Q queues per function

- Dynamic VF slots: P × V
- Total functions: P × (V + 1)
- Doorbells: P × (V + 1) × Q
- MSI-X vectors: P × (V + 1) × (Q + 1)

6.30.7.8.1.3 Per-VF resource footprints

Every configured VF owns:

- One FW representor and dynamically assigned VF index
- One persistent VF TLP device
- One PCI/virtio configuration template
- One vblk_ctrl and an independent offload engine
- Q virtqueues
- One I/O context, DMA context, and DMA PE per application I/O thread
- Queue-statistics state when statistics are enabled

6.30.7.8.1.4 Runtime Lifecycle

- VFE 0 → 1: Marks the selected VFs present and sets dp_want_up. The OE thread enables their existing engines.
- VFE 1 → 0: Removes config/MMIO visibility and disables engines. Controllers, TLP devices, I/O contexts, and shadows remain allocated.
- PF FLR: N/A
- VF FLR: Recycles only the affected VF through UP/DOWN → FLR_TEARDOWN → FLR_REBUILD → UP/DOWN. Each I/O thread destroys and recreates its own VF I/O context; the OE thread restarts the engine.
- PF unplug or application shutdown: I/O threads drain VF contexts, the OE thread stops VF TLP devices and destroys VF controllers, then the TLP-side teardown closes VF representors and finally destroys the PF representor

Note: No guarantee of in-flight IO draining while FLR

6.30.7.8.2 Init Flow

6.30.7.8.2.1 Global SR-IOV initialization

vblk_pci_dev_run()
└── vblk_pci_init()
├── vblk_pci_sriov_probe_caps()
├── vblk_pci_sriov_pool_cleanup()
└── vblk_pci_sriov_pool_init()

├── vblk_pci_sriov_validate_caps()
└── Validate VF pool, MSI-X and doorbell capacity

└── vblk_pci_switch_init()
└── Allocate devs_config entries for all configured VFs

6.30.7.8.2.2 Per-PF SR-IOV creation

Static mode caller:

vblk_io_ctx_thread() [OE thread]
└── plug_ep_ctrl()

Hotplug-mode caller:

oe_check_plug_requests()
└── plug_ep_ctrl()

Both continue through the same path:

plug_ep_ctrl()
└── vblk_ctrl_init()
└── vblk_pci_virtio_dev_create()

6.30.7.8.2.3 PF representor and VF-range reservation

vblk_pci_virtio_dev_create()
└── vblk_pci_create_pf_rep()
└── vblk_pci_sriov_create_pf_rep_ex()
├── vblk_pci_sriov_pool_alloc()
├── Configure TotalVFs, VF index, stride, MSI-X and DB
└── doca_devemu_pci_type_create_rep_ex()

create_rep_ex() creates the PF representor and reserves backing resources for every VF.

6.30.7.8.2.4 Per-PF SR-IOV state

After PF configuration initialization:

vblk_pci_virtio_dev_create()
└── vblk_pci_sriov_init_pf_state()
├── Allocate struct vblk_sriov_pf_state
├── Allocate sriov->vf_slots[]
├── Initialize each VF config/template slot
├── vblk_pci_sriov_discover_vf_reps()
└── vblk_pci_sriov_start_vfs()

6.30.7.8.2.5 VF representor discovery

vblk_pci_sriov_init_pf_state()
└── vblk_pci_sriov_discover_vf_reps()
├── doca_devinfo_rep_create_list()
├── Match VF by parent PF VUID and PF-local VF index
├── doca_dev_rep_open()
└── vblk_pci_virtio_dev_init_vf()
└── Create slot->vf_dev

Initialization fails if any configured VF representor is missing.

6.30.7.8.2.6 VF TLP-device creation

```
vblk_pci_sriov_init_pf_state()
└── vblk_pci_sriov_start_vfs()
└── For each VF:
vblk_pci_sriov_start_vf_slot()
├── doca_devemu_pci_tlp_dev_create()
├── doca_devemu_pci_tlp_dev_start()
└── Store TLP device in slot->vf_dev->pci_tlp_dev
```

VF TLP devices are created eagerly and persist across VFE changes.

Because VFE=0 and NumVFs=0 initially, all VF resources exist but all VFs remain hidden.

6.30.7.8.2.7 VF datapath construction

```
vblk_io_ctx_thread() / oe_check_plug_requests()
└── oe_build_vf_engines_for_ep
└── For each VF:
vblk_ctrl_init_vf_engine()
├── Allocate VF queues
├── Create VF offload engine
└── Leave engine disabled
```

6.30.7.8.3 runtime model

SR-IOV resources are persistent after initialization. Runtime operations change VF visibility and datapath state; they do not normally allocate or destroy VF representors, TLP devices, controllers, or I/O contexts.

6.30.7.8.3.1 Control path

```
Host writes NumVFs/VFE/MSE
→ vblk_pci_sriov_ext_cap_write()
→ update SR-IOV shadows and VF presence
→ vblk_pci_sriov_set_vf_datapath_wanted()
```

- NumVFs: selects active VFs.
- VFE: controls VF presence and engine intent.
- MSE: controls VF MMIO decoding.

6.30.7.8.3.2 Datapath transition

```
DOWN --VFE=1--> UP
DOWN <--VFE=0-- UP
```

oe_check_vf_dp() reads dp_want_up and calls vblk_ctrl_enable() or vblk_ctrl_disable().

io_vf_progress() processes VF DMA and queues only while dp_state == UP.

6.30.7.8.3.3 Reset

- VF FLR: I/O threads destroy/rebuild that VF's I/O contexts; OE restarts its engine.

- PF FLR: N/A
- Unplug/shutdown: Permanently drains and destroys VF datapaths.

Ownership: TLP thread handles control, OE thread manages engines, and I/O threads manage per-core VF I/O contexts.

6.30.7.8.4 Final teardown

Final teardown occurs on PF unplug or application shutdown. Each I/O thread first drains and stops its VF I/O contexts; then the OE thread stops VF TLP devices and destroys VF controllers and engines. Finally, VF templates and representors are released, the PF representor is destroyed, and its dynamic VF range is returned to the pool.

6.30.8 Virtio-blk Operations Support Summary

The following table summarizes the virtio-blk operations supported by the reference application:

Virtio-blk Operation	Type Code	Status	Description
Read	`VBLK_T_IN` (0)	✅ Supported	Read data from block device
Write	`VBLK_T_OUT` (1)	✅ Supported	Write data to block device
Flush	`VBLK_T_FLUSH` (4)	❌ Not Implemented	Flush cached data to persistent storage
Get Device ID	`VBLK_T_GET_ID` (8)	✅ Supported	Return 20-byte device serial number
Get Lifetime	`VBLK_T_GET_LIFETIME` (10)	❌ Not Implemented	Query device lifetime statistics
Discard	`VBLK_T_DISCARD` (11)	❌ Not Implemented	Discard (TRIM/UNMAP) a range of sectors
Write Zeroes	`VBLK_T_WRITE_ZEROES` (13)	❌ Not Implemented	Write zeroes to a range of sectors
Secure Erase	`VBLK_T_SECURE_ERASE` (14)	❌ Not Implemented	Securely erase a range of sectors

6.30.8.1 Virtio Feature Bits Support Summary

Feature Bit	Status	Description
`VIRTIO_F_VERSION_1` (bit 32)	✅ Supported	Virtio-blk 1.0 compliance
`VIRTIO_F_ACCESS_PLATFORM` (bit 33)	✅ Supported	Device access restricted to platform IOMMU
`VIRTIO_BLK_F_SIZE_MAX` (bit 1)	✅ Supported	Maximum segment size is reported
`VIRTIO_BLK_F_SEG_MAX` (bit 2)	✅ Supported	Maximum segments per request is reported
`VIRTIO_BLK_F_MQ` (bit 12)	✅ Supported	Multi-queue support

Feature Bit	Status	Description
`VIRTIO_F_INDIRECT_DESC` (bit 28)	✅ Supported (optional)	Indirect descriptor support (enabled via `--indirect` flag)

6.31 DOCA Telemetry Applications

DOCA provides the following reference applications relating to DOCA Telemetry functionality:

- DOCA Telemetry OpenTelemetry Application
- DOCA Telemetry Traceback Application

6.31.1 DOCA Telemetry OpenTelemetry Application

This guide provides an example of an OpenTelemetry implementation on top of NVIDIA® BlueField® DPU or NVIDIA® ConnectX® NIC.

6.31.1.1 Introduction

The DOCA Telemetry OpenTelemetry reference application demonstrates how to directly link DOCA Telemetry libraries with the open-source OpenTelemetry SDK to export hardware metrics.

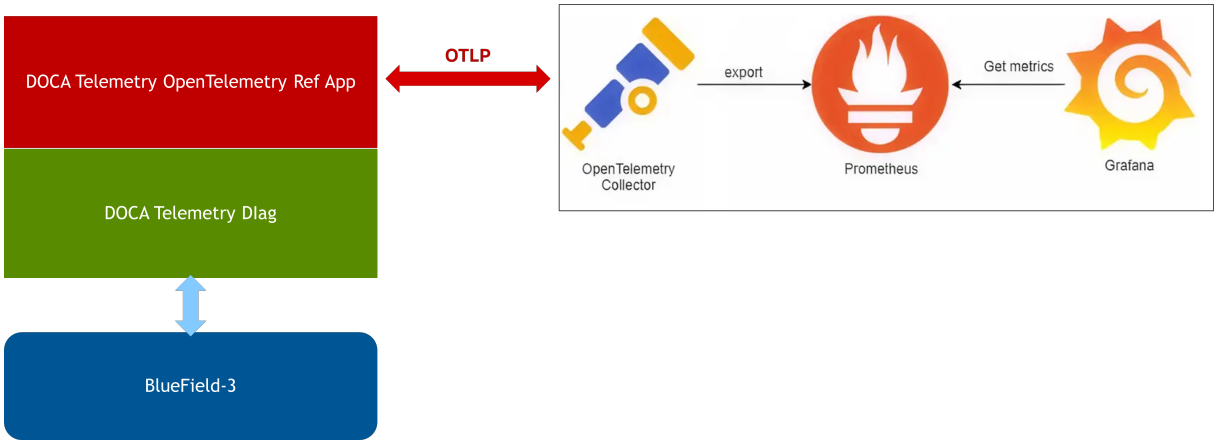
The application utilizes the DOCA Telemetry Diag library to periodically extract "on-demand" statistics from an NVIDIA® BlueField® DPU or ConnectX® NIC. These statistics are packaged as OpenTelemetry Protocol (OTLP) metrics and exported via HTTP.

While this reference application focuses strictly on exporting OpenTelemetry metrics, the underlying architecture can be adapted to handle other OpenTelemetry formats (such as traces and logs) or to support alternative DOCA Telemetry libraries for statistic generation.

6.31.1.2 System Design

The `doca_telemetry_opentelemetry` application is designed to operate as a data exporter within a broader telemetry infrastructure.

For example, a full implementation may look as follows:



In a standard deployment pipeline:

	Stage	Details
1	Data extraction	The application pulls telemetry data from the DPU/NIC via the DOCA Telemetry Diag library.
2	Export	The data is exported via HTTP to an OpenTelemetry Collector.
3	Translation and storage	The Collector converts the OTLP data into a format suitable for a time-series database (e.g., Prometheus).
4	Visualization	An external dashboard (e.g., Grafana) queries the database to display and analyze the metrics.

Note on Vendor Agnosticism

OTLP is fundamentally vendor-agnostic. While Prometheus and Grafana are cited in this reference design, administrators can route the generated OTLP statistics to any compatible backend service. Data routing and backend storage management are strictly out of scope for this application.

6.31.1.3 Application Architecture

The application periodically polls the underlying device "on demand," meaning a single sample of the specified `data_id`s is captured during each polling cycle. The application calculates the deltas for each `data_id`, packages them as OpenTelemetry metrics, and exports them to a designated IP address at a user-defined interval.

6.31.1.3.1 Supported Diag Data IDs

The application currently supports 15 DOCA Diag `data_id` types. Administrators can modify and apply each ID multiple times to collect metrics across different contexts (e.g. polling data from different physical ports).

⚠ For `data_id` bitmask variables, `XX` represents the local port, and `Y` represents the local port priority.

Name	Description	Data ID
`port_rx_bytes`	The number of received bytes on the physical port	`0x10200001000000XX`
`port_priority_rx_bytes`	The number of received bytes on the physical port and priority	`0x10200002000000YXX`
`port_rx_packets`	The number of received packets on the physical port	`0x10200003000000XX`
`port_priority_rx_packets`	The number of received packets on the physical port and priority	`0x10200004000000YXX`
`port_rx_discard_buf_packets`	The number of received packets dropped due to lack of buffers on a physical port	`0x10200005000000XX`
`port_priority_rx_pauses_packets`	The number of link-layer pause packets received on a physical port and priority	`0x10200006000000YXX`
`port_rx_transport_ecn_packets`	The number of RoCEv2 packets received by the notification point marked as experiencing congestion	`0x10800004000000XX`
`port_rx_transport_cnp_handled_packets`	The number of CNP received packets handled by the Reaction Point, per port	`0x10800005000000XX`
`port_tx_transport_cnp_sent_packets`	The number of CNP packets sent by the Notification Point, per port	`0x11000001000000XX`
`tx_transport_done_due_to_cc_deschedule_events`	The number of QP de-scheduled due to congestion control rate limitation	`0x11000002000000000`
`port_tx_bytes`	The number of transmitted bytes on the physical port (excluding loopback traffic)	`0x11400001000000XX`
`port_priority_tx_bytes`	The number of transmitted bytes on the physical port and priority (excluding loopback traffic)	`0x11400002000000YXX`
`port_tx_packets`	The number of transmitted packets on the physical port (excluding loopback traffic)	`0x11400003000000XX`
`port_priority_tx_packets`	The number of transmitted packets on the physical port and priority (excluding loopback traffic)	`0x11400004000000YXX`
`port_priority_tx_pauses_packets`	The number of link-layer pause packets transmitted on a physical port and priority	`0x11400005000000YXX`

6.31.1.3.2 OpenTelemetry Labelling

Every exported counter metric is tagged with the following contextual labels (where applicable), enabling backend applications to accurately filter and process the telemetry data:

Label Key	Description
`Name`	The exact Diag `data_id` name (including the port number, if applicable)
`Description`	A human-readable description of the Diag `data_id`
`Unit`	The measurement unit (e.g., Packets, Bytes, or Events). This may be implicitly included in the metric name.
`host.name`	The hostname of the server executing the OpenTelemetry application
`device.pcie`	The PCIe address of the target NIC/DPU on the host
`hw.network.io.direction`	The traffic direction updating the counter (`transmit` or `receive`)
`hw.network.io.physical.port`	The specific port number associated with the metric
`hw.network.io.priority`	The specific priority number associated with the metric

6.31.1.4 Libraries

6.31.1.4.1 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Telemetry Diag

Refer to its official programming guide for extended capability details.

6.31.1.4.2 OpenTelemetry C++ SDK

The DOCA Telemetry OpenTelemetry application relies on the C++ OpenTelemetry-SDK. Administrators must install this SDK on the target system before the DOCA application can be compiled.

1. Ensure the necessary compilation tools (`cmake` , `curl` , etc.) are present on the system:

```
# For Ubuntu/Debian:
$ sudo apt-get install -y git cmake g++ libcurl4-openssl-dev

# For RHEL/CentOS:
$ sudo yum install -y git cmake gcc-c++ libcurl-devel
```

2. Clone the OpenTelemetry repository and compile it using the specific CMake flags required by the DOCA reference application.

```
$ git clone https://github.com/open-telemetry/opentelemetry-cpp.git
$ cd opentelemetry-cpp/
```

```
$ mkdir build && cd build
$ cmake .. -DCMAKE_BUILD_TYPE=Release -DWITH_OTLP_HTTP=ON -DWITH_HTTP_CLIENT_CURL=ON
$ cmake --build . --config Release -- -j$(nproc)
$ sudo cmake --install . --prefix /usr
```

CMake Installation Path

The DOCA Meson build file strictly expects the **opentelemetry-cpp** CMake configuration to reside at **/usr/lib/cmake/opentelemetry-cpp**. If you choose to install the SDK in a custom directory (by altering the **cmake --install --prefix** path), you must manually update the Meson build file and your **LD_LIBRARY_PATH** to reflect the new location.

6.31.1.5 Compiling the Application

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for this application is located at:

```
/opt/mellanox/doca/applications/telemetry/opentelemetry/
```

6.31.1.5.1 Compiling All Applications

By default, the DOCA Meson project is configured to build every reference application simultaneously.

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

The compiled binaries will be generated in **/tmp/build/telemetry/**.

6.31.1.5.2 Compiling Only the Current Application

To significantly reduce build times, administrators can configure Meson to isolate and compile only the Telemetry OpenTelemetry application. This can be achieved via CLI flags or by modifying the configuration file.

6.31.1.5.2.1 Option 1: Command Line Configuration

Append the following flags to the **meson** setup command to disable all other applications and explicitly enable OpenTelemetry:

```
$ cd /opt/mellanox/doca/applications/
$ meson /tmp/build -Denable_all_applications=false -Denable_telemetry=true -Denable_telemetry_opentelemetry=true
 -Denable_telemetry_traceback=false
$ ninja -C /tmp/build
```

6.31.1.5.2.2 Option 2: Configuration File

You can persistently isolate the build by editing the **meson_options.txt** file directly.

1. Open **/opt/mellanox/doca/applications/meson_options.txt** and set the following parameters:
   - **enable_all_applications** = **false**
   - **enable_telemetry** = **true**
   - **enable_telemetry_opentelemetry** = **true**
   - **enable_telemetry_traceback** = **false**

   > **Configuration Rule**
   >
   > If **enable_all_applications** is set to **false**, the main **enable_telemetry** flag must be set to **true** for any underlying telemetry application to compile successfully. The individual applications default to **true**, but you can toggle them on or off as needed once the master telemetry flag is enabled.

2. Once the file is saved, run the standard compilation commands:

```
$ cd /opt/mellanox/doca/applications/
$ meson /tmp/build
$ ninja -C /tmp/build
```

6.31.1.6 Running the Application

6.31.1.6.1 Application Execution

The DOCA Telemetry OpenTelemetry application is distributed as source code and must be compiled prior to execution.

Execution syntax:

```
$ doca_telemetry_opentelemetry [DOCA Flags] [Program Flags]
```

Execution example:

```
$ sudo ./doca_telemetry_opentelemetry -p 03:00.0
```

> **Root Privileges**

The application strictly requires `sudo` (root privileges) to access hardware-level telemetry data on the device.

PCIe Addressing

Ensure the `-p` parameter (e.g., `03:00.0`) exactly matches the physical PCIe address of your target device.

6.31.1.6.2 Command Line Flags

6.31.1.6.2.1 General Flags

Short Flag	Long Flag	Description
`-h`	`--help`	Prints a help synopsis and exits
`-v`	`--version`	Prints program version information and exits
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.31.1.6.2.2 Program Flags

Short Flag	Long Flag	Description
`-p`	`--pci-addr`	Mandatory flag. The exact PCIe address of the device to read telemetry information from.
`-d`	`--data-id`	A specific DOCA Diag `data_id` (e.g., `0x1020000100000000`) representing a telemetry value to export via OTLP. Can be applied multiple times.

Short Flag	Long Flag	Description
`-f`	`--data-id-file`	The path to a file containing a list of `data_id` s (one per line) to read and export.
`-i`	`--ip`	The IPv4 address of the target OpenTelemetry server. Data is sent via HTTP POST to the OTLP default port `4318`. Defaults to `localhost`.
`-t`	`--time`	The polling interval in milliseconds between each HTTP export. Defaults to `2` seconds (`2000` ms).
`-vb`	`--verbose`	Runs the application in verbose mode, printing telemetry counters as DOCA logs in addition to exporting them.

Empty Data IDs

If no `-d` flags or `-f` file are provided, the application automatically defaults to monitoring all supported DOCA Diag `data_id` s with ports and priorities set to `0`.

Unsupported Data IDs

It is not guaranteed that all `data_id` s supported by the application are supported by your specific hardware. The application will report any unsupported IDs, requiring administrators to adjust their inputs accordingly.

6.31.1.6.3 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.31.1.7 Application Code Flow

6.31.1.7.1 Phase 1: Argument Parsing

The application begins by configuring the DOCA argument parser and consuming the user's CLI flags.

- `doca_argp_init()` : Initializes the argument parser resources and registers standard DOCA parameters.
- `register_opentelemetry_params()` : Registers the application-specific OpenTelemetry parameters (e.g., IP address, polling time).
- `doca_argp_start()` : Executes the parser against the provided inputs.

6.31.1.7.2 Phase 2: OpenTelemetry Initialization

The application establishes the connection to the hardware and provisions the OpenTelemetry metric pipelines.

- `opentelemetry_init()` : Initializes the core application.

- Opens the specified DOCA device for Diag Telemetry processing.
- Verifies hardware compatibility: checks support for DOCA Telemetry Diag, confirms it can handle the requested number of `data_id` s, and ensures it supports `on_demand` processing.
- Configures the Diag Context: sets the sample mode to `on_demand` , configures the output format (`FORMAT_0`), applies required `no_sync` options, and loads the target `data_id` s.

- `opentel_metrics_init()` : Initializes the OpenTelemetry metrics framework.
  - Creates resource attributes identifying the DOCA reference application as the data source.
  - Configures the OTLP exporter and reader based on the target IP and export interval.
  - Registers a global Meter Provider so all subsequently created counters inherit this configuration.
- `opentel_counter_init()` : Initializes the specific metrics counters.
  - For each `data_id` applied to the Diag Context, it creates an OpenTelemetry 64-bit counter and compiles the corresponding label information (name, unit, port, etc.).

6.31.1.7.3 Phase 3: Main Execution Loop

The application enters its primary monitoring state, continuously polling and exporting data.

- `opentelemetry_run()` : Triggers the execution loop.
  - Starts the DOCA Diag context to begin hardware sampling.
  - Enters a continuous loop (until interrupted by `CTRL+C`):
    - Queries the Diag context for `on_demand` counter data.
    - Iterates through each returned `data_id` , creating a new label instance and updating the associated OpenTelemetry counter with the calculated delta.
    - Sleeps for half the configured export time to ensure fresh data is available for the background OTLP export thread.
  - Stops the DOCA Diag context upon loop termination.

6.31.1.7.4 Phase 4: Teardown and Cleanup

The application safely releases all hardware and software resources.

- `opentelemetry_destroy()` : Destroys the core application objects.
  - Destroys the DOCA Diag context.
  - Closes the connection to the DOCA device.
- `doca_argp_destroy()` : Destroys the argument parser resources.

> ⚠ OpenTelemetry SDK objects are automatically destroyed when they fall out of scope and do not require manual destruction.

6.31.1.8 References

- `/opt/mellanox/doca/applications/telemetry/opentelemetry/`

6.31.2 DOCA Telemetry Traceback Application

6.31.2.1 Introduction

The DOCA Telemetry Traceback application provides a framework for monitoring telemetry data to identify long-term trends locally on the device. Because funneling massive volumes of raw telemetry data to a centralized data store can overwhelm network resources at a data-center scale, this application processes data directly on the NVIDIA® BlueField® DPU or NVIDIA® ConnectX® NIC.

The application gathers telemetry data into a circular buffer, retaining the last *N* samples from the device. This historical data is continuously evaluated against user-defined rules to detect specific "events" (anomalies or notable threshold breaches). When an event triggers, the application writes the preceding historical data and all subsequent active event data to a log file.

> The included rules are purposefully simplistic, as this application is intended to serve as a reference traceback model rather than a fully featured data processing engine

6.31.2.2 System Design

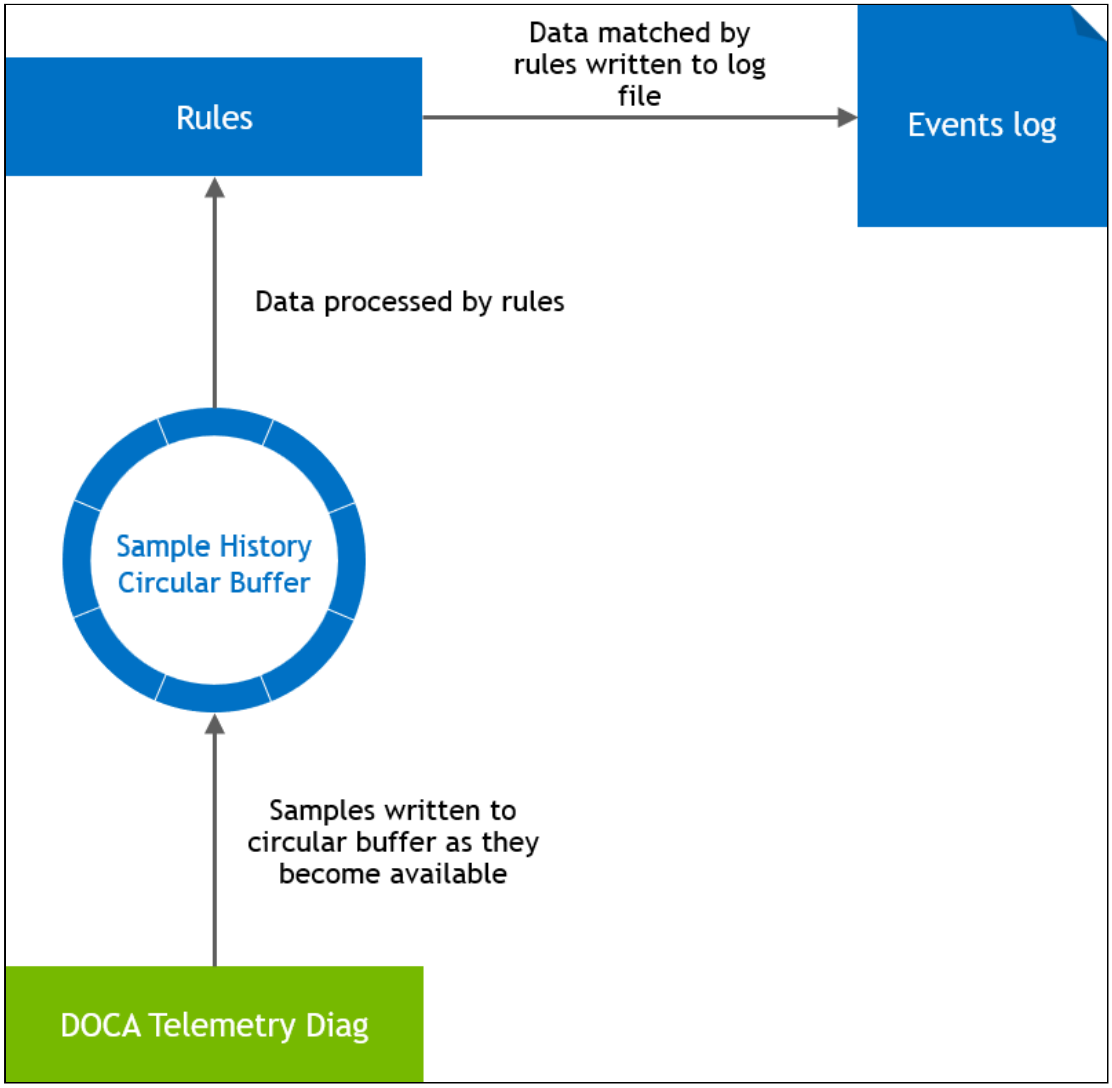
The **doca_telemetry_traceback** application is a standalone application which runs on top of a NVIDIA® BlueField® DPU or NVIDIA® ConnectX® NIC.

6.31.2.3 Application Architecture

The **doca_telemetry_traceback** application is a standalone service composed of four primary components:

1. DOCA Telemetry Diag (**doca_telemetry_diag**) instance – The underlying data provider. This instance is configured using parameters from the application's JSON configuration file (e.g., sample period, data IDs) alongside hardcoded application values.
2. Sample history circular buffer (**samples_circular_buffer**) – A contiguous block of memory that stores the telemetry samples. Each record is stored as **timestamp, data_id[0], ... data_id[N]** . To ensure there is an equal amount of historical data and active event data, the buffer size is automatically doubled based on the user's requested depth.
3. Rule (**rule**) collection – A set of logical monitors. Each rule monitors a specific **data_id** against defined logic. When the logic conditions are met, an "event start" is triggered. The rule remains active until new data arrives that no longer meets the conditions, triggering an "event end".
4. Events log file – The output destination for event data. Upon execution, the application logs its configuration to ensure absolute certainty regarding the parameters that triggered an event. When an event starts, the log captures the available history, an "event started" marker, all data collected during the event, and an "event end" marker.

If multiple events activate simultaneously, individual begin/end rows are inserted to track their respective lifecycles. Data logging pauses once all events conclude and resumes upon the next activation.



6.31.2.3.1 Traceback Configuration File

The application retrieves its operational parameters from a JSON configuration file.

Example configuration:

```
{
  "diag_sample_period_ns": 100000,
  "sample_history_depth": 10000,
  "data-ids": [
    {
      "name": "port_0_rx_bytes",
      "data_id": "0x1020000100000000"
    }
```

```
  ],
  "rules": [
    {
      "name": "Rx Bytes/s > 1MeBs",
      "data_id": "0x1020000100000000",
      "comparison_type": "differential_greater_than",
      "comparison_value": 1048576
    }
  ]
}
```

Field	Type	Notes
`diag_sample_period_ns`	Integer	The sampling interval in nanoseconds. This is passed directly to the diag instance via `doca_telemetry_diag_get_sample_period`.
`sample_history_depth`	Integer	The number of historical records a rule evaluates. The application automatically provisions a circular buffer double this size.
`data-ids`	Array	A collection of data ID definitions.
`data-ids[N].name`	String	A friendly name for the `data_id` utilized as the column header in the events log.
`data-ids[N].data_id`	String	A 16-character hexadecimal value. Format: `0x[0-9a-fA-F]{16}`.
`rules`	String	A collection of rule definitions.
`rules[N].name`	String	A friendly name used to identify the rule in the events log.
`rules[N].data_id`	String	The 16-character hexadecimal `data_id` this rule monitors. Must match an ID defined in the `data-ids` array.
`rules[N].comparison_type`	String	Defines the rule's operational logic. Acceptable values: • `incremental_greater_than` (`inc_gt`) • `differential_greater_than` (`dt_gt`) • `differential_less_than` (`dt_lt`) • `greater_than` (`gt`) • `less_than` (`lt`) See section "Rule Definitions" for details.
`rules[N].comparison_value`	Integer	The numerical threshold utilized when evaluating the rule. See section "Rule Definitions" for details.

6.31.2.3.2 Rule Definitions

The application supports five distinct rule types. Each rule establishes its own hysteresis logic, defining precisely when an event activates and deactivates.

6.31.2.3.2.1 Incremental Greater Than (`inc_gt`)

Designed for incrementally increasing values to guarantee triggering each time a value increases by the comparison threshold.

Activation Trigger	Activates if the sampled value exceeds the calculated setpoint (Initial Setpoint = First Sampled Value + `comparison_value`).
Deactivation Trigger	Deactivates immediately once any new sampled value exceeds the current setpoint, and a new setpoint is calculated. ⚠ Rapid data spikes exceeding the setpoint multiple times simultaneously will only activate the rule once.

6.31.2.3.2.2 Differential Greater Than (`dt_gt`)

Evaluates the delta between the current sample and an older historical sample. Requires at least `sample_history_depth` values in the buffer before functioning.

Activation Trigger	Activates if the delta between the newest sample and the sample at `[N-(sample_history_depth-1)]` exceeds the `comparison_value`.
Deactivation Trigger	Deactivates only after `sample_history_depth` consecutive samples are acquired where each new delta remains below the `comparison_value`.

6.31.2.3.2.3 Differential Less Than (`dt_lt`)

Operates using the identical historical delta logic as `dt_gt`, but activates when the delta is *less* than the `comparison_value`.

6.31.2.3.2.4 Greater Than (`gt`)

Designed strictly for values that can fluctuate (increase and decrease). It is not suitable for metrics that only increment (e.g., total bytes transferred).

Activation Trigger	Activates when the sampled value explicitly exceeds the `comparison_value` for `sample_history_depth` times consecutively.
Deactivation Trigger	Deactivates only after `sample_history_depth` consecutive samples fall *below* the `comparison_value`.

6.31.2.3.2.5 Less Than (`lt`)

Operates using the identical fluctuation logic as `gt`, but activates when the sampled value remains below the `comparison_value` for the required consecutive depth.

6.31.2.4 DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Telemetry Diag

Refer to its official programming guide for extended capability details.

6.31.2.5 Compiling the Application

> Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

DOCA reference applications are installed with full source code and build instructions. This allows you to compile them as-is or modify the source code to create custom versions.

> For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The source code for this application is located at:

```
/opt/mellanox/doca/applications/telemetry/traceback/
```

6.31.2.5.1 Compiling All Applications

By default, the DOCA Meson project is configured to build every reference application simultaneously.

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

> The compiled binaries will be generated in **/tmp/build/telemetry/**.

6.31.2.5.2 Compiling Only the Current Application

To significantly reduce build times, administrators can configure Meson to isolate and compile only the Telemetry Traceback application. This can be achieved via CLI flags or by modifying the configuration file.

6.31.2.5.2.1 Option 1: Command Line Configuration

Append the following flags to the **meson** setup command to disable all other applications and explicitly enable Traceback:

```
$ cd /opt/mellanox/doca/applications/
$ meson /tmp/build -Denable_all_applications=false -Denable_telemetry=true -Denable_telemetry_traceback=true
$ ninja -C /tmp/build
```

6.31.2.5.2.2 Option 2: Configuration File

You can persistently isolate the build by editing the **meson_options.txt** file directly.

1. Open **/opt/mellanox/doca/applications/meson_options.txt** and set the following parameters:

- `enable_all_applications` = `false`
- `enable_telemetry` = `true`
- `enable_telemetry_traceback` = `true`

> **Configuration Rule**
>
> If `enable_all_applications` is set to `false`, the main `enable_telemetry` flag must be set to `true` for any underlying telemetry application to compile successfully. The individual applications default to `true`, but you can toggle them on or off as needed once the main telemetry flag is enabled.

2. Once the file is saved, run the standard compilation commands:

```
$ cd /opt/mellanox/doca/applications/
$ meson /tmp/build
$ ninja -C /tmp/build
```

6.31.2.6 Running the Application

6.31.2.6.1 Application Execution

The DOCA Telemetry Traceback application is distributed as source code and must be compiled prior to execution.

Execution syntax:

```
$ doca_telemetry_traceback [DOCA Flags] [Program Flags]
```

Execution example:

```
$ sudo doca_telemetry_traceback -d 03:00.0 --rules my_rules_file.json
```

> **Root Privileges**
>
> The application strictly requires `sudo` (root privileges) to access hardware-level telemetry data on the device.

> **PCIe Addressing**
>
> Ensure the `-d` parameter (e.g., `03:00.0`) exactly matches the physical PCIe address of your target device.

6.31.2.6.2 Command Line Flags

6.31.2.6.2.1 General Flags

Short Flag	Long Flag	Description
`-h`	`--help`	Prints a help synopsis and exits
`-v`	`--version`	Prints program version information and exits
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

6.31.2.6.2.2 Program Flags

Short Flag	Long Flag	Description
`-d`	`--device`	Mandatory flag. The exact PCIe address of the device to read telemetry information from.
`-r`	`--rules`	The path to a JSON file containing the application configuration. See section "Traceback Configuration File".
N/A	`--event-log`	Path to output log file which will be populated with events information
N/A	`--force-ownership`	Force ownership of the diag domain. Only applies when the device firmware is in single domain mode.

Unsupported Data IDs

It is not guaranteed that all data IDs supported by the application are supported by your specific device. The application will report any unsupported IDs, requiring administrators to adjust their inputs accordingly.

6.31.2.6.3 Troubleshooting

Refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue encountered with the compilation, installation, or execution of the DOCA applications.

6.31.2.7 Application Code Flow

The DOCA Telemetry Traceback application executes in three distinct operational phases: argument parsing, component initialization, and a continuous data polling loop.

6.31.2.7.1 Phase 1: Argument Parsing

The application initializes the logger and parses both CLI arguments and JSON configurations to establish its operational parameters.

- **create_doca_logger_backend()** : Creates the logger backend to display system messages.
- **parse_configuration()** : Parses the CLI arguments. Values unique to the application instance are passed via the CLI, while values shared across a multi-server deployment are deferred to the JSON file.
- **load_file_into_buffer()** and **parse_json_configuration()** : Loads the shared configuration from the defined JSON file (defaults to **rules.json** if not overridden via CLI).
- **validate_configuration()** : Performs initial validity checks to catch expected configuration errors before allocating hardware resources.

6.31.2.7.2 Phase 2: Component Initialization

The application prepares its memory buffers, defines its rule objects, and establishes the connection to the DOCA Telemetry Diag context.

6.31.2.7.2.1 Data and Logging Preparation

- History buffer: Calls **init_data_aligned_ring()** to prepare the main circular data store. The buffer is deliberately sized to double the requested depth, ensuring there is exactly as much history available as there is active data.
- Rule objects: Populates a C++ vector with rule objects.

> **Memory Optimization**
>
> The rules vector is specifically sized using **reserve** because the correct size is known upfront, allowing it to be sized correctly with a single memory allocation to prevent fragmentation.

> **Data Indexing**
>
> For each rule, the application calculates an array index (**sample_record_offset**) of **1 + N** . This explicit math is required because the application hardcodes the **DOCA_TELEMETRY_DIAG_OUTPUT_FORMAT_1** format, which produces a timestamp followed by **N** values.

- Events log: Creates or truncates the **events.log** file using **std::ofstream** and immediately writes the application configuration to the header via **log_app_start_header()** .

6.31.2.7.2.2 Hardware and Telemetry Initialization

- Device open: Identifies and opens the requested DOCA device using **open_doca_dev()**.

> ⚠ The device pointer is stored in a RAII **unique_ptr** to guarantee automatic cleanup upon application exit without requiring complex **goto** trees.

- Context creation: Verifies that the device supports telemetry data (requiring elevated permissions), creates the context via **create_and_prepare_diag_ctx()**, and starts the data sampling process with **doca_telemetry_diag_start()**.
- Context validation: Calls **doca_telemetry_diag_get_sample_size()** and **doca_telemetry_diag_get_sample_period()** to confirm the hardware accepted the requested parameters.

> ℹ If the actual sample size or period does not match the requested parameters, the application explicitly stops. While this could be worked around, the codebase relies on these assumptions for simplicity and requires them to be correct.

6.31.2.7.3 Phase 3: Main Execution Loop

The application enters a continuous polling state, checking the telemetry instance for new data, updating the circular buffer, and evaluating rule logic.

6.31.2.7.3.1 Setup and Signal Handling

- Allocates a small read buffer (**sample_buffer**) to ingest data from the telemetry instance.

> ℹ Data is read here, evaluated, mutated, and then copied into the history buffer. This decoupled design future-proofs the application, allowing users to read from multiple telemetry sources where the formats between the read buffer and history buffer could differ significantly.

- Installs a **SIGINT** signal handler allowing administrators to terminate the process cleanly via **CTRL+C**.

6.31.2.7.3.2 Datapath Loop

- Data polling: Calls **doca_telemetry_diag_query_counters()**. If no data is available (0 samples), the thread sleeps for **diag_sample_period_ns** to prevent CPU exhaustion. Typically, 1 sample is read, though 2+ may be read if the application experiences wake-up latency.
- Sample processing:
  - Casts the read pointer to **doca_telemetry_diag_data_sample_format_1**.
  - Advances the circular buffer head (**advance_head_idx()**) and copies the hardware timestamp and data values into the new history record.

> Bit manipulation is applied here to accommodate the `DOCA_TELEMETRY_DIAG_TIMESTAMP_SOURCE_RTC` format.

- If any rules are currently active, the sample is immediately written to the events log file.

6.31.2.7.3.3 Rule Evaluation and State Changes

Each rule evaluates the new sample using `monitor.process_sample()`.

If a rule does not change state during the processing of this new sample, there is no more work to be done, and the application immediately moves to process the next rule.

Rule Activates	The application increments `active_monitors_count`. • If this is the first rule activating, the application writes the data column headers and dumps the historical buffer to the events log. • For all rule activations (whether the first or the Nth rule), an "event started" message is explicitly written to the log. **Note on Sequencing** All samples read before the rule activated have already been written to the history. Data received while any rule is active is logged strictly before the rules processing step for each sample, continuing until all rules deactivate.
Rule Deactivates	Decrements `active_monitors_count` and writes an "event completed" marker. If the count reaches 0, no further data is written to the log file until a rule activates again in the future.

> Because device cleanup is handled via C++ RAII `unique_ptr` objects, all resources are safely released automatically when the `SIGINT` signal breaks the execution loop

6.31.2.8 References

- `/opt/mellanox/doca/applications/telemetry/traceback/`

7 DOCA Tools

DOCA tools are a set of executables/scripts that are needed to produce inputs to some of the DOCA libraries and applications.

All tools are installed with DOCA, as part of the doca-tools package, and can either be directly accessed from the terminal or can be found at `/opt/mellanox/doca/tools`. Refer to DOCA Installation Guide for Linux for more information.

> For questions, comments, and feedback, please contact us at doca-feedback@nvidia.com.

7.1 DOCA Bench

7.1.1 Introduction

NVIDIA DOCA Bench allows users to evaluate the performance of DOCA applications, with reasonable accuracy for real-world applications. It provides a flexible architecture to evaluate multiple features in series with multi-core scaling to provide detailed throughput and latency analysis.

This tool can be used to evaluate the performance of multiple DOCA operations, gain insight into each stage in complex DOCA operations and understand how items such as buffer sizing, scaling, and GGA configuration affect throughput and latency.

7.1.2 Feature Overview

DOCA Bench is designed as a unified testing tool for all BlueField accelerators. It, therefore, provides these major features:

- BlueField execution, utilizing the Arm cores and GGAs "locally"
- Host (x86) execution, utilizing x86 cores and the GGAs on the BlueField over PCIe
- Support for the following DOCA/DPU features:
  - DOCA AES GCM
  - DOCA Comch
  - DOCA Compress
  - DOCA DMA
  - DOCA EC
  - DOCA Eth
  - DOCA RDMA
  - DOCA SHA
  - DOCA GPUNetIO
- Multi-core/multi-thread support
- Schedule executions based on time, job counts, etc.
- Ability to construct complex pipelines with multiple GGAs (where data moves serially through the pipeline)
- Various data sources (random data, file data, groups of files, etc.)
- Remote memory operations

    - Use data location on the host x86 platform as input to GGAs
- Comprehensive output to screen or CSV
- Query function to report supported software and hardware feature
- Sweeping of parameters between a start and end value, using a specific increment each time
- Specific attributes can be set per GGA instance, allowing fine control of GGA operation
- Batching of jobs

7.1.3 Installation

DOCA Bench is installed and available in both DOCA-for-Host and DOCA BlueField Arm packages. It is located under the **/opt/mellanox/doca/tools** folder.

7.1.3.1 Prerequisites

DOCA 2.7.0 and higher.

7.1.3.2 Granular Build Support

DOCA Bench supports a granular build environment which allows users to determine which DOCA libraries are installed on any target system. During initialization, DOCA Bench probes all available and supported DOCA libraries, and provides the ability to test those libraries. For example, if the DOCA SHA library is not present then DOCA Bench does not allow SHA to be tested.

DOCA Bench provides a query system where device capabilities can be queried to see if the library is indeed installed and supported (under the "installed : yes / no" section of each library). Please see section "DOCA Bench#Queries" for details.

7.1.4 Operating Modes

DOCA Bench measures performance of either throughput (bandwidth) or latency.

7.1.5 Throughput Measurements

In this mode, DOCA Bench measures the maximum performance of a given pipeline (see "DOCA Bench#Core Principles"). At the end of the execution, a short summary along with more detailed statistics is presented:

```
Aggregate stats
        Duration:      3000049 micro seconds
        Enqueued jobs: 17135128
        Dequeued jobs: 17135128
        Throughput:    5712042 Operations/s
        Ingress rate:  063.832 Gib/s
        Egress rate:   063.832 Gib/s
```

7.1.5.1 Latency Measurements

Latency is the measurement of time taken to perform a particular operation. In this instance, DOCA Bench measures the time taken between submitting a job and receiving a response.

DOCA Bench provides two different types of latency measurement figures:

- Bulk latency mode – attempts to submit a group of jobs in parallel to gain maximum throughput, while reporting latency as the time between the first job submitted in the group and the last job received.
- Precision latency mode – used to ensure that only one job is submitted and measured before the next job is scheduled.

7.1.5.1.1 Bulk Latency

This latency mode effectively runs the pipelines at full rate, trying to maintain the maximum throughput of any pipeline while also recording latency figures for jobs submitted.

To record latency, while operating at the pipeline's maximum throughput, users must place the latency figures inside groups or "buckets" (rather than record each individual job latency). Using this method, users can avoid the large memory and CPU overheads associated with recording millions of latency figures per second (which would otherwise significantly reduce the performance).

As each pipeline operation is different, and therefore has different latency characteristics, the user can supply the boundaries of the latency measure. DOCA Bench internally creates 100 buckets, of which the user can specify the starting value and the width or size of each bucket. The first and last bucket have significance:

- The first bucket contains all jobs that executed faster than the starting period
- The last bucket contains a count of jobs that took longer than the maximum time allowed

The command line option **`--latency-bucket-range`** is used to supply two values representing the starting time period of the first bucket, and the width of each sequential bucket. For example, **`--latency-bucket-range 10us,100us`** would start with the lowest bucket measuring <10µs response times, then 100 buckets which are 100µs wide, and a final bucket for results taking longer than 10010µs.

The report generated by bulk mode visualizes the latency data in two methods:

1. A bar graph is provided to visually show the spread of values across the range specified by the **`--latency-bucket-range`** option:

```
Latency report:
                    :
                    :
                    :
                    :
                    :
                    ::
                    ::
                    ::
                    ::
                   .:::.  .    .    ..
-----------------------------------------------------------------------------------------------------
```

2. A breakdown of the number of jobs per bucket is presented. This example shortens the output to show that the majority of values lie between 27000ns and 31000ns.

```
[25000ns -> 25999ns]: 0
[26000ns -> 26999ns]: 0
[27000ns -> 27999ns]: 128
[28000ns -> 28999ns]: 2176
[29000ns -> 29999ns]: 1152
[30000ns -> 30999ns]: 128
[31000ns -> 31999ns]: 0
[32000ns -> 32999ns]: 0
[33000ns -> 33999ns]: 128
```

```
[34000ns -> 34999ns]: 0
[35000ns -> 35999ns]: 0
```

7.1.5.1.2 Precision Latency

This latency mode operates on a single job at a time. At the cost of greatly reduced throughput, this allows the minimum latency to be precisely recorded. As shown below, the statistics generated are precise and include various fields such as min, max, median, and percentile values.

```
Aggregate stats

        ....
        min:            1878 ns
        max:            4956 ns
        median:         2134 ns
        mean:           2145 ns
        90th %ile:      2243 ns
        95th %ile:      2285 ns
        99th %ile:      2465 ns
        99.9th %ile:    3193 ns
        99.99th %ile:   4487 ns
```

7.1.6 Core Principles

The following subsections elaborate on principles which are essential to understand how DOCA Bench operates.

7.1.6.1 Host or BlueField Arm Execution

Whether executing DOCA Bench on an x86 host or BlueField Arm, the behavior of DOCA Bench is identical. The performance measured is dependent on the environment.

> Only execution on x86 hosts is supported.

7.1.6.2 Pipelines

DOCA Bench is a highly flexible tool, providing the ability to configure how and what operations occur and in what order. To accomplish this, DOCA Bench uses a pipeline of operations, which are termed "steps". These steps can be a particular function (e.g., Ethernet receive, SHA hash generation, data compression). Therefore, a pipeline of steps can accomplish a number of sequential operations. DOCA Bench can measure the throughput performance or latency of these pipelines, whether running on single or multiple cores/threads.

> Currently, DOCA supports running only one pipeline at a time.

7.1.6.3 Warm-up Period

To ensure correct measurement, the pipelines must be run "hot" (i.e., any initial memory, caches, and hardware subsystems must be running prior to actual performance measurements begin). This is known as the "warm-up" period and, by default, runs approximately 250 jobs through the pipeline before starting measurements.

7.1.6.4 Defaults

DOCA Bench has a large number of parameters but, to simplify execution, only a few must be supplied to commence a performance measurement. Therefore various parameters have defaults which should be sufficient for most cases. To fine tune performance, users should pay close attention to any default parameters which may affect their pipeline's operation.

> When executed, DOCA Bench reports a full list of all parameters and configured values.

7.1.6.5 Optimizing Performance

To obtain maximum performance, a certain amount of tuning is required for any given environment. While outside the scope of this documentation, it is recommended for users to:

- Avoid using CPU 0 as most OS processes and interrupt request (IRQ) handlers are scheduled to execute on this core
- Enable CPU/IRQ isolation in the kernel boot parameters to remove kernel activities from any cores they wish to execute performance tests on
- On hosts, ensure to not cross any non-uniform memory access (NUMA) regions when addressing the BlueField
- Understand the memory allocation requirements of scenarios, to avoid over-allocating or running into near out-of-memory situations

7.1.7 Supported BlueField Feature Matrix

DOCA Bench can be executed on both host and BlueField Arm environments, and can target BlueField networking platforms.

The following table shows which operations are possible using either DOCA Bench. It also provides two columns showing whether remote memory can be used as an input or output to that operation. For example, DMA operations on the BlueField Arm can access remote memory as an input to pull memory from the host into the BlueField Arm).

	BlueField-2 Networking Platform	BlueField-3 Networking Platform	Execute on Host Side	Execute on BlueField Arm	Remote Memory as Input Allowed?	Remote Memory as Output Allowed?
doca_compress::compress	✓		✓	✓	✓	✓
doca_compress::decompress	✓	✓	✓	✓	✓ Input remote memory is not supported for lz4 decompression.	✓

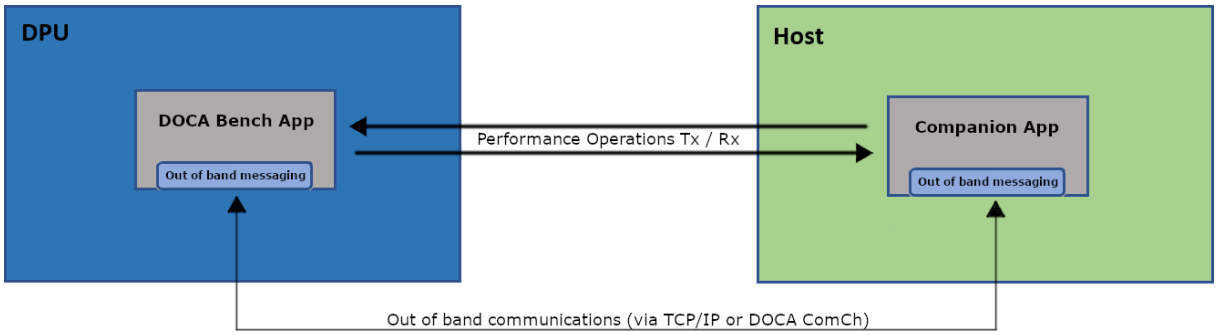
doca_dma	✓	✓	✓	✓	✓	✓
doca_ec::create		✓	✓	✓	✓	✓
doca_ec::recover		✓	✓	✓	✓	✓
doca_ec::update		✓	✓	✓	✓	✓
doca_sha	✓		✓	✓	✓	
doca_rdma::send	✓	✓	✓	✓	✓	✓
doca_rdma::receive	✓	✓	✓	✓	✓	✓
doca_aes_gcm::encrypt		✓	✓	✓	✓	✓
doca_aes_gcm::decrypt		✓	✓	✓	✓	✓
doca_cc::client_producer	✓	✓	✓	✓		
doca_cc::client_consumer	✓	✓	✓	✓		
doca_eth::rx	✓	✓	✓			
doca_eth::tx	✓	✓	✓			
doca_gpunetio::eth_tx	✓	✓	✓			
doca_gpunetio::eth_rx	✓	✓	✓			
doca_gpunetio::eth_bidir	✓	✓	✓			

7.1.8 Remote Operations

A subset of BlueField operations have a remote element, whether this is an RDMA connection, Ethernet connectivity, or memory residing on an x86 host. All these operations require an agent to be present on the far side to facilitate the benchmarking of that particular feature.

In DOCA Bench, this agent is an additional standalone application called the "companion app". It provides the remote benchmarking facilities and is part of the standard DOCA Bench installation.

The following diagram provides an overview of the function and communications between DOCA Bench and the companion app:



In this particular setup, the BlueField executes "DOCA Bench" while the host (x86) is executes the companion App.

DOCA Bench also acts as the controller of the tests, instructing the companion app to perform the necessary operations as required. There is an out-of-band communications channel operating between the two applications that utilizes either standard TCP/IP sockets or a DOCA Comch channel (depending on the test scenario/user preferences).

DOCA Bench tool is not intended to be used in a production deployment. If choosing to do so, please be aware that the out-of-band communications might contain sensitive information and thus should be done over a secure channel when using the standard TCP/IP sockets

7.1.9 CPU Core and Thread Selection

Selection of the correct CPU cores and threads has a significant impact on the performance or latency obtained. Read this section carefully.

A key requirement to scale any application is the number of CPU cores or threads allocated to any given activity. DOCA Bench provides the ability to specify the numbers of cores, and the number of threads to be created per core, to maximize the number of jobs submitted to a given pipeline.

The following care should be given when selecting the number of CPU's or threads:

- Threads that are on cores located on distant NUMA regions (i.e., not the same NUMA region the BlueField is connected to) will experience lower performance and higher latency
- Core 0 is often most used by the OS and should be avoided
- Standard Linux Kernel installations allow the OS to move processes on any CPU core resulting in unexpected drops in performance, or higher latency, due to process switching

The selection of CPU cores is provided through the `--core-mask`, `--core-list`, `--core-count` parameters, while thread selection is made via the `--threads-per-core` parameter.

7.1.10 Device Selection

7.1.10.1 BlueField Devices

When executing from a host (x86) environment DOCA Bench can target one or more BlueField devices within an installed environment. When executing from the BlueField Arm, the target is always the local BlueField.

The default method of targeting a given BlueField from either the host or the BlueField Arm is using the `--device` or `-A` parameters, which can be provided as:

- Device PCIe address (i.e., `03:00.0`);
- Device IB name (`mlx5_0`); or
- Device interface name (`ens4f0`)

From the BlueField Arm environment, DOCA Bench should be targeted at the local PCIe address (i.e., `--device 03:00.0`) or the IB device name (i.e., `mlx5_0`).

7.1.10.2 GPU Devices

When executing a DOCA GPUNetIO benchmark, a GPU device must be specified. This can be done using the `--gpu` or `-G` parameters, which can only be provided as a device PCIe address (i.e., `03:00.0`).

7.1.11 Input Data Selection and Sizing of Jobs

DOCA Bench supports different methods of supplying data to jobs and providing information on the amount of data to process per job. These are referred to as "Data Providers".

7.1.11.1 Input Data Selection

The following subsections provide the modes available to provide data for input into any operation.

7.1.11.1.1 File

A single file is used as input to the operation. The contents of the file are not important for certain operations (e.g., DMA, SHA, etc.) but must be valid and specific for others (e.g., decompress, etc). The data may be used multiple times and repeated if the operations required more data than the single file contains. For more information on how file data is handled in complex operations, see section "DOCA Bench#Command-line Parameters".

7.1.11.1.2 File Sets

File sets are a group of files that are primarily used for structured data. The data in the file set is effectively a list of files, separated by a new line that is used sequentially as input data for jobs. Each file pointed to by the file set would have its entire contents read into a single buffer. This is

useful for operations that require structured data (i.e., a complete valid block of data, such as decompression or AES).

7.1.11.1.3 Random Data

Random data is provided when the actual data required for the given operation is not specific (e.g., DMA).

> ⚠ The use of random data for certain operations may reduce the maximum performance obtained. For example, compressing random data results in lower performance than compressing actual file data (due to the lack of repeating patterns in random data).

7.1.11.2 Job Sizing

Each job in DOCA Bench consists of three buffers: An original input buffer, an output, and an intermediate buffer.

The input buffer is provided by the data provider for the first step in the pipeline to use, after which the following steps use the output and intermediate buffers (can be sized by using `--job-output-buffer-size`) in a ping-pong fashion. This means, the pipeline can always start with the same deterministic data while allowing for each step to provide its newly generated output data to be used as input to the next step.

The input buffer is specified in one of two ways: using uniform-job-size to make every input buffer the exact same size, or using a file set to size each buffer based on the size of the selected input data file(s). Users should ensure the data generated by each step in the pipeline will fit in the provided output buffer.

7.1.12 Controlling Test Duration

DOCA Bench has a variety of ways to control the length of executing tests—whether based on data or time limit.

7.1.12.1 Limit to Specific Number of Seconds

Using the `--run-limit-seconds` or `-s` parameter ensures that the execution continues for a specific number of seconds.

7.1.12.2 Limited Through Total Number of Jobs

It may be desirable to measure a specific number of jobs passing through a pipeline. The `--run-limit-jobs` or `-J` parameter is used to specify the exact number of jobs submitted to the pipeline and allowed to complete before execution finishes.

7.1.13 GGA-specific Attributes

As DOCA Bench supports a wide range of both GGA and software based DOCA libraries, the ability to fine tune their invocation is important. Command-line parameters are generally used for

configuration options that apply to all aspects of DOCA Bench, without being specific to a particular DOCA library.

Attributes are the method of providing configuration options to a particular DOCA Library, whilst some shared attributes exist the majority of libraries have specific attributes designed to control their specific behavior.

For example, the attribute **doca_ec.data_block_count** allows you to set the data block count for the DOCA EC library, whilst the attribute **doca_sha.algorithm** controls the selection of the SHA algorithm.

For a full list of support attributes, see the "DOCA Bench#Command-line Parameters" section.

Due to batching it is possible that more than the supplied jobs are executed.

7.1.14 Command-line Parameters

DOCA Bench allows users to specify a series of operations to be performed and then scale that workload across multiple CPU cores/threads to get an estimation of how that workload performs and some insight into which stage(s), if any, cause performance problems for them. The user can then modify various configuration properties to explore how issues can be tuned to better serve their need.

When running, DOCA Bench creates a number of execution threads with affinities to the specific CPU specified by the user. Each thread creates, uniquely for themselves, a jobs pool (with job data initialized by a data provider) and a pipeline of workload steps.

7.1.14.1 CPU Core and Thread Count Configuration

There are many factors involved when carrying out performance tests, one of these is the CPU selection:

- The user should consider NUMA regions when selecting which cores to use, as using a CPU which is distant from the device under test can impact the performance achievable
- The user may also wish to avoid core 0 as this is typically the default core for kernel interrupt handlers.

CPU core selection has an impact on the total memory footprint of the test. See section "DOCA Bench#Test Memory Footprint" for more details.

7.1.14.1.1 --core-mask

Default value: 0x02

Core mask is the simplest way to specify which cores to use but is limited in that it can only specify up to 32 CPUs (0-31). Usage example: **--core-mask 0xF001** selects CPU cores 0, 12, 13, 14, and 15.

7.1.14.1.2 --core-list

Core list can specify any/all CPU cores in a given system as a list, range, or combination of the two. Usage example: `--core-list 0,3,6-10` selects CPU cores 0, 3, 6, 7, 8, 9, and 10.

7.1.14.1.3 --core-count

The user can select the first N cores from a given core set (list or mask) if desired. Usage example: `--core-count N`.

> Sweep testing is supported. See section "DOCA Bench#Sweep Tests" for more details.

7.1.14.1.4 --threads-per-core -t

To test the impacts of contention within a single CPU core, the user can specify this value so that instead of only one thread being created per core, N threads are created with their affinity mask set to the given core for each core selected. For example, 3 cores and 2 threads per core create 6 threads total.

> Sweep testing is supported. See section "DOCA Bench#Sweep Tests" for more details.

7.1.14.2 Device Configuration

The test requires the use of at least one BlueField to execute. With remote system testing, a second device may be required.

7.1.14.2.1 --device -A

Specify the device to use from the perspective of the system under test. The value can be for any one of either the device PCIe address (e.g., `03:00.0`), the device IB device name (e.g., `mlx5_0`), or the device interface name (e.g., `ens4f0`).

7.1.14.2.2 --representor -R

This option is used only when performing remote memory operations between a BlueField device and its host using DOCA Comch. This is typically automated by the companion connection string but exists for some developer debug use-cases.

> This option used to be important before the companion connection string property was introduced but now is rarely used.

7.1.14.3 Input Data and Buffer Size Configuration

DOCA Bench supports multiple methods of acquiring data to use to initialize job buffers. The user can also configure the output/intermediate buffers associated with each job.

Input data and buffer size configuration has an impact on the total memory footprint of the test. See section "DOCA Bench#Test Memory Footprint" for more details.

7.1.14.3.1 --data-provider -I

DOCA Bench supports several different input data sources:

- `file`
- `file-set`
- `random-data`

7.1.14.3.1.1 File Data Provider

The file data provider produces uniform/non-structured data buffers by using a single input file. The input data is stripped and or repeated to fill each data buffer as required, returning back to the start of the file each time it is exhausted to collect more data. This is desirable when the performance of the component(s) under test is meant to show different performance characteristics depending on the input data supplied.

For example, `doca_dma` and `doca_sha` would execute in constant time regardless of the input data. Whereas `doca_compress` would be faster with data with more duplication and slower for truly random data and would produce different output depending on the input data.

Example 1 – Small Input File with Large Buffers

Given a small input data (i.e., smaller than the data buffer size), the file contents are repeated until the buffer is filled and then continue onto the next buffer(s). So, if the input file contained the data `012345` and the user requested two 20-byte buffers, the buffers would appear as follows:

- `01234501234501234501`
- `23450123450123450123`

Example 2 – Large Input File with Smaller Buffers

Given a large input data (i.e., greater than the data buffer size), the file contents are distributed across the data buffers. If the the input file contained the data `0123456789abcdef` and the user requested three 12-byte buffers, the buffers would appear as follows:

- `0123456789ab`
- `cdef01234567`
- `89abcdef0123`

7.1.14.3.1.2 File Set Data Provider

The file set data provider produces structured data. The file set input file itself is a file containing one or more filenames (relative to the input `"command working directory (cwd)"` not relative to the file set file). Each file listed inside the file set would have its entire contents used as a job buffer. This is useful for operations where the data must be a complete valid data block for the operation to succeed like decompression with `doca_compress` or decryption with `doca_aes`.

Example – File Set and Its Contents

Given a file set in the **"command working directory (cwd)"** referring to **data_1.bin** and **data_2.bin** (one file name per line), and **data_1.bin** contains 33 bytes and **data_2.bin** contains 69 bytes, then the data required by the buffers would be filled with these two files in a round-robin manner until the buffers are full. Unlike uniform (non-structured) data each task can have different lengths.

7.1.14.3.1.3 Random-data Data Provider

The random data data provider provides uniform (non-structured) data from a random data source. Each buffer will have unique (pseudo) random bytes of content.

7.1.14.3.2 --data-provider-job-count

Default value: 128

Each thread in DOCA Bench has its own allocation of job data buffers to avoid memory contention issues. Users may select how many jobs should be created per thread using this parameter.

> Sweep testing is supported. See section "DOCA Bench#Sweep Tests" for more details.

7.1.14.3.3 --data-provider-input-file

For data providers which use an input file, the filename can be specified here. The filename is relative to the **input_cwd**.

> Sweep testing is supported. See section "DOCA Bench#Sweep Tests" for more details.

7.1.14.3.4 --uniform-job-size

Specify the size of uniform input buffers (in bytes) that should be created.

> Does not apply and should not be specified when using structured data input sources.

> Sweep testing is supported. See section "DOCA Bench#Sweep Tests" for more details.

7.1.14.3.5 --job-output-buffer-size

Default value: 16384

Specify the size of output/intermediate buffers (in bytes). Each job has 3 buffers: immutable input buffer and two output/intermediate buffers. This allows for a pipeline to mutate the data an infinite number of times throughout the pipeline, while allowing for it to be reset and re-used at the end and allowing any step to use the new mutated data created by the previous step.

7.1.14.3.6 --input-cwd -i

To ease configuration management, the user may opt to use a separate folder for the input data for a given scenario outside of the DOCA build/install directory.

> It is recommended to use relative file paths for the input files.

7.1.14.3.6.1 Example 1 – Running DOCA Bench from Current Working Directory

Considering a user executing DOCA Bench from **/home/bob/doca/build**, values specified in **--data-provider-input-file** and filenames within a file set would search relative to the shell's "command working directory (cwd)": **/home/bob/doca/build**. Their command might look something like:

```
doca_bench --data-provider file-set --data-provider-input-file my_file_set.txt
```

And assuming **my_file_set.txt** contains **data_1.bin**, the files that would be loaded by DOCA Bench after path resolution would be:

- **/home/bob/doca/build/my_file_set.txt**
- **/home/bob/doca/build/data_1.bin**

7.1.14.3.6.2 Example 2 – Running DOCA Bench from Another Directory

Considering the user executed that same test from one level up. Something like:

```
build/doca_bench --data-provider file-set --data-provider-input-file build/my_file_set.txt
```

The files to be loaded would be:

- **/home/bob/doca/build/my_file_set.txt**
- **/home/bob/doca/data_1.bin**

Notice how both files were loaded relative to the **"command working directory (cwd)"** and the data file was not loaded relative to the file set.

7.1.14.3.6.3 Example 3 – Example 2 Revisited Using input-cwd

The user can solve this easily by keeping all input files in a single directory and then referring to that directory using the parameter **input-cwd**. In this case, the command like may look something like:

```
build/doca_bench --data-provider file-set --data-provider-input-file my_file_set.txt --input-cwd build
```

Note that the value for **--data-provider-input-file** also changed to be relative to the new **"command working directory (cwd)"**.

The files loaded this time are back to being what is expected:

- `/home/bob/doca/build/my_file_set.txt`
- `/home/bob/doca/build/data_1.bin`

7.1.14.4 Test Execution Control

DOCA Bench supports multiple test modes and run execution limits to allow the user to configure the test type and duration.

7.1.14.4.1 --mode

Default value: throughput

Select which type of test is to be performed.

7.1.14.4.1.1 Throughput Mode

Throughput mode is optimized to increase the volume of data processed in a given period with little or no regard for latency impact. Throughput mode tries to keep each component under test as busy as possible. A summary of the bandwidth and job execution rate are provided as output.

7.1.14.4.1.2 Bulk-latency Mode

Bulk latency mode strikes a balance between throughout and latency, submitting a batch of jobs and waiting for them all to complete to measure the latency of each job. This mode uses a bucketing mechanism to allow DOCA Bench to handle many millions of jobs worth of results. DOCA Bench keeps a count of the number of jobs that complete within each bucket to allow it to run for long periods of time. A summery of the distribution of results with an ASCII histogram of the results are provided as output. The latency reported is the time taken between the first job submission (for a batch of jobs) until the final job response is received (for that same batch of jobs).

7.1.14.4.1.3 Precision-latency Mode

Precision latency mode executes one job at a time to allow DOCA Bench to calculate the minimum possible latency of the jobs. This causes the components which can process many jobs in parallel to be vastly underutilized and so greatly reduces bandwidth. As this mode records every result individually, it should not be used to execute more than several thousand jobs. Precision latency mode requires 8 bytes of storage for each result, so be mindful of the memory overhead of the number of jobs to be executed.

A statistical analysis including minimum, maximum, mean, median and some percentiles of the latency value are provided as output.

7.1.14.4.2 --latency-bucket-range

Default value: 100ms,10ms

Only applicable to bulk-latency mode. Allows the user to specify the starting value of the buckets, and the width of each bucket. There are 100 buckets of the given size and an under flow and overflow bucket for results that fall outside of the central range.

For example:

```
--latency-bucket-range 10us,100us
```

This would start with the lowest bucket measuring <10µs response times, then 100 buckets which are 100µs wide, and a final bucket for results taking longer than >10010µs.

7.1.14.5 Blocking Mode

DOCA supports two methods of waiting on completion of tasks:

- Busy-wait (or polling) mode
- Notification-driven mode

> Refer to "DOCA SDK Architecture" documentation for more information.

By default, DOCA Bench uses the busy-wait to ensure maximum bandwidth (and low latency) for any given pipeline and its tasks with high utilization of any allocated CPU resources.

As with all high-performance software, utilizing GGAs or hardware accelerators, performance is usually CPU-bound at smaller packet sizes (i.e., at smaller payload sizes, the CPU spends a long time generating tasks and dealing with completions). For larger packet sizes, the CPU submits less tasks, as each task contains more data, therefore it may easily submit more data than the GGA or hardware accelerator can accept, resulting in periods where the CPU is busy-waiting on completions before being able to submit further tasks.

> To execute any tests using an "notification-driven mode", use the options detailed in the following subsections.

7.1.14.5.1 --use-blocking-mode

This option causes DOCA Bench to use the "notification-drive mode" method of waiting on task completion.

> At smaller packet sizes, the benchmark may still be CPU bound.

7.1.14.5.2 --record-cpu-usage

If specified, this option reports CPU statistics for any CPU cores DOCA Bench is executing on. This provides guidance on how much CPU time is returned, and thus available to other processes or threads, should the "notification-driven" mode be active.

> Short duration tests may not result in sufficient produced data to generate CPU usage statistics.

The statistics provided include min, max, median, and mean values for the CPU usage. Also included are a number of percentile results, showing 90th, 95th, and a number of 99th percentile values. Example output:

```
CPU Usage stats
        min:            25%
        max:            50%
        median:         50%
        mean:           45.8333%
        90th %ile:      50%
        95th %ile:      50%
        99th %ile:      50%
        99.9th %ile:    50%
        99.99th %ile:   50%
```

7.1.14.6 Execution Limits

By default, a test runs forever. This is typically undesirable so the user can specify a limit to the test.

> ⚠ Precision-latency mode only supports job limited execution.

7.1.14.6.1 --run-limit-seconds -s

Runs the test for N seconds as specified by the user.

7.1.14.6.2 --run-limit-jobs -J

Runs the test until at least N jobs have been submitted, then allowing in-flight jobs to complete before exiting. More jobs than N may be executed based on batch size.

7.1.14.6.3 --run-limit-bytes -b

Runs the test until at least N bytes of data have been submitted, then allowing in-flight jobs to complete before exiting. More data may be processed than desired if the limit is not a multiple of the job input buffer size.

7.1.14.7 Batching Support

DOCA Bench supports submission of jobs in batches to improve performance.

7.1.14.7.1 --batch-mode

Default value: none

Specifies the batching mode to use. The following options are available:

- **`--batch-mode none`** – no batching, ring the doorbell after each job is submitted
- **`--batch-mode batch-submit`** – submit a batch of jobs and then ring the doorbell. The number of jobs to be submitted is specified by the **`--batch-size`** parameter. If the number of jobs left to be submitted is fewer than **`batch-size`**, then the doorbell is rung after the last job is submitted.
- **`batch-submit-minimal-reports`** – submit a batch of jobs and ring the doorbell, but defer the completion callback of the job until a later time. Reduces the number of completion notifications received after submission of a batch of jobs by receiving a single completion notification for the entire batch.

7.1.14.7.2 --batch-size

Default value: 1

Specifies the number of jobs to be included in each batch before ringing the doorbell.

> ⚠ Batching is currently only supported for the **doca_sha** and **doca_dma** step types and can only be specified in the throughput mode of operation.

7.1.14.8 Gather/Scatter Support

Gather support involved breaking incoming input data from a single buffer into multiple buffers, which are "gathered" into a single gather list. Currently only gather is supported.

7.1.14.8.1 --gather-value

Default value: 1

Specifies the partitioning of input data from a single buffer into a gather list. The value can be specified in two flavors:

- **--gather-value 4** – splits input buffers into 4 parts as evenly as possible with odd bytes in the last segment
- **--gather-value 4KiB** – splits buffers after each 4KB of data. See **doca_bench/utility/byte_unit.hpp** for the list of possible units.

7.1.14.8.2 --scatter-value

Default value: 1

Specifies the partitioning of output data into a scatter list. The value can be specified in two flavors:

- **--scatter-value 4** – splits output buffers into 4 parts as evenly as possible with odd bytes in the last segment
- **--scatter-value 4KiB** – splits buffers after each 4KB of data. See **doca_bench/utility/byte_unit.hpp** for the list of possible units.

7.1.14.9 Stats Output

7.1.14.9.1 --rt-stats-interval

By default, DOCA Bench emits the results of an iteration once it completes. The user can ask for transient snapshots of the stats as the test progresses by providing the **--rt-stats-interval** argument with a value representing the number of milliseconds between stat prints. The end-result of the run is still displayed as normal.

⚠ This may produce a large amount of console output.

7.1.14.9.2 --csv-output-file

DOCA Bench can produce an output file as part of its execution which can contain stats and the configuration values used to produce that stat. This is enabled by specifying the **--csv-output-file** argument with a file path as the value. Providing a value for this argument enables CSV stats output (in addition to the normal console output). When performing a sweep test, one line per iteration of the sweep test is populated.

By default, the CSV output contains every possible value. The user can tune this by applying a filter.

7.1.14.9.3 --csv-stats

Provide one or more filters (positive or negative) to tune which stats are displayed. The value for this argument is a comma-separated list of filter strings. Negative filters start with a minus sign ('`-`').

7.1.14.9.3.1 Example 1 – Emit Only Statistical Values (No Configuration Values)

```
--csv-stats "stats.*"
```

⚠ The quotes around the `*` prevent the shell from interpreting it as a wild card for filenames in the command.

7.1.14.9.3.2 Example 2 – Emit Statistical Values and Some Configuration Values (Remove Attribute Values)

```
--csv-stats "stats.*,-attribute*
```

7.1.14.9.4 --csv-append-mode

Default: **false**

When enabled, DOCA Bench appends to a CSV file if it exists or creates a new one. It is assumed that all invocation uses the exact same set of output values. This is not verified by DOCA Bench. The user must ensure that all tests that append to the CSV use the same set of output values.

7.1.14.9.5 --csv-separate-dynamic-values

A special case which creates a non-standard CSV file. All values that are not supported by sweep tests are reported only once first, then a new line of headers for values emitted during the test, then a row for each test result. This is reserved for an internal use case and should not be relied upon by anyone else.

7.1.14.9.6 --enable-environment-information

Instructs DOCA Bench to collect some detailed system information as part of the test startup procedure which is then made available for output in the CSV. These also gather the same details from the companion side if the companion is in use.

> This collection can take a long time (up to a few minutes in some circumstances) to complete, so it is not recommended unless you know you need it.

7.1.14.10 Remote Memory Testing

Some libraries (e.g., `doca_dma`) support the use of remote memory. To enable this, the user can specify one or both of the remote memory flags `--use-remote-input-buffers` and `--use-remote-output-buffers`. This tells DOCA Bench to use the companion to create a remote mmap. This remote mmap is then used to create buffers that are submitted to the component under test.

> These flags should be used with caution and an understanding that if the underlying components under test can support this scenario, there is no automated checking. It is user responsibility to ensure these are used appropriately.

7.1.14.10.1 --use-remote-input-buffers

Specifies that the memory used for the initial immutable job input buffers into a pipeline should be backed by an mmap on the remote side.

> Requires the companion app to be configured.

7.1.14.10.2 --use-remote-output-buffers

Specifies that all output and translation buffers in use are backed by an mmap on the remote side.

> Requires the companion app to be configured.

7.1.15 Network Options

7.1.15.1 --mtu-size

For use with `doca_rdma`. Value is an enum: `256B` `512B` `1KB` `2KB` `4KB` or `raw_eth`.

7.1.15.2 --receive-queue-size

For use with `doca_rdma`. Configure the RDMA RQ size independently of the SQ size.

7.1.15.3 --send-queue-size

For use with `doca_rdma`. Configure the RDMA SQ size independently of the RQ size.

7.1.15.4 DOCA Lib Configuration Options

7.1.15.4.1 --task-pool-size

Default value: 1024

Configure the maximum task pool size used when libraries initialize task pools.

7.1.15.5 Pipeline Configuration

DOCA Bench is based on a pipeline of operations, This allow for complex test scenarios where multiple components are tested in parallel. Currently only a single chain of operations in a pipeline is supported (but scaled across multiple cores or threads), future versions will allow for varied pipeline's per CPU core.

A pipeline is described as a series of steps. All steps have a few general characteristics:

- Step type: `doca_dma`, `doca_sha`, `doca_compress`, etc.
- An operation category – transformative or non-transformative
- An input data category – structured or non structured

Individual step types may also have some additional metadata information or configuration as defined on a per step basis.

Metadata examples:

- `doca_compress` requires an operation type: `compress` or `decompress`
- `doca_aes` requires an operation type: `encrypt` or `decrypt`
- `doca_ec` requires an operation type: `create`, `recover` or `update`
- `doca_rdma` requires a direction: `send`, `receive` or `bidir`

Configuration examples:

- `--pipeline-steps doca_dma`
- `--pipeline-steps doca_compress::compress,doca_compress::decompress`

7.1.15.5.1 --pipeline-steps

Define the step(s) (comma-separated list) to be executed by each thread of the test.

The following is the list of supported steps:

- `doca_compress::compress`
- `doca_compress::decompress`
- `doca_dma`
- `doca_ec::create`
- `doca_ec::recover`

- doca_ec::update
- doca_sha
- doca_rdma::send
- doca_rdma::receive
- doca_rdma::bidir
- doca_aes_gcm::encrypt
- doca_aes_gcm::decrypt
- doca_cc::client_producer
- doca_cc::client_consumer
- doca_eth::rx
- doca_eth::tx
- doca_gpunetio::eth_tx
- doca_gpunetio::eth_rx
- doca_gpunetio::eth_bidir

Some modules may be unavailable if they were not compiled as part of DOCA when DOCA Bench was compiled.

7.1.15.5.2 --attribute

Some of the options are very niche or specific to a single step/mmo type, so they are defined simply as attributes instead of a unique command-line argument.

The following is the list of supported options:

- doption.mmp.log_qp_depth
- doption.mmo.log.num_qps
- doption.companion_app.path
- doca_compress.algorithm
- doca_ec.matrix_count
- doca_ec.data_block_count
- doca_ec.redundancy_block_count
- doca_sha.algorithm
- doca_rdma.gid-index
- doca_eth.max_burst_size
- doca_eth.l3_chksum_offload
- doca_eth.l4_chksum_offload
- doca_gpunetio.threads-per-block
- doca_gpunetio.num-blocks
- doca_gpunetio.eth-rx-timout-ns
- doca_gpunetio.eth-rx-max-rx-packets

7.1.15.5.3 --warm-up-jobs

Default value: 100

Warm-up serves two purposes:

- Firstly, it runs N tasks in a round robin fashion to get the data path code, tasks memory, and tasks data buffers memory into the CPU caches before the measurement of the test begins
- Secondly, it uses **doca_task_try_submit** instead of **doca_task_submit** to validate the jobs. This validation is not desirable during the proper hot path as it costs time revalidating the task each execution.

The user should ensure their warmup count equals or exceeds the number of tasks being used per thread (see **--data-provider-job-count**).

7.1.15.6 Companion Configuration

Some tests require a remote system to function. For this purpose, DOCA Bench comes bundled with a companion application (this application is installed as part of the DOCA-for-Host or BlueField packages). The companion is responsible for providing services to DOCA Bench such as creating a **doca_mmap** on the remote side and exporting it for use with remote operations like **doca_dma** / **doca_sha**, or other **doca_libs** that support remote memory input buffers. DOCA Bench can also provide remote worker processes for libraries that require them such as **doca_rdma** and **doca_cc**. The companion is enabled by providing the **--companion-connection-string** argument. Companion remote workers are enabled by providing either of the arguments **--companion-core-list** or **--companion-core-mask**.

> DOCA Bench requires that an SSH key is configured to allow the user specified to SSH without a password to the remote system using the supplied address (to launch the companion). Refer to your OS's documentation for information on how to achieve this.

The companion connection may also specify the **no-launch** option.

> This is reserved for expert developer use.

The user may also specify a path to a specific companion binary to allow them to test companion binaries not in the default install path using the following command:

```
--attribute doption.companion_app.path=/tmp/my_doca_build/tools/bench/doca_bench_companion
```

> This is reserved for expert developer use.

7.1.15.6.1 --companion-connection-string

Specifies the details required to establish a connection to and execute the companion process.

- Example of running DOCA Bench from the host side using the BlueField for the remote side using **doca_comch** as the communications method:

```
--companion-connection-string "proto=dcc,mode=DPU,user=bob,addr=172.17.0.1,dev=03:00.0,rep=d8:00.0"
```

- Example of running DOCA Bench from the BlueField side using the host for the remote side using **doca_comch** as the communications method:

```
--companion-connection-string "proto=dcc,mode=host,user=bob,addr=172.17.0.1,dev=d8:00.0"
```

- Example of running DOCA Bench on one host with the companion on another host using TCP as the communications method:

```
--companion-connection-string "proto=tcp,user=bob,addr=172.17.0.1,port=12345,dev=d8:00.0"
```

> ⚠ For **doca_rdma** only.

7.1.15.6.2 --companion-core-list

Works the same way as **--core-list** but defines the cores to be used on the companion side.

> ⚠ Must be at least as large as the **--core-list**.

7.1.15.6.3 --companion-core-mask

Works the same way as **--core-mask** but defines the cores to be used on the companion side.

> ⚠ Must be at least as large as the **--core-mask**.

7.1.15.7 Sweep Tests

7.1.15.7.1 --sweep

DOCA Bench supports executing a set of tests based on a number of value ranges. For example, to understand the performance of multi-threading, the user may wish to run the same test for various CPU core counts. They may also wish to vary more than one aspect of the test. Providing one or more **--sweep** parameters activates sweep test mode where every combination of values is tested with a single invocation of DOCA Bench.

The following is a list of the supported sweep test options:

- **core-count**
- **data-provider-input-file**
- **data-provider-job-count**
- **gather-value**

- mtu-size
- receive-queue-size
- send-queue-size
- threads-per-core
- task-pool-size
- uniform-job-size
- doption.mmo.log_qp_depth
- doption.mmo.log_num_qps
- doca_rdma.transport-type
- doca_rdma.gid-index

Sweep test argument values take one of three forms:

- --sweep param,start_value,end_value,+N
- --sweep param,start_value,end_value,*N
- --sweep param,value1,...,valueN

Sweep core count and input file example:

```
--sweep core-count,1,8,*2 -sweep data-provider-input-file,file1.bin,file2.bin
```

This would sweep cores 1-8, inclusive, multiplying the value each time as 1,2,4,8 and two different input files resulting in a cumulative 8 test cases:

Iteration Number	Core Count	Input File
1	1	file1.bin
2	2	file1.bin
3	4	file1.bin
4	8	file1.bin
5	1	file2.bin
6	2	file2.bin
7	4	file2.bin
8	8	file2.bin

7.1.15.8 Queries

7.1.15.8.1 Device Capabilities

DOCA Bench allows the querying of a device to report which step types are available as well as information of valid configuration options for each step. A device must be specified:

```
tools/bench/doca_bench --device 03:00.0 --query device-capabilities
```

For each supported library, this would report:

- Capable – if that library is enabled in DOCA Bench at compile time (if not capable, installing the library would not make it become available to bench)
- Installed – if the library is installed on the machine executing the query (if not installed, installing it would make it available to bench)
- Library wide attributes
- A list of supported task types (~= step name)
    - If the task type is supported
    - Task specific attributes/capabilities

```
doca_compress:
    Capable: yes
    Installed: yes
    Tasks:
        compress::deflate:
            Supported: no
        compress::lz4:
            Supported: no
        compress::lz4_stream:
            Supported: no
        decompress::deflate:
            Supported: yes
            Max buffer length: 134217728
        decompress::lz4:
            Supported: yes
            Max buffer length: 134217728
        decompress::lz4_stream:
            Supported: yes
            Max buffer length: 134217728
```

In addition to querying BlueField devices, GPU devices can also be queried. This requires DOCA GPUNetIO and the DOCA Bench extension package to be installed. A GPU device must be specified:

```
tools/bench/doca_bench --gpu b5:00.0 --query device-capabilities
```

7.1.15.8.2 Supported Sweep Attributes

Shows the possible parameters that can be used with the sweep test parameter

```
tools/bench/doca_bench --query sweep-properties
```

Example output:

```
Supported query properties: [
        core-count
        threads-per-core
        uniform-job-size
        task-pool-size
        data-provider-job-count
        gather-value
        mtu-size
        send-queue-size
        receive-queue-size
        doption.mmo.log_qp_depth
        doption.mmo.log_num_qps
        doca_rdma.transport-type
        doca_rdma.gid-index
]
```

7.1.16 Test Memory Footprint

DOCA Bench allocates memory for all the tasks required by the test based on the input buffer size, output/intermediate buffer size, number of cores, number of threads, and number of jobs in

use. All jobs contain an input buffer, an output buffer, and an intermediate buffer. The input buffer is immutable and sized based on the data provider in use. The output and intermediate buffers are sized based on the users specification or automatically calculated at the users request. For a library which produces the same amount of output as it consumes (e.g., `doca_dma`), typically the user should set the buffers all to the same size to make things as efficient as possible.

The memory footprint for job buffers can be calculated as: `(number-of-tasks) * (number-of-cores) * (number-of-threads-per-core) * (input-buffer-size + (output/intermediate-buffer-size * 2))`. For a 1KB job with the default of 32 jobs, 1 core, and 1 core per thread, the memory footprint would be 96KB.

For sweep testing and structured data input, it can be difficult to pick a suitable output buffer size so the user may choose to specify 0 and have DOCA Bench try all the tasks once to calculate the required output buffer sizes. This only has a cost in terms of time taken to perform the calculation. After this, there is no difference between auto-sizing and manually sizing the jobs output buffers.

> ⚠ When running DOCA Bench on the BlueField and on some host OSs, it may be necessary to increase the limit of how much memory the process can acquire. Consult your OS's documentation for details of how to do this.

7.1.17 GPUNetIO Benchmarks

DOCA Bench supports benchmarks where task submission and processing are offloaded to a GPU using DOCA GPUNetIO.

Requirements:

- DOCA GPUNetIO, DOCA Bench Extension, and NVIDIA GDRCopy library..
- The `GDRCOPY_PATH_L` environment variable must be set.

Limitations:

- Both `doca_bench` and `doca_bench_companion` must run as `sudo` or root.
- Only throughput testing is currently supported. However, test duration can be defined by time, job count, or byte limits.
- Use of multiple CPU cores with `doca_bench` and `doca_bench_companion` is not supported.
- Multi-step tests are not supported.

7.1.17.1 Controlling CUDA Threads

You can control the number of CUDA threads used for execution by specifying the block count and threads per block.

- `--attribute doca_gpunetio.threads-per-block=<number>` (Default: 32)
- `--attribute doca_gpunetio.num-blocks=<number>` (Default: 1)

We don't have a way to export this macro.

7.1.17.2 GPUNetIO Eth

This benchmark uses DOCA GPUNetIO Eth to transmit or receive Ethernet packets to/from a remote host.

- A GPU is required only on the **doca_bench** side. The remote side runs **doca_bench_companion** as a CPU-driven packet generator or sink.
- Benchmark modes: Unidirectional Send, Unidirectional Receive, and Bidirectional

For unidirectional receive and bidirectional, two additional attributes can be configured:

- **--attribute doca_gpunetio.eth-rx-timout-ns=<number>** (default 2000)
- **--attribute doca_gpunetio.eth-rx-max-rx-packets=<number>** (default 2048)

Example DOCA Bench commands for GPUNetIO Eth bidirectional test:

```
tools/bench/doca_bench \
  --pipeline-steps doca_gpunetio::eth_bidir \
  --device b1:00.0 --gpu b5:00.0 \
  --data-provider random-data \
  --uniform-job-size 512 \
  --run-limit-seconds 1 \
  --job-output-buffer-size 512 \
  --companion-connection-string proto=tcp,dev=d8:00.0,user=bob,addr=172.17.0.1,port=12345 \
  --companion-core-mask 0x1
```

7.1.18 DOCA Bench Sample Invocations

7.1.18.1 Overview

This guide provides examples of various invocations of the tool to help provide guidance and insight into it and the feature under test.

To make the samples clearer, certain verbose output and repeated information has been removed or shortened, in particular to output of the configuration or defaults when DOCA Bench is first executed is removed.

The command line options may need to be updated to suit your environment (e.g., TCP addresses, port numbers, interface names, usernames). See the "Command-line Parameters" section for more information.

7.1.18.2 DOCA Eth Receive Sample

- This test invokes DOCA Bench to run in Ethernet receive mode, configured to receive Ethernet frames of size 1500 bytes.
- The test runs for 3 seconds using a single core and use a maximum burst size of 512 frames.
- The test runs in the default throughput mode, with throughput figures displayed at the end of the test run.

- The companion application uses 6 cores to continuously transmit Ethernet frames of size 1500 bytes until it is stopped by DOCA Bench.

7.1.18.2.1 Command Line

```
doca_bench --core-mask 0x02 \
                                --pipeline-steps doca_eth::rx \
                                    --device b1:00.1 \
                                    --data-provider random-data \
                                    --uniform-job-size 1500 \
                                    --run-limit-seconds 3 \
                                    --attribute doca_eth.max-burst-size=512 \
                                    --companion-connection-string
proto=tcp,addr=10.10.10.10,port=12345,user=bob,dev=ens4f1np1 \
                                    --attribute doption.companion_app.path=/opt/mellanox/doca/tools/
doca_bench_companion \
                                    --companion-core-list 6 \
                                    --job-output-buffer-size 1500 \
                                    --mtu-size raw_eth
```

7.1.18.2.2 Results Output

```
[main] doca_bench : 2.7.0084
[main] release build
+ + + +
```

```
EAL: Detected shared linkage of DPDK
EAL: Multi-process socket /run/user/48679/dpdk/rte/mp_socket
EAL: Selected IOVA mode 'PA'
EAL: VFIO support initialized
TELEMETRY: No legacy callbacks, legacy socket not created
EAL: Probe PCI driver: mlx5_pci (15b3:a2d6) device: 0000:b1:00.1 (socket 2)
[08:19:32:110524][398304][DOCA][WRN][engine_model.c:90][adapt_queue_depth] adapting queue depth to 128.
Executing...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Cleanup...
[main] Completed! tearing down...
Aggregate stats
        Duration:      3000633 micro seconds
        Enqueued jobs: 611215
        Dequeued jobs: 611215
        Throughput:    000.204 MOperations/s
        Ingress rate:  002.276 Gib/s
        Egress rate:   002.276 Gib/s
```

7.1.18.2.3 Results Overview

As a single core is specified, there is a single section of statistics output displayed.

7.1.18.3 DOCA Eth Send Sample

- This test invokes DOCA Bench to run in Ethernet send mode, configured to transmit Ethernet frames of size 1500 bytes
- Random data is used to populate the Ethernet frames
- The test runs for 3 seconds using a single core and uses a maximum burst size of 512 frames
- L3 and L4 checksum offloading is not enabled
- The test runs in the default throughput mode, with throughput figures displayed at the end of the test run
- The companion application uses 6 cores to continuously receive Ethernet frames of size 1500 bytes until it is stopped by DOCA Bench

7.1.18.3.1 Command Line

```
doca_bench --core-mask 0x02 \
                                    --pipeline-steps doca_eth::tx \
                                    --device b1:00.1 \
                                    --data-provider random-data \
                                    --uniform-job-size 1500 \
                                    --run-limit-seconds 3 \
                                    --attribute doca_eth.max-burst-size=512 \
                                    --attribute doca_eth.l4-chksum-offload=false \
                                    --attribute doca_eth.l3-chksum-offload=false \
                                    --companion-connection-string
proto=tcp,addr=10.10.10.10,port=12345,user=bob,dev=ens4f1np1 \
                                    --attribute doption.companion_app.path=/opt/mellanox/doca/tools/
doca_bench_companion \
                                    --companion-core-list 6 \
                                    --job-output-buffer-size 1500
```

7.1.18.3.2 Results Output

```
[main] doca_bench : 2.7.0084
[main] release build
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
DOCA bench supported modules: [doca_comm_channel, doca_compress, doca_dma, doca_ec, doca_eth, doca_sha, doca_comch,
doca_rdma, doca_aes_gcm]
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +

DOCA bench configuration
Static configuration: [
        Attributes: [doca_eth.l4-chksum-offload:false, doca_eth.max-burst-size:512, doption.companion_app.path:/
opt/mellanox/doca/tools/doca_bench_companion, doca_eth.l3-chksum-offload:false]
        Companion configuration: [
                Device: ens4f1np1
                Remote IP address: "bob@10.10.10.10"
                Core set: [6]
        ]
```

```
        Pipelines: [
                Steps: [
                        name: "doca_eth::tx"
                        attributes: []
                ]
                Use remote input buffers: no
                Use remote output buffers: no
                Latency bucket_range: 10000ns-110000ns
        ]
        Run limits: [
                Max execution time: 3seconds
                Max jobs executed: -- not configured --
                Max bytes processed: -- not configured --
        ]
        Data provider: [
                Name: "random-data"
                Job output buffer size: 1500
        ]
        Device: "b1:00.1"
        Device representor: "-- not configured --"
        Warm up job count: 100
        Input files dir: "-- not configured --"
        Output files dir: "-- not configured --"
        Core set: [1]
        Benchmark mode: throughput
        Warnings as errors: no
        CSV output: [
                File name: -- not configured --
                Selected stats: []
                Deselected stats: []
                Separate dynamic values: no
                Collect environment information: no
                Append to stats file: no
        ]
]
Test permutations: [
        Attributes: []
        Uniform job size: 1500
        Core count: 1
        Per core thread count: 1
        Task pool size: 1024
        Data provider job count: 128
        MTU size: -- not configured --
        SQ depth: -- not configured --
        RQ depth: -- not configured --
        Input data file: -- not configured --
]

[main] Initialize framework...
[main] Start execution...
Preparing...
Executing...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Cleanup...
[main] Completed! tearing down...
Aggregate stats
        Duration:      3000049 micro seconds
        Enqueued jobs: 17135128
        Dequeued jobs: 17135128
        Throughput:    005.712 MOperations/s
        Ingress rate:  063.832 Gib/s
        Egress rate:   063.832 Gib/s
```

7.1.18.3.3 Results Overview

As a single core is specified, there is a single section of statistics output displayed.

7.1.18.4 Host-side AES-GCM Decrypt Sample

- This test invokes DOCA Bench on the x86 host side to run the AES-GM Decryption step
- A file-set file is used to indicate which file is to be decrypted. The content of the file-set file lists the filename to be decrypted.
- The key to be used for the encryption and decryption is specified using the **doca_aes_gcm.key** -file attribute. This contains the key to be used.
- It will run until 5000 jobs have been processed
- It runs in the precision-latency mode, with latency and throughput figures displayed at the end of the test run
- A core mask is specified to indicate that cores 12, 13, 14, and 15 are to be used for this test

7.1.18.4.1 Command Line

```
doca_bench  --mode precision-latency \
            --core-mask 0xf000 \
            --warm-up-jobs 32 \
            --device 17:00.0 \
            --data-provider file-set \
            --data-provider-input-file aes_64_128.fileset \
            --run-limit-jobs 5000 \
            --pipeline-steps doca_aes_gcm::decrypt \
            --attribute doca_aes_gcm.key-file='aes128.key' \
            --job-output-buffer-size 80
```

7.1.18.4.2 Results Output

```
[main] Completed! tearing down...
Worker thread[0](core: 12) stats:
        Duration:      10697 micro seconds
        Enqueued jobs: 5000
        Dequeued jobs: 5000
        Throughput:    000.467 MOperations/s
        Ingress rate:  000.265 Gib/s
        Egress rate:   000.223 Gib/s
Worker thread[1](core: 13) stats:
        Duration:      10700 micro seconds
        Enqueued jobs: 5000
        Dequeued jobs: 5000
        Throughput:    000.467 MOperations/s
        Ingress rate:  000.265 Gib/s
        Egress rate:   000.223 Gib/s
Worker thread[2](core: 14) stats:
        Duration:      10733 micro seconds
        Enqueued jobs: 5000
        Dequeued jobs: 5000
        Throughput:    000.466 MOperations/s
        Ingress rate:  000.264 Gib/s
        Egress rate:   000.222 Gib/s
Worker thread[3](core: 15) stats:
        Duration:      10788 micro seconds
        Enqueued jobs: 5000
        Dequeued jobs: 5000
        Throughput:    000.463 MOperations/s
        Ingress rate:  000.262 Gib/s
        Egress rate:   000.221 Gib/s
Aggregate stats
        Duration:      10788 micro seconds
        Enqueued jobs: 20000
        Dequeued jobs: 20000
        Throughput:    001.854 MOperations/s
        Ingress rate:  001.050 Gib/s
        Egress rate:   000.884 Gib/s
        min:           1878 ns
        max:           4956 ns
        median:        2134 ns
        mean:          2145 ns
        90th %ile:     2243 ns
        95th %ile:     2285 ns
        99th %ile:     2465 ns
        99.9th %ile:   3193 ns
        99.99th %ile:  4487 ns
```

7.1.18.4.3 Results Overview

Since a core mask is specified but no core count, then all cores in the mask are used.

There is a section of statistics displayed for each core used as well as the aggregate statistics.

7.1.18.5 BlueField-side AES-GCM Encrypt Sample

- This test invokes DOCA Bench on the BlueField side to run the AES-GM encryption step
- A text file of size 2KB is the input for the encryption stage
- The key to be used for the encryption and decryption is specified using the **doca_aes_gcm.key** attribute
- It runs until 2000 jobs have been processed

- It runs in the bulk-latency mode, with latency and throughput figures displayed at the end of the test run
- A single core is specified with 2 threads

7.1.18.5.1 Command Line

```
doca_bench  --mode bulk-latency \
            --core-list 3 \
            --threads-per-core 2 \
            --warm-up-jobs 32 \
            --device 03:00.0 \
            --data-provider file \
            --data-provider-input-file plaintext_2k.txt \
            --run-limit-jobs 2000 \
            --pipeline-steps doca_aes_gcm::encrypt \
            --attribute doca_aes_gcm.key="0123456789abcdef0123456789abcdef" \
            --uniform-job-size 2048 \
            --job-output-buffer-size 4096
```

7.1.18.5.2 Results Output

```
[main] Completed! tearing down...
Worker thread[0](core: 3) stats:
        Duration:       501 micro seconds
        Enqueued jobs: 2048
        Dequeued jobs: 2048
        Throughput:     004.082 MOperations/s
        Ingress rate:   062.279 Gib/s
        Egress rate:    062.644 Gib/s
Worker thread[1](core: 3) stats:
        Duration:       466 micro seconds
        Enqueued jobs: 2048
        Dequeued jobs: 2048
        Throughput:     004.386 MOperations/s
        Ingress rate:   066.922 Gib/s
        Egress rate:    067.314 Gib/s
Aggregate stats
        Duration:       501 micro seconds
        Enqueued jobs: 4096
        Dequeued jobs: 4096
        Throughput:     008.163 MOperations/s
        Ingress rate:   124.558 Gib/s
        Egress rate:    125.287 Gib/s
Latency report:
                    :
                    :
                    :
                    :
                    :
                    ::
                    ::
                    ::
                    ::
                  .::.  .    .    ..
--------------------------------------------------------------------------------------------------------------
[<10000ns]: 0
.. OUTPUT RETRACTED (SHORTENED) ..
[26000ns -> 26999ns]: 0
[27000ns -> 27999ns]: 128
[28000ns -> 28999ns]: 2176
[29000ns -> 29999ns]: 1152
[30000ns -> 30999ns]: 128
[31000ns -> 31999ns]: 0
[32000ns -> 32999ns]: 0
[33000ns -> 33999ns]: 128
[34000ns -> 34999ns]: 0
[35000ns -> 35999ns]: 0
[36000ns -> 36999ns]: 0
[37000ns -> 37999ns]: 0
[38000ns -> 38999ns]: 128
[39000ns -> 39999ns]: 0
[40000ns -> 40999ns]: 0
[41000ns -> 41999ns]: 0
[42000ns -> 42999ns]: 0
[43000ns -> 43999ns]: 128
[44000ns -> 44999ns]: 128
[45000ns -> 45999ns]: 0
.. OUTPUT RETRACTED (SHORTENED) ..
[>110000ns]: 0
```

7.1.18.5.3 Results Overview

Since a single core is specified, there is a single section of statistics output displayed.

7.1.18.6 Host-side AES-GCM Encrypt and Decrypt Sample

- This test invokes DOCA Bench on the host side to run 2 AES-GM steps in the pipeline, first to encrypt a text file and then to decrypt the associated output from the encrypt step
- A text file of size 2KB is the input for the encryption stage
- The **input-cwd** option instructs DOCA Bench to look in a different location for the input file, in the parent directory in this case
- The key to be used for the encryption and decryption is specified using the **doca_aes_gcm.key** -file attribute, indicating that the key can be found in the specified file
- It runs until 204800 bytes have been processed
- It runs in the default throughput mode, with throughput figures displayed at the end of the test run

7.1.18.6.1 Command Line

```
doca_bench  --core-mask 0xf00 \
            --core-count 1 \
            --warm-up-jobs 32 \
            --device 17:00.0 \
            --data-provider file \
            --input-cwd ../. \
            --data-provider-input-file plaintext_2k.txt \
            --run-limit-bytes 204800 \
            --pipeline-steps doca_aes_gcm::encrypt,doca_aes_gcm::decrypt \
            --attribute doca_aes_gcm.key-file='aes128.key' \
            --uniform-job-size 2048 \
            --job-output-buffer-size 4096
```

7.1.18.6.2 Results Output

```
Executing...
Worker thread[0](core: 8) [doca_aes_gcm::encrypt>>doca_aes_gcm::decrypt] started...
Worker thread[0] Executing 32 warm-up tasks using 32 unique tasks
Cleanup...
[main] Completed! tearing down...
Aggregate stats
        Duration:      79 micro seconds
        Enqueued jobs: 214
        Dequeued jobs: 214
        Throughput:    002.701 MOperations/s
        Ingress rate:  041.214 Gib/s
        Egress rate:   041.214 Gib/s
```

7.1.18.6.3 Results Overview

Since a single core is specified, there is a single section of statistics output displayed.

7.1.18.7 Host-side SHA with CSV Output File Sample

- This test invokes DOCA Bench on the host side to execute the SHA operation using the SHA256 algorithm and to create a CSV file containing the test configuration and statistics
- A list of 1 core is provided with a count of 2 threads per core

7.1.18.7.1 Command Line

```
doca_bench  --core-mask 2 \
            --threads-per-core 2 \
            --pipeline-steps doca_sha \
            --device d8:00.0 \
```

```
            --data-provider random-data \
            --uniform-job-size 2048 \
            --job-output-buffer-size 2048 \
            --run-limit-seconds 3 \
            --attribute doca_sha.algorithm=sha256 \
            --warm-up-jobs 100 \
            --csv-output-file /tmp/sha_256_test.csv
```

7.1.18.7.2 Results Output

```
Executing...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Data path thread [1] started...
WT[1] Executing 100 warm-up tasks using 100 unique tasks
Cleanup...
[main] Completed! tearing down...
Stats for thread[0](core: 1)
        Duration:      3000064 micro seconds
        Enqueued jobs: 3713935
        Dequeued jobs: 3713935
        Throughput:    001.238 MOperations/s
        Ingress rate:  018.890 Gib/s
        Egress rate:   000.295 Gib/s
Stats for thread[1](core: 1)
        Duration:      3000056 micro seconds
        Enqueued jobs: 3757335
        Dequeued jobs: 3757335
        Throughput:    001.252 MOperations/s
        Ingress rate:  019.110 Gib/s
        Egress rate:   000.299 Gib/s
Aggregate stats
        Duration:      3000064 micro seconds
        Enqueued jobs: 7471270
        Dequeued jobs: 7471270
        Throughput:    002.490 MOperations/s
        Ingress rate:  038.000 Gib/s
        Egress rate:   000.594 Gib/s
```

7.1.18.7.3 Results Overview

As a single core has been specified with a thread count of 2, there are statistics displayed for each thread as well as the aggregate statistics.

It can also be observed that 2 threads are started on core 1 with each thread executing the warm-up jobs.

The contents of the **/tmp/sha_256_test.csv** are shown below. It can be seen that the configuration used for the test and the associated statistics from the test run are listed:

```
cfg.companion.connection_string,cfg.pipeline.steps,cfg.pipeline.use_remote_input_buffers,cfg.pipeline.use_remote_ou
tput_buffers,cfg.pipeline.bulk_latency.lower_bound,cfg.pipeline.bulk_latency.bucket_width,cfg.run_limit.duration,cf
g.r
un_limit.jobs,cfg.run_limit.bytes,cfg.data_provider.type,cfg.data_provider.output_buffer_size,cfg.device.pci_addres
s,cfg.input.cwd,cfg.output.cwd,cfg.warmup_job_count,cfg.core_set,cfg.benchmark_mode,cfg.warnings_are_errors,cfg.att
rib
ute.doca_compress.algorithm,cfg.attribute.doca_ec.matrix_type,cfg.attribute.doca_ec.data_block_count,cfg.attribute.
doca_ec.redundancy_block_count,cfg.attribute.doca_ec.use_precomputed_matrix,cfg.attribute.doca_eth.l3_chksum_offloa
d,c
fg.attribute.doca_eth.l4_chksum_offload,cfg.attribute.doca_sha.algorithm,cfg.uniform_job_size,cfg.core_count,cfg.per
_core_thread_count,cfg.task_pool_size,cfg.data_provider_job_count,cfg.sg_config,cfg.mtu-size,cfg.send-queue-
size,cfg.    receive-queue-size,cfg.data-provider-input-
file,cfg.attribute.mmo.log_qp_depth,cfg.attribute.mmo.log_num_qps,stats.input.job_count,stats.output.job_count,stat
s.input.byte_count,stats.output.byte_count,stats.input.throughput.bytes,sta
ts.output.throughput.bytes,stats.input.throughput.rate,stats.output.throughput.rate
,[doca_sha],0,0,10000,1000,3,,,random-data,2048,d8:00.0,,,100,
[1],throughput,0,,,,,,,,sha256,2048,1,2,1024,128,1 fragments,,,,,,,7471270,7471270,15301160960,239109312,038.000 Gi
b/s,000.594 Gib/s,2.490370 MOperations/s,2.490370 MOpera    tions/s
```

7.1.18.8 Host-side SHA with CSV Appended Output File Sample

- This test invokes DOCA Bench on the Host side to execute the SHA operation using the SHA512 algorithm and to create a csv file containing the test configuration and statistics,
- The command is repeated with the added option of csv-append-mode. This instructs DOCA Bench to append the test run statistics to the existing csv file.

- A list of 1 core is provided with a count of 2 threads per core.

7.1.18.8.1 Command Line

1. Create the initial **/tmp/sha_512_test.csv** file:

```
doca_bench    --core-list 2 \
                --threads-per-core 2 \
                --pipeline-steps doca_sha \
                --device d8:00.0 \
                --data-provider random-data \
                --uniform-job-size 2048 \
                --job-output-buffer-size 2048 \
                --run-limit-seconds 3 \
                --attribute doca_sha.algorithm=sha512 \
                --warm-up-jobs 100 \
                --csv-output-file /tmp/sha_512_test.csv
```

2. The second command is:

```
./doca_bench    --core-list 2 \
                --threads-per-core 2 \
                --pipeline-steps doca_sha \
                --device d8:00.0 \
                --data-provider random-data \
                --uniform-job-size 2048 \
                --job-output-buffer-size 2048 \
                --run-limit-seconds 3 \
                --attribute doca_sha.algorithm=sha512 \
                --warm-up-jobs 100 \
                --csv-output-file /tmp/sha_512_test.csv \
                --csv-append-mode
```

   This causes DOCA Bench to append the configuration and statistics from the second command run to the **/tmp/sha_512_test.csv** file.

7.1.18.8.2 Results Output

This is a snapshot of the results output from the first command run:

```
Executing...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Data path thread [1] started...
WT[1] Executing 100 warm-up tasks using 100 unique tasks
Cleanup...
[main] Completed! tearing down...
Stats for thread[0](core: 2)
        Duration:      3015185 micro seconds
        Enqueued jobs: 3590717
        Dequeued jobs: 3590717
        Throughput:    001.191 MOperations/s
        Ingress rate:  018.171 Gib/s
        Egress rate:   000.568 Gib/s
Stats for thread[1](core: 2)
        Duration:      3000203 micro seconds
        Enqueued jobs: 3656044
        Dequeued jobs: 3656044
        Throughput:    001.219 MOperations/s
        Ingress rate:  018.594 Gib/s
        Egress rate:   000.581 Gib/s
Aggregate stats
        Duration:      3015185 micro seconds
        Enqueued jobs: 7246761
        Dequeued jobs: 7246761
        Throughput:    002.403 MOperations/s
        Ingress rate:  036.673 Gib/s
        Egress rate:   001.146 Gib/s
```

This is a snapshot of the results output from the second command run:

```
Executing...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Data path thread [1] started...
WT[1] Executing 100 warm-up tasks using 100 unique tasks
Cleanup...
[main] Completed! tearing down...
Stats for thread[0](core: 2)
```

```
        Duration:      3000072 micro seconds
        Enqueued jobs: 3602562
        Dequeued jobs: 3602562
        Throughput:    001.201 MOperations/s
        Ingress rate:  018.323 Gib/s
        Egress rate:   000.573 Gib/s
Stats for thread[1](core: 2)
        Duration:      3000062 micro seconds
        Enqueued jobs: 3659148
        Dequeued jobs: 3659148
        Throughput:    001.220 MOperations/s
        Ingress rate:  018.611 Gib/s
        Egress rate:   000.582 Gib/s
Aggregate stats
        Duration:      3000072 micro seconds
        Enqueued jobs: 7261710
        Dequeued jobs: 7261710
        Throughput:    002.421 MOperations/s
        Ingress rate:  036.934 Gib/s
        Egress rate:   001.154 Gib/s
```

7.1.18.8.3 Results Overview

Since a single core has been specified with a thread count of 2, there are statistics displayed for each thread as well as the aggregate statistics.

It can also be observed that 2 threads are started on core 1 with each thread executing the warm-up jobs.

The contents of the **/tmp/sha_256_test.csv**, after the first command has been run, are shown below. It can be seen that the configuration used for the test and the associated statistics from the test run are listed:

```
cfg.companion.connection_string,cfg.pipeline.steps,cfg.pipeline.use_remote_input_buffers,cfg.pipeline.use_remote_ou
tput_buffers,cfg.pipeline.bulk_latency.lower_bound,cfg.pipeline.bulk_latency.bucket_width,cfg.run_limit.duration,cf
g.run_limit.jobs,cfg.run_limit.bytes,cfg.data_provider.type,cfg.data_provider.output_buffer_size,cfg.device.pci_add
ress,cfg.input.cwd,cfg.output.cwd,cfg.warmup_job_count,cfg.core_set,cfg.benchmark_mode,cfg.warnings_are_errors,cfg.
attribute.doca_compress.algorithm,cfg.attribute.doca_ec.matrix_type,cfg.attribute.doca_ec.data_block_count,cfg.attr
ibute.doca_ec.redundancy_block_count,cfg.attribute.doca_ec.use_precomputed_matrix,cfg.attribute.doca_eth.l3_chksum_
offload,cfg.attribute.doca_eth.l4_chksum_offload,cfg.attribute.doca_sha.algorithm,cfg.uniform_job_size,cfg.core_cou
nt,cfg.per_core_thread_count,cfg.task_pool_size,cfg.data_provider_job_count,cfg.sg_config,cfg.mtu-size,cfg.send-
queue-size,cfg.receive-queue-size,cfg.data-provider-input-
file,cfg.attribute.mmo.log_qp_depth,cfg.attribute.mmo.log_num_qps,stats.input.job_count,stats.output.job_count,stat
s.input.byte_count,stats.output.byte_count,stats.input.throughput.bytes,stats.output.throughput.bytes,stats.input.t
hroughput.rate,stats.output.throughput.rate
,[doca_sha],0,0,10000,1000,3,,,random-data,2048,d8:00.0,,,100,
[2],throughput,0,,,,,,,,sha512,2048,1,2,1024,128,1 fragments,,,,,,,7246761,7246761,14841366528,463850048,036.673 Gi
b/s,001.146 Gib/s,2.403422 MOperations/s,2.403422 MOperations/s
```

The contents of the **/tmp/sha_256_test.csv**, after the second command has been run, are shown below. It can be seen that a second entry has been added detailing the configuration used for the test and the associated statistics from the test run:

```
cfg.companion.connection_string,cfg.pipeline.steps,cfg.pipeline.use_remote_input_buffers,cfg.pipeline.use_remote_ou
tput_buffers,cfg.pipeline.bulk_latency.lower_bound,cfg.pipeline.bulk_latency.bucket_width,cfg.run_limit.duration,cf
g.run_limit.jobs,cfg.run_limit.bytes,cfg.data_provider.type,cfg.data_provider.output_buffer_size,cfg.device.pci_add
ress,cfg.input.cwd,cfg.output.cwd,cfg.warmup_job_count,cfg.core_set,cfg.benchmark_mode,cfg.warnings_are_errors,cfg.
attribute.doca_compress.algorithm,cfg.attribute.doca_ec.matrix_type,cfg.attribute.doca_ec.data_block_count,cfg.attr
ibute.doca_ec.redundancy_block_count,cfg.attribute.doca_ec.use_precomputed_matrix,cfg.attribute.doca_eth.l3_chksum_
offload,cfg.attribute.doca_eth.l4_chksum_offload,cfg.attribute.doca_sha.algorithm,cfg.uniform_job_size,cfg.core_cou
nt,cfg.per_core_thread_count,cfg.task_pool_size,cfg.data_provider_job_count,cfg.sg_config,cfg.mtu-size,cfg.send-
queue-size,cfg.receive-queue-size,cfg.data-provider-input-
file,cfg.attribute.mmo.log_qp_depth,cfg.attribute.mmo.log_num_qps,stats.input.job_count,stats.output.job_count,stat
s.input.byte_count,stats.output.byte_count,stats.input.throughput.bytes,stats.output.throughput.bytes,stats.input.t
hroughput.rate,stats.output.throughput.rate
,[doca_sha],0,0,10000,1000,3,,,random-data,2048,d8:00.0,,,100,
[2],throughput,0,,,,,,,,sha512,2048,1,2,1024,128,1 fragments,,,,,,,7246761,7246761,14841366528,463850048,036.673 Gi
b/s,001.146 Gib/s,2.403422 MOperations/s,2.403422 MOperations/s
,[doca_sha],0,0,10000,1000,3,,,random-data,2048,d8:00.0,,,100,
[2],throughput,0,,,,,,,,sha512,2048,1,2,1024,128,1 fragments,,,,,,,7261710,7261710,14871982080,464806784,036.934 Gi
b/s,001.154 Gib/s,2.420512 MOperations/s,2.420512 MOperations/s
```

7.1.18.9 BlueField-side SHA with Transient Statistics Sample

- This test invokes DOCA Bench on the BlueField side to execute the SHA operation using the SHA1 algorithm and to display statistics every 2000 milliseconds during the test run

- A list of 3 cores is provided with a count of 2 threads per core and a core-count of 1
- The core-count instructs DOCA Bench to use the first core number in the core list, in this case core number 2

7.1.18.9.1 Command Line

```
doca_bench    --core-list 2,3,4 \
                --core-count 1 \
                --threads-per-core 2 \
                --pipeline-steps doca_sha \
                --device 03:00.0 \
                --data-provider random-data \
                --uniform-job-size 2048 \
                --job-output-buffer-size 2048 \
                --run-limit-seconds 3 \
                -attribute doca_sha.algorithm=sha1 \
                --warm-up-jobs 100 \
                --rt-stats-interval 2000
```

7.1.18.9.2 Results Output

```
Executing...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Data path thread [1] started...
WT[1] Executing 100 warm-up tasks using 100 unique tasks
Stats for thread[0](core: 2)
        Duration:       965645 micro seconds
        Enqueued jobs: 1171228
        Dequeued jobs: 1171228
        Throughput:     001.213 MOperations/s
        Ingress rate:   018.505 Gib/s
        Egress rate:    000.181 Gib/s
Stats for thread[1](core: 2)
        Duration:       965645 micro seconds
        Enqueued jobs: 1171754
        Dequeued jobs: 1171754
        Throughput:     001.213 MOperations/s
        Ingress rate:   018.514 Gib/s
        Egress rate:    000.181 Gib/s
Aggregate stats
        Duration:       965645 micro seconds
        Enqueued jobs: 2342982
        Dequeued jobs: 2342982
        Throughput:     002.426 MOperations/s
        Ingress rate:   037.019 Gib/s
        Egress rate:    000.362 Gib/s
Stats for thread[0](core: 2)
        Duration:       2968088 micro seconds
        Enqueued jobs: 3653691
        Dequeued jobs: 3653691
        Throughput:     001.231 MOperations/s
        Ingress rate:   018.783 Gib/s
        Egress rate:    000.183 Gib/s
Stats for thread[1](core: 2)
        Duration:       2968088 micro seconds
        Enqueued jobs: 3689198
        Dequeued jobs: 3689198
        Throughput:     001.243 MOperations/s
        Ingress rate:   018.965 Gib/s
        Egress rate:    000.185 Gib/s
Aggregate stats
        Duration:       2968088 micro seconds
        Enqueued jobs: 7342889
        Dequeued jobs: 7342889
        Throughput:     002.474 MOperations/s
        Ingress rate:   037.748 Gib/s
        Egress rate:    000.369 Gib/s
Cleanup...
[main] Completed! tearing down...
Stats for thread[0](core: 2)
        Duration:       3000122 micro seconds
        Enqueued jobs: 3694128
        Dequeued jobs: 3694128
        Throughput:     001.231 MOperations/s
        Ingress rate:   018.789 Gib/s
        Egress rate:    000.184 Gib/s
Stats for thread[1](core: 2)
        Duration:       3000089 micro seconds
        Enqueued jobs: 3751128
        Dequeued jobs: 3751128
        Throughput:     001.250 MOperations/s
        Ingress rate:   019.079 Gib/s
        Egress rate:    000.186 Gib/s
Aggregate stats
        Duration:       3000122 micro seconds
        Enqueued jobs: 7445256
        Dequeued jobs: 7445256
        Throughput:     002.482 MOperations/s
```

```
        Ingress rate:  037.867 Gib/s
        Egress rate:   000.370 Gib/s
```

7.1.18.9.3 Results Overview

Although a core list of 3 cores has been specified, the core-count value of 1 instructs DOCA Bench to use the first entry in the core list.

It can be seen that as a thread-count of 2 has been specified, there are 2 threads created on core 2.

A transient statistics interval of 2000 milliseconds has been specified, and the transient statistics per thread can be seen, as well as the final aggregate statistics.

7.1.18.10 Host-side Local DMA with Core Sweep Sample

- This test invokes DOCA Bench to execute a local DMA operation on the host
- It specifies that a core sweep should be carried out using core counts of 1, 2, and 4 using the option `--sweep core-count,1,4,*2`
- Test output is to be saved in a CSV file `/tmp/dma_sweep.csv` and a filter is applied so that only statistics information is recorded. No configuration information is to be recorded.

7.1.18.10.1 Command Line

```
doca_bench    --core-mask 0xff \
                --sweep core-count,1,4,*2 \
                --pipeline-steps doca_dma \
                --device d8:00.0 \
                --data-provider random-data \
                --uniform-job-size 2048 \
                --job-output-buffer-size 2048 \
                --run-limit-seconds 5 \
                --csv-output-file /tmp/dma_sweep.csv \
                --csv-stats "stats.*"
```

7.1.18.10.2 Results Overview

```
Test permutations: [
        Attributes: []
        Uniform job size: 2048
        Core count: 1
        Per core thread count: 1
        Task pool size: 1024
        Data provider job count: 128
        MTU size: -- not configured --
        SQ depth: -- not configured --
        RQ depth: -- not configured --
        Input data file: -- not configured --
        --------------------------------
        Attributes: []
        Uniform job size: 2048
        Core count: 2
        Per core thread count: 1
        Task pool size: 1024
        Data provider job count: 128
        MTU size: -- not configured --
        SQ depth: -- not configured --
        RQ depth: -- not configured --
        Input data file: -- not configured --
        --------------------------------
        Attributes: []
        Uniform job size: 2048
        Core count: 4
        Per core thread count: 1
        Task pool size: 1024
        Data provider job count: 128
        MTU size: -- not configured --
        SQ depth: -- not configured --
        RQ depth: -- not configured --
        Input data file: -- not configured --
]

[main] Initialize framework...
```

```
[main] Start execution...
Preparing permutation 1 of 3...
Executing permutation 1 of 3...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Cleanup permutation 1 of 3...
Aggregate stats
        Duration:      5000191 micro seconds
        Enqueued jobs: 22999128
        Dequeued jobs: 22999128
        Throughput:    004.600 MOperations/s
        Ingress rate:  070.185 Gib/s
        Egress rate:   070.185 Gib/s
Preparing permutation 2 of 3...
Executing permutation 2 of 3...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Data path thread [1] started...
WT[1] Executing 100 warm-up tasks using 100 unique tasks
Cleanup permutation 2 of 3...
Stats for thread[0](core: 0)
        Duration:      5000066 micro seconds
        Enqueued jobs: 14409794
        Dequeued jobs: 14409794
        Throughput:    002.882 MOperations/s
        Ingress rate:  043.975 Gib/s
        Egress rate:   043.975 Gib/s
Stats for thread[1](core: 1)
        Duration:      5000188 micro seconds
        Enqueued jobs: 14404708
        Dequeued jobs: 14404708
        Throughput:    002.881 MOperations/s
        Ingress rate:  043.958 Gib/s
        Egress rate:   043.958 Gib/s
Aggregate stats
        Duration:      5000188 micro seconds
        Enqueued jobs: 28814502
        Dequeued jobs: 28814502
        Throughput:    005.763 MOperations/s
        Ingress rate:  087.932 Gib/s
        Egress rate:   087.932 Gib/s
Preparing permutation 3 of 3...
Executing permutation 3 of 3...
Data path thread [1] started...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
WT[1] Executing 100 warm-up tasks using 100 unique tasks
Data path thread [3] started...
WT[3] Executing 100 warm-up tasks using 100 unique tasks
Data path thread [2] started...
WT[2] Executing 100 warm-up tasks using 100 unique tasks
Cleanup permutation 3 of 3...
[main] Completed! tearing down...
Stats for thread[0](core: 0)
        Duration:      5000092 micro seconds
        Enqueued jobs: 7227025
        Dequeued jobs: 7227025
        Throughput:    001.445 MOperations/s
        Ingress rate:  022.055 Gib/s
        Egress rate:   022.055 Gib/s
Stats for thread[1](core: 1)
        Duration:      5000081 micro seconds
        Enqueued jobs: 7223269
        Dequeued jobs: 7223269
        Throughput:    001.445 MOperations/s
        Ingress rate:  022.043 Gib/s
        Egress rate:   022.043 Gib/s
Stats for thread[2](core: 2)
        Duration:      5000047 micro seconds
        Enqueued jobs: 7229678
        Dequeued jobs: 7229678
        Throughput:    001.446 MOperations/s
        Ingress rate:  022.063 Gib/s
        Egress rate:   022.063 Gib/s
Stats for thread[3](core: 3)
        Duration:      5000056 micro seconds
        Enqueued jobs: 7223037
        Dequeued jobs: 7223037
        Throughput:    001.445 MOperations/s
        Ingress rate:  022.043 Gib/s
        Egress rate:   022.043 Gib/s
Aggregate stats
        Duration:      5000092 micro seconds
        Enqueued jobs: 28903009
        Dequeued jobs: 28903009
        Throughput:    005.780 MOperations/s
        Ingress rate:  088.203 Gib/s
        Egress rate:   088.203 Gib/s
```

7.1.18.10.3 Results Overview

The output gives a summary of the permutations being carried out and then proceeds to display the statistics for each of the permutations.

The CSV output file contents can be seen to contain only statistics information. Configuration information is not included.

There is an entry for each of the sweep permutations:

```
stats.input.job_count,stats.output.job_count,stats.input.byte_count,stats.output.byte_count,stats.input.throughput.
bytes,stats.output.throughput.bytes,stats.input.throughput.rate,stats.output.throughput.rate
22999128,22999128,47102214144,47102214144,070.185 Gib/s,070.185 Gib/s,4.599650 MOperations/s,4.599650 MOperations/s
28814502,28814502,59012100096,59012100096,087.932 Gib/s,087.932 Gib/s,5.762683 MOperations/s,5.762683 MOperations/s
28903009,28903009,59193362432,59193362432,088.203 Gib/s,088.203 Gib/s,5.780495 MOperations/s,5.780495 MOperations/s
```

7.1.18.11 Host-side Local DMA with Job Size Sweep Sample

This test invokes DOCA Bench to execute a local DMA operation on the host.

It specifies that a uniform job size sweep should be carried out using job sizes 1024 and 2048 using the option **`--sweep uniform-job-size,1024,2048`**.

Test output is to be saved in a CSV file **`/tmp/dma_sweep_job_size.csv`** and collection of environment information is enabled.

7.1.18.11.1 Command Line

```
doca_bench    --core-mask 0xff \
                --core-count 1 \
                --pipeline-steps doca_dma \
                --device d8:00.0 \
                --data-provider random-data \
                --sweep uniform-job-size,1024,2048 \
                --job-output-buffer-size 2048 \
                --run-limit-seconds 5 \
                --csv-output-file /tmp/dma_sweep_job_size.csv \
                --enable-environment-information
```

7.1.18.11.2 Results Overview

```
Test permutations: [
        Attributes: []
        Uniform job size: 1024
        Core count: 1
        Per core thread count: 1
        Task pool size: 1024
        Data provider job count: 128
        MTU size: -- not configured --
        SQ depth: -- not configured --
        RQ depth: -- not configured --
        Input data file: -- not configured --
        ---------------------------------
        Attributes: []
        Uniform job size: 2048
        Core count: 1
        Per core thread count: 1
        Task pool size: 1024
        Data provider job count: 128
        MTU size: -- not configured --
        SQ depth: -- not configured --
        RQ depth: -- not configured --
        Input data file: -- not configured --
]

[main] Initialize framework...
[main] Start execution...
Preparing permutation 1 of 2...
Executing permutation 1 of 2...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Cleanup permutation 1 of 2...
Aggregate stats
        Duration:      5000083 micro seconds
        Enqueued jobs: 23645128
        Dequeued jobs: 23645128
        Throughput:    004.729 MOperations/s
        Ingress rate:  036.079 Gib/s
        Egress rate:   036.079 Gib/s
Preparing permutation 2 of 2...
Executing permutation 2 of 2...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Cleanup permutation 2 of 2...
[main] Completed! tearing down...
Aggregate stats
        Duration:      5000027 micro seconds
```

```
        Enqueued jobs: 22963128
        Dequeued jobs: 22963128
        Throughput:    004.593 MOperations/s
        Ingress rate:  070.078 Gib/s
        Egress rate:   070.078 Gib/s
```

7.1.18.11.3 Results Overview

The output gives a summary of the permutations being carried out and then proceeds to display the statistics for each of the permutations.

The CSV output file contents can be seen to contain statistics information and the environment information.

There is an entry for each of the sweep permutations.

```
cfg.companion.connection_string,cfg.pipeline.steps,cfg.pipeline.use_remote_input_buffers,cfg.pipeline.use_remote_ou
tput_buffers,cfg.pipeline.bulk_latency.lower_bound,cfg.pipeline.bulk_latency.bucket_width,cfg.run_limit.duration,cf
g.run_limit.jobs,cfg.run_limit.bytes,cfg.data_provider.type,cfg.data_provider.output_buffer_size,cfg.device.pci_add
ress,cfg.input.cwd,cfg.output.cwd,cfg.warmup_job_count,cfg.core_set,cfg.benchmark_mode,cfg.warnings_are_errors,cfg.
attribute.doca_compress.algorithm,cfg.attribute.doca_ec.matrix_type,cfg.attribute.doca_ec.data_block_count,cfg.attr
ibute.doca_ec.redundancy_block_count,cfg.attribute.doca_ec.use_precomputed_matrix,cfg.attribute.doca_eth.l3_chksum_
offload,cfg.attribute.doca_eth.l4_chksum_offload,cfg.attribute.doca_sha.algorithm,cfg.uniform_job_size,cfg.core_cou
nt,cfg.per_core_thread_count,cfg.task_pool_size,cfg.data_provider_job_count,cfg.sg_config,cfg.mtu-size,cfg.send-
queue-size,cfg.receive-queue-size,cfg.data-provider-input-
file,cfg.attribute.mmo.log_qp_depth,cfg.attribute.mmo.log_num_qps,stats.input.job_count,stats.output.job_count,stat
s.input.byte_count,stats.output.byte_count,stats.input.throughput.bytes,stats.output.throughput.bytes,stats.input.t
hroughput.rate,stats.output.throughput.rate,host.pci.3.address,host.pci.3.ext_tag,host.pci.3.link_type,host.pci.2.e
xt_tag,host.pci.2.address,host.cpu.0.model,host.ofed_version,host.pci.4.max_read_request,host.pci.2.width,host.cpu.
1.logical_cores,host.pci.2.eswitch_mode,host.pci.3.max_read_request,host.pci.4.address,host.pci.2.link_type,host.pc
i.1.max_read_request,host.pci.4.link_type,host.cpu.socket_count,host.pci.0.ext_tag,host.pci.6.port_speed,host.cpu.0
.physical_cores,host.pci.7.port_speed,host.memory.dimm_slot_count,host.cpu.1.model,host.pci.0.max_payload_size,host
.pci.6.relaxed_ordering,host.doca_host_package_version,host.pci.6.max_payload_size,host.pci.0.gen,host.pci.4.width,
host.pci.2.gen,host.pci.1.max_payload_size,host.pci.4.relaxed_ordering,host.pci.3.width,host.cpu.0.logical_cores,ho
st.cpu.0.arch,host.pci.4.port_speed,host.pci.4.eswitch_mode,host.pci.7.address,host.pci.5.eswitch_mode,host.pci.5.a
ddress,host.cpu.1.arch,host.pci.0.eswitch_mode,host.pci.7.width,host.pci.7.link_type,host.pci.1.link_type,host.pci.
3.gen,host.pci.7.max_read_request,host.pci.7.eswitch_mode,host.pci.6.gen,host.pci.2.port_speed,host.pci.7.gen,host.
pci.2.relaxed_ordering,host.pci.6.width,host.pci.4.gen,host.pci.6.address,host.hostname,host.pci.5.link_type,host.p
ci.6.link_type,host.pci.6.max_read_request,host.pci.7.max_payload_size,host.pci.5.gen,host.pci.6.eswitch_mode,host.
pci.5.width,host.pci.3.relaxed_ordering,host.pci.4.ext_tag,host.pci.0.width,host.pci.5.port_speed,host.pci.2.max_pa
yload_size,host.pci.3.max_payload_size,host.pci.5.max_payload_size,host.pci.2.max_read_request,host.pci.0.address,h
ost.pci.gen,host.os.family,host.pci.1.gen,host.pci.5.relaxed_ordering,host.pci.1.port_speed,host.pci.7.ext_tag,host
.pci.1.address,host.pci.3.eswitch_mode,host.pci.3.port_speed,host.pci.0.max_read_request,host.pci.1.ext_tag,host.pc
i.0.relaxed_ordering,host.pci.0.link_type,host.pci.5.max_read_request,host.pci.4.max_payload_size,host.pci.device_c
ount,host.memory.populated_dimm_count,host.memory.installed_capacity,host.pci.6.ext_tag,host.os.kernel_version,host
.pci.0.port_speed,host.pci.1.width,host.pci.7.relaxed_ordering,host.pci.1.relaxed_ordering,host.os.version,host.os.
name,host.cpu.1.physical_cores,host.numa_node_count,host.pci.5.ext_tag,host.pci.1.eswitch_mode
,[doca_dma],0,0,10000,1000,5,,,random-data,2048,d8:00.0,,,100,"[0, 1, 2, 3, 4, 5, 6,
7]",throughput,0,,,,,,,,1024,1,1,1024,128,1 fragments,,,,,,,23645128,23645128,24212611072,24212611072,036.079 Gib/
s,036.079 Gib/s,4.728947 MOperations/s,4.728947 MOperations/s,0000:5e:00.1,true,Infiniband,true,0000:5e:00.0,N/
A,OFED-internal-24.04-0.4.8,N/A,x63,N/A,N/A,N/A,N/A,0000:af:00.0,Infiniband,N/A,Ethernet,2,true,N/A,N/A,N/A,N/A,N/A,N/
A,true,<none>,N/A,Gen15,x63,Gen15,N/A,true,x63,N/A,x86_64,104857600000,N/A,0000:d8:00.1,N/A,0000:af:00.1,x86_64,N/
A,x63,Ethernet,Infiniband,Gen15,N/A,N/A,Gen15,N/A,Gen15,true,x63,Gen15,0000:d8:00.0,zibal,Ethernet,Ethernet,N/A,N/
A,Gen15,N/A,x63,true,true,x63,104857600000,N/A,N/A,N/A,N/A,0000:3b:00.0,N/A,Linux,Gen15,true,N/
A,true,0000:3b:00.1,N/A,N/A,N/A,N/A,true,true,Infiniband,N/A,N/A,8,N/A,270049112064,true,5.4.0-174-generic,N/
A,x63,true,true,20.04.1 LTS (Focal Fossa),Ubuntu,N/A,2,true,N/A
,[doca_dma],0,0,10000,1000,5,,,random-data,2048,d8:00.0,,,100,"[0, 1, 2, 3, 4, 5, 6,
7]",throughput,0,,,,,,,,2048,1,1,1024,128,1 fragments,,,,,,,22963128,22963128,47028486144,47028486144,070.078 Gib/
s,070.078 Gib/s,4.592600 MOperations/s,4.592600 MOperations/s,0000:5e:00.1,true,Infiniband,true,0000:5e:00.0,N/
A,OFED-internal-24.04-0.4.8,N/A,x63,N/A,N/A,N/A,N/A,0000:af:00.0,Infiniband,N/A,Ethernet,2,true,N/A,N/A,N/A,N/A,N/A,N/
A,true,<none>,N/A,Gen15,x63,Gen15,N/A,true,x63,N/A,x86_64,104857600000,N/A,0000:d8:00.1,N/A,0000:af:00.1,x86_64,N/
A,x63,Ethernet,Infiniband,Gen15,N/A,N/A,Gen15,N/A,Gen15,true,x63,Gen15,0000:d8:00.0,zibal,Ethernet,Ethernet,N/A,N/
A,Gen15,N/A,x63,true,true,x63,104857600000,N/A,N/A,N/A,N/A,0000:3b:00.0,N/A,Linux,Gen15,true,N/
A,true,0000:3b:00.1,N/A,N/A,N/A,N/A,true,true,Infiniband,N/A,N/A,8,N/A,270049112064,true,5.4.0-174-generic,N/
A,x63,true,true,20.04.1 LTS (Focal Fossa),Ubuntu,N/A,2,true,N/A
```

7.1.18.12 BlueField-side Remote DMA Sample

- This test invokes DOCA Bench to execute a remote DMA operation on the host
- It specifies the companion connection details to be used on the host and that remote output buffers are to be used

7.1.18.12.1 Command Line

```
doca_bench  --core-list 12 \
            --pipeline-steps doca_dma \
            --device 03:00.0 \
            --data-provider random-data \
            --uniform-job-size 2048 \
            --job-output-buffer-size 2048 \
```

```
            --use-remote-output-buffers \
            --companion-connection-string proto=tcp,port=12345,mode=host,dev=17:00.0,user=bob,addr=10.10.10.10 \
            --run-limit-seconds 5
```

7.1.18.12.2 Results Overview

```
Executing...
Worker thread[0](core: 12) [doca_dma] started...
Worker thread[0] Executing 100 warm-up tasks using 100 unique tasks
Cleanup...
[main] Completed! tearing down...
Aggregate stats
        Duration:      5000073 micro seconds
        Enqueued jobs: 32202128
        Dequeued jobs: 32202128
        Throughput:    006.440 MOperations/s
        Ingress rate:  098.272 Gib/s
        Egress rate:   098.272 Gib/s
```

7.1.18.12.3 Results Overview

None.

7.1.18.13 Compress BlueField-side Sample

This test is relevant for BlueField-2 only.

- This test invokes DOCA Bench to run compression using random data as input
- The compression algorithm specified is "deflate"

7.1.18.13.1 Command Line

```
doca_bench    --core-list 2 \
                --pipeline-steps doca_compress::compress \
                --device 03:00.0 \
                --data-provider random-data \
                --uniform-job-size 2048 \
                --job-output-buffer-size 4096 \
                --run-limit-seconds 3 \
                --attribute doca_compress.algorithm="deflate"
```

7.1.18.13.2 Result Output

```
Executing...
Data path thread [0] started...
WT[0] Executing 100 warm-up tasks using 100 unique tasks
Cleanup...
[main] Completed! tearing down...
Aggregate stats
        Duration:      3000146 micro seconds
        Enqueued jobs: 5340128
        Dequeued jobs: 5340128
        Throughput:    001.780 MOperations/s
        Ingress rate:  027.160 Gib/s
        Egress rate:   027.748 Gib/s
```

7.1.18.13.3 Results Overview

None

7.1.18.14 BlueField-side Decompress LZ4 Sample

- This test invokes DOCA Bench to run decompression using random data as input
- This test specifies a data provider of file set which contains the filename of an LZ4 compressed file
- Remote input buffers are specified to be used for the input jobs
- It specifies the companion connection details to be used on the host for the remote input buffers

7.1.18.14.1 Command Line

```
doca_bench --core-list 12 \
            --pipeline-steps doca_compress::decompress \
            --device 03:00.0 \
            --data-provider file-set \
            --data-provider-input-file lz4_compressed_64b_buffers.fs \
            --job-output-buffer-size 4096 \
            --run-limit-seconds 3 \
            --attribute doca_compress.algorithm="lz4" \
            --use-remote-output-buffers \
            --companion-connection-string proto=tcp,port=12345,mode=host,dev=17:00.0,user=bob,addr=10.10.10.10
```

7.1.18.14.2 Results Output

```
Executing...
Worker thread[0](core: 12) [doca_compress::decompress] started...
Worker thread[0] Executing 100 warm-up tasks using 100 unique tasks
Cleanup...
[main] Completed! tearing down...
Aggregate stats
        Duration:      3000043 micro seconds
        Enqueued jobs: 15306128
        Dequeued jobs: 15306128
        Throughput:    005.102 MOperations/s
        Ingress rate:  003.155 Gib/s
        Egress rate:   002.433 Gib/s
```

7.1.18.14.3 Results Comment

None

7.1.18.15 Host-side EC Creation in Bulk Latency Mode Sample

- This test invokes DOCA Bench to run the EC creation step.
- It runs in bulk latency mode and specifies the **doca_ec** attributes of **data_block_count**, **redundancy_block_count**, and **matrix_type**

7.1.18.15.1 Command Line

```
doca_bench --mode bulk-latency \
            --core-list 12 \
            --pipeline-steps doca_ec::create \
            --device 17:00.0 \
            --data-provider random-data \
            --uniform-job-size 1024 \
            --job-output-buffer-size 1024 \
            --run-limit-seconds 3 \
            --attribute doca_ec.data_block_count=16 \
            --attribute doca_ec.redundancy_block_count=16 \
            --attribute doca_ec.matrix_type=cauchy
```

7.1.18.15.2 Results Output

Bulk latency output will be similar to that presented in section "BlueField-side Decompress LZ4 Sample".

7.1.18.15.3 Results Comment

Bulk latency output will be similar to that presented earlier on this page.

7.1.18.16 BlueField-side EC Creation in Precision Latency Mode Sample

- This test invokes DOCA Bench to run the EC creation step
- It runs in precision latency mode and specifies the **doca_ec** attributes of **data_block_count**, **redundancy_block_count**, and **matrix_type**

7.1.18.16.1 Command Line

```
doca_bench    --mode precision-latency \
                --core-list 12 \
                --pipeline-steps doca_ec::create \
                --device 03:00.0 \
                --data-provider random-data \
                --uniform-job-size 1024 \
                --job-output-buffer-size 1024 \
                --run-limit-jobs 5000 \
                --attribute doca_ec.data_block_count=16 \
                --attribute doca_ec.redundancy_block_count=16 \
                --attribute doca_ec.matrix_type=cauchy
```

7.1.18.16.2 Results Output

None

7.1.18.16.3 Results Comment

Precision latency output will be similar to that presented earlier on this page.

7.1.18.17 Comch Consumer from Host Side Sample

- This test invokes DOCA Bench in Comch consumer mode using a core-list on host side and BlueField side
- The run-limit is 500 jobs

7.1.18.17.1 Command Line

```
./doca_bench --core-list 4 --warm-up-jobs 32 --pipeline-steps doca_comch::consumer --device ca:00.0 --data-provider
random-data --run-limit-jobs 500 --core-count 1 --uniform-job-size 4096 --job-output-buffer-size 4096 --companion-
connection-string proto=tcp,mode=dpu,dev=03:00.0,user=bob,addr=10.10.10.10,port=12345 --attribute
dopt.companion_app.path=<path to DPU doca_bench_companion application location> --data-provider-job-count 256 --
companion-core-list 12
```

7.1.18.17.2 Results Output

```
[main] Completed! tearing down...
Aggregate stats
        Duration:      1415 micro seconds
        Enqueued jobs: 500
        Dequeued jobs: 500
        Throughput:    000.353 MOperations/s
        Ingress rate:  000.000 Gib/s
        Egress rate:   010.782 Gib/s
```

7.1.18.17.3 Results Comment

The aggregate statistics show the test completed after 500 jobs were processed.

7.1.18.18 Host-side Comch Producer Sample

- This test invokes DOCA Bench in Comch producer mode using a core-mask on the host side and BlueField side
- The run-limit is 1000 jobs

7.1.18.18.1 Command Line

```
doca_bench    --core-list 4 \
                --warm-up-jobs 32 \
                --pipeline-steps doca_comch::producer \
                --device ca:00.0 \
                --data-provider random-data \
                --run-limit-jobs 500 \
                --core-count 1 \
                --uniform-job-size 4096 \
                --job-output-buffer-size 4096 \
                --companion-connection-string proto=tcp,mode=dpu,dev=03:00.0,user=bob,addr=10.10.10.10,port=12345 \
                --attribute dopt.companion_app.path=<path to DPU doca_bench_companion location> \
                --data-provider-job-count 256 \
                --companion-core-list 12
```

7.1.18.18.2 Results Overview

```
[main] Completed! tearing down...
Aggregate stats
        Duration:      407 micro seconds
        Enqueued jobs: 500
        Dequeued jobs: 500
        Throughput:    001.226 MOperations/s
        Ingress rate:  037.402 Gib/s
        Egress rate:   000.000 Gib/s
```

7.1.18.18.3 Results Comment

The aggregate statistics show the test completed after 500 jobs were processed.

7.1.18.19 Host-side RDMA Send Sample

- This test invokes DOCA Bench in RDMA send mode using a core-list on the send and receive side
- The send queue size is configured to 50 entries

7.1.18.19.1 Command Line

```
doca_bench  --pipeline-steps doca_rdma::send \
            --device d8:00.0 \
            --data-provider random-data \
            --uniform-job-size 2048 \
            --job-output-buffer-size 2048 \
            --run-limit-seconds 3 \
            --send-queue-size 50 \
            --companion-connection-string proto=tcp,addr=10.10.10.10,port=12345,user=bob,dev=ca:00.0 \
            --companion-core-list 12 \
            --core-list 12
```

7.1.18.19.2 Results Output

```
Test permutations: [
        Attributes: []
        Uniform job size: 2048
        Core count: 1
        Per core thread count: 1
        Task pool size: 1024
        Data provider job count: 128
        MTU size: -- not configured --
        SQ depth: 50
        RQ depth: -- not configured --
        Input data file: -- not configured --
]
```

7.1.18.19.3 Results Comment

The configuration output shows the send queue size configured to 50.

7.1.18.20 Host-side RDMA Receive Sample

- This test invokes DOCA Bench in RDMA receive mode using a core-list on the send and receive side
- The receive queue size is configured to 100 entries

7.1.18.20.1 Command Line

```
doca_bench  --pipeline-steps doca_rdma::receive \
            --device d8:00.0 \
            --data-provider random-data \
            --uniform-job-size 2048 \
            --job-output-buffer-size 2048 \
            --run-limit-seconds 3 \
            --receive-queue-size 100 \
            --companion-connection-string proto=tcp,addr=10.10.10.10,port=12345,user=bob,dev=ca:00.0 \
            --companion-core-list 12 \
            --core-list 12
```

7.1.18.20.2 Results Output

```
Test permutations: [
        Attributes: []
        Uniform job size: 2048
        Core count: 1
        Per core thread count: 1
        Task pool size: 1024
        Data provider job count: 128
        MTU size: -- not configured --
        SQ depth: -- not configured --
        RQ depth: 100
        Input data file: -- not configured --
]
```

7.1.18.20.3 Results Overview

The configuration output shows the receive queue size configured to 100.

7.2 DOCA Capabilities Print Tool

This document provides instruction on the usage of the DOCA Capabilities Print Tool.

7.2.1 Introduction

This tool is used to print all the available DOCA libraries and devices. For each DOCA device, the tool prints its representor devices and the capabilities it supports in each DOCA library.

7.2.2 Prerequisites

DOCA 2.6.0 and higher.

7.2.3 Description

This tool can be executed on the host or Arm sides.

The following capabilities are supported by this tool:

- DOCA device list – print the PCIe device of every available DOCA device and its attributes
- DOCA representor device list – for every DOCA device, print the PCIe device of every available DOCA representor device and its attributes
- DOCA library list – print the available DOCA libraries supported by the running OS and their availability for specific OSs
- DOCA library capabilities – for every DOCA device, print the capabilities it supports in every DOCA library
- DOCA logger list – print the available logger names of DOCA libraries

7.2.4 Execution

- To print all the available DOCA devices and their capabilities, run:

```
/opt/mellanox/doca/tools/doca_caps --list-devs
```

> Printing the capabilities of a specific DOCA device can be done using the **`--pci-addr`** flag.

Example output:

```
/opt/mellanox/doca/tools/doca_caps --list-devs
PCI: 0000:08:00.0
        ibdev_name                                  mlx5_0
        iface_name                                  eth2
        iface_index                                 4
        pci_func_type                               PF
        uplink_ib_port                              7
        mac_addr                                    e8:eb:d3:98:21:ec
```

```
        ipv4_addr                       0.0.0.0
        ipv6_addr                       fe80:0000:0000:0000:eaeb:d3ff:fe98:21ec
PCI: 0000:08:00.1
        ibdev_name                      mlx5_1
        iface_name                      eth3
        iface_index                     5
        pci_func_type                   PF
        uplink_ib_port                  7
        mac_addr                        e8:eb:d3:98:21:ed
        ipv4_addr                       0.0.0.0
        ipv6_addr                       fe80:0000:0000:0000:eaeb:d3ff:fe98:21ed
PCI: 0000:08:00.2
        ibdev_name                      mlx5_2
        iface_name                      eth7
        iface_index                     10
        pci_func_type                   VF
        uplink_ib_port                  1
        mac_addr                        00:00:00:00:00:00
        ipv4_addr                       0.0.0.0
        ipv6_addr                       fe80:0000:0000:0000:fcd2:6aff:fe2b:7620
PCI: 0000:08:00.3
        ibdev_name                      mlx5_3
        iface_name                      eth8
        iface_index                     11
        pci_func_type                   VF
        uplink_ib_port                  1
        mac_addr                        00:00:00:00:00:00
        ipv4_addr                       0.0.0.0
        ipv6_addr                       fe80:0000:0000:0000:78a1:42ff:fe8b:a275
PCI: 0000:08:00.4
        ibdev_name                      mlx5_4
        iface_name                      eth9
        iface_index                     12
        pci_func_type                   VF
        uplink_ib_port                  1
        mac_addr                        00:00:00:00:00:00
        ipv4_addr                       0.0.0.0
        ipv6_addr                       fe80:0000:0000:0000:1c75:c5ff:fe89:ef35
PCI: 0000:08:00.5
        ibdev_name                      mlx5_5
        iface_name                      eth12
        iface_index                     15
        pci_func_type                   VF
        uplink_ib_port                  1
        mac_addr                        00:00:00:00:00:00
        ipv4_addr                       0.0.0.0
        ipv6_addr                       fe80:0000:0000:0000:70f9:c2ff:fe39:efec
PCI: 0000:08:00.6
        ibdev_name                      mlx5_6
        iface_name                      eth13
        iface_index                     16
        pci_func_type                   VF
        uplink_ib_port                  1
        mac_addr                        00:00:00:00:00:00
        ipv4_addr                       0.0.0.0
        ipv6_addr                       fe80:0000:0000:0000:00a7:37ff:feab:27d7
PCI: 0000:08:00.0
        ibdev_name                      mlx5_7
        iface_name                      enp8s0f0s5
        iface_index                     18
        pci_func_type                   SF
        sf_index                        7
        uplink_ib_port                  1
        mac_addr                        02:25:f2:8d:a2:4c
        ipv4_addr                       0.0.0.0
        ipv6_addr                       fe80:0000:0000:0000:0025:f2ff:fe8d:a24c
PCI: 0000:08:00.1
        ibdev_name                      mlx5_8
        iface_name                      enp8s0f1s4
        iface_index                     20
        pci_func_type                   SF
        sf_index                        8
        uplink_ib_port                  1
        mac_addr                        02:25:f2:8d:a2:4d
        ipv4_addr                       0.0.0.0
        ipv6_addr                       fe80:0000:0000:0000:0025:f2ff:fe8d:a24d
PCI: 0000:08:00.1
        ibdev_name                      mlx5_9
        iface_name                      enp8s0f1s8
        iface_index                     22
        pci_func_type                   SF
        sf_index                        9
        uplink_ib_port                  1
        mac_addr                        02:25:f2:8d:a2:4e
        ipv4_addr                       0.0.0.0
        ipv6_addr                       fe80:0000:0000:0000:0025:f2ff:fe8d:a24e
```

- To print all the available DOCA representor devices and their capabilities, run:

```
/opt/mellanox/doca/tools/doca_caps --list-rep-devs
```

Printing the representor list of a specific DOCA device can be done using the **`--pci-addr`** flag.

Example output:

```
/opt/mellanox/doca/tools/doca_caps --list-rep-devs
PCI: 0000:08:00.0
        representor-PCI: 0000:08:00.2
            ib_port                                 2
            host_index                              0
            pf_index                                0
            vf_index                                0
            pci_func_type                           VF
            hotplug                                 no
            vuid                                    MT2228X06397MLNXS0D0F0VF1
            iface_name                              eth4
            iface_index                             7
        representor-PCI: 0000:08:00.3
            ib_port                                 3
            host_index                              0
            pf_index                                0
            vf_index                                1
            pci_func_type                           VF
            hotplug                                 no
            vuid                                    MT2228X06397MLNXS0D0F0VF2
            iface_name                              eth5
            iface_index                             8
        representor-PCI: 0000:08:00.4
            ib_port                                 4
            host_index                              0
            pf_index                                0
            vf_index                                2
            pci_func_type                           VF
            hotplug                                 no
            vuid                                    MT2228X06397MLNXS0D0F0VF3
            iface_name                              eth6
            iface_index                             9
        representor-PCI: 0000:08:00.0
            ib_port                                 5
            host_index                              0
            pf_index                                0
            sf_index                                5
            pci_func_type                           SF
            hotplug                                 no
            vuid                                    MT2228X06397MLNXS0D0F0SF32768
            iface_name                              en8f0pf0sf5
            iface_index                             17
PCI: 0000:08:00.1
        representor-PCI: 0000:08:00.5
            ib_port                                 2
            host_index                              0
            pf_index                                1
            vf_index                                0
            pci_func_type                           VF
            hotplug                                 no
            vuid                                    MT2228X06397MLNXS0D0F1VF1
            iface_name                              eth10
            iface_index                             13
        representor-PCI: 0000:08:00.6
            ib_port                                 3
            host_index                              0
            pf_index                                1
            vf_index                                1
            pci_func_type                           VF
            hotplug                                 no
            vuid                                    MT2228X06397MLNXS0D0F1VF2
            iface_name                              eth11
            iface_index                             14
        representor-PCI: 0000:08:00.1
            ib_port                                 5
            host_index                              0
            pf_index                                1
            sf_index                                4
            pci_func_type                           SF
            hotplug                                 no
            vuid                                    MT2228X06397MLNXS0D0F1SF32768
            iface_name                              en8f1pf1sf4
            iface_index                             19
        representor-PCI: 0000:08:00.1
            ib_port                                 6
            host_index                              0
            pf_index                                1
            sf_index                                8
            pci_func_type                           SF
            hotplug                                 no
            vuid                                    MT2228X06397MLNXS0D0F1SF32769
            iface_name                              en8f1pf1sf8
            iface_index                             21
PCI: 0000:08:00.2
PCI: 0000:08:00.3
PCI: 0000:08:00.4
PCI: 0000:08:00.5
PCI: 0000:08:00.6
PCI: 0000:08:00.0
PCI: 0000:08:00.1
PCI: 0000:08:00.1
```

- To print all the supported DOCA libraries by the OS and their availability status, run:

```
/opt/mellanox/doca/tools/doca_caps --list-libs
```

Different OSs may support different DOCA libraries.

Example output:

```
/opt/mellanox/doca/tools/doca_caps --list-libs
    common                                      installed
    aes_gcm                                     installed
    apsh                                        installed
    argp                                        installed
    cc                                          installed
    comm_channel                                installed
    compress                                    installed
    dma                                         installed
    dpa                                         installed
    dpdk_bridge                                 installed
    erasure_coding                              installed
    eth                                         installed
    ipsec                                       installed
    flow                                        installed
    flow_ct                                     installed
    pcc                                         installed
    rdma                                        installed
    sha                                         installed
    telemetry                                   installed
```

- To print all the capabilities for all the available libraries, that have capabilities, for every DOCA device, run:

```
/opt/mellanox/doca/tools/doca_caps
```

Printing the capabilities of one specific DOCA device can be done using the **--pci-addr** flag.

Printing the capabilities of one specific DOCA library can be done using the **--lib** flag.

Example output:

```
/opt/mellanox/doca/tools/doca_caps
PCI: 0000:03:00.0
    common
        mmap_export_pci                         supported
        mmap_create_from_export_pci             supported
        hotplug_manager                         unsupported
        rep_filter_all                          supported
        rep_filter_net                          supported
        rep_filter_emulated                     unsupported
    aes_gcm
        task_encrypt                            supported
        task_encrypt_get_max_iv_len             12
        task_encrypt_tag_96                     supported
        task_encrypt_tag_128                    supported
        task_encrypt_128b_key                   supported
        task_encrypt_256b_key                   supported
        task_encrypt_max_buf_size               2097152
        task_encrypt_max_list_buf_num_elem      128
        task_decrypt                            supported
        task_decrypt_get_max_iv_len             12
        task_decrypt_tag_96                     supported
        task_decrypt_tag_128                    supported
        task_decrypt_128b_key                   supported
        task_decrypt_256b_key                   supported
        task_decrypt_max_buf_size               2097152
        task_decrypt_max_list_buf_num_elem      128
        max_num_tasks                           65536
    cc
        server                                  supported
        client                                  supported
        max_name_len                            120
        max_msg_size                            4080
        max_recv_queue_size                     8192
        max_send_tasks                          8192
        max_clients                             512
        consumer                                supported
        consumer_max_num_tasks                  65536
        consumer_max_buf_size                   2097152
        producer                                supported
```

```
        producer_max_num_tasks                          65536
        producer_max_buf_size                           2097152
    comm_channel
        max_service_name_len                            120
        max_message_size                                4080
        max_send_queue_size                             8192
        max_recv_queue_size                             8192
        service_max_num_connections                     512
    compress
        task_compress_deflate                           unsupported
        task_compress_deflate_get_max_buf_size          0
        task_compress_deflate_get_max_buf_list_len      0
        task_decompress_deflate                         supported
        task_decompress_deflate_get_max_buf_size        2097152
        task_decompress_deflate_get_max_buf_list_len    128
        task_decompress_lz4                             supported
        task_decompress_lz4_get_max_buf_size            2097152
        task_decompress_lz4_get_max_buf_list_len        128
        max_num_tasks                                   65536
    dma
        task_memcpy                                     supported
        max_buf_size                                    2097152
        max_buf_list_len                                64
        max_num_tasks                                   65536
    dpa
        dpa                                             supported
        max_threads_per_kernel                          128
        kernel_max_run_time                             12
    erasure_coding
        task_galois_mul                                 supported
        task_create                                     supported
        task_update                                     supported
        task_recover                                    supported
        max_block_size                                  1048576
        max_buf_list_len                                128
    eth
        rxq_cyclic_cpu                                  unsupported
        rxq_cyclic_gpu                                  supported
        rxq_managed_mempool_cpu                         unsupported
        rxq_managed_mempool_gpu                         supported
        rxq_regular_cpu                                 unsupported
        rxq_regular_gpu                                 supported
        rxq_max_recv_buf_list_len                       32
        rxq_max_packet_size                             16384
        rxq_max_burst_size                              32768
        txq_regular_cpu                                 unsupported
        txq_regular_gpu                                 supported
        txq_max_send_buf_list_len                       48
        txq_max_lso_header_size                         256
        txq_txq_max_lso_msg_size                        262144
        txq_l3_chksum_offload                           supported
        txq_l4_chksum_offload                           supported
        txq_wait_on_time_type                           unsupported
    flow_ct
        flow_ct                                         supported
    ipsec
        task_sa_create                                  supported
        task_sa_destroy                                 supported
    nvrd_transport
        task_write                                      supported
        rc_max_src_buf_list_len                         0
        dc_max_src_buf_list_len                         0
    pcc
        pcc                                             unsupported
        pcc_np                                          unsupported
        min_num_threads                                 0
        max_num_threads                                 0
    rdma
        task_send                                       supported
        task_send_imm                                   supported
        task_read                                       supported
        task_write                                      supported
        task_write_imm                                  supported
        task_atomic_cmp_swp                             supported
        task_atomic_fetch_add                           supported
        task_receive                                    supported
        rc_transport_type                               supported
        dc_transport_type                               unsupported
        rc_task_receive_get_max_dst_buf_list_len        31
        dc_task_receive_get_max_dst_buf_list_len        0
        task_remote_net_sync_event_get                  supported
        task_remote_net_sync_event_notify_set           supported
        task_remote_net_sync_event_notify_add           supported
        max_send_queue_size                             32768
        max_recv_queue_size                             32768
        max_send_buf_list_len                           13
        max_message_size                                1073741824
    sha
        sha1                                            unsupported
        sha256                                          unsupported
        sha512                                          unsupported
        sha1_partial                                    unsupported
        sha256_partial                                  unsupported
        sha512_partial                                  unsupported
        max_list_num_elem                               0
        max_src_buf_size                                0
        sha1_min_dst_buf_size                           0
        sha256_min_dst_buf_size                         0
        sha512_min_dst_buf_size                         0
        sha1_partial_hash_block_size                    0
        sha256_partial_hash_block_size                  0
        sha512_partial_hash_block_size                  0
PCI: 0000:03:00.1
    common
        mmap_export_pci                                 supported
```

```
    mmap_create_from_export_pci                 supported
    hotplug_manager                             unsupported
    rep_filter_all                              supported
    rep_filter_net                              supported
    rep_filter_emulated                         unsupported
aes_gcm
    task_encrypt                                supported
    task_encrypt_get_max_iv_len                 12
    task_encrypt_tag_96                         supported
    task_encrypt_tag_128                        supported
    task_encrypt_128b_key                       supported
    task_encrypt_256b_key                       supported
    task_encrypt_max_buf_size                   2097152
    task_encrypt_max_list_buf_num_elem          128
    task_decrypt                                supported
    task_decrypt_get_max_iv_len                 12
    task_decrypt_tag_96                         supported
    task_decrypt_tag_128                        supported
    task_decrypt_128b_key                       supported
    task_decrypt_256b_key                       supported
    task_decrypt_max_buf_size                   2097152
    task_decrypt_max_list_buf_num_elem          128
    max_num_tasks                               65536
cc
    server                                      supported
    client                                      supported
    max_name_len                                120
    max_msg_size                                4080
    max_recv_queue_size                         8192
    max_send_tasks                              8192
    max_clients                                 512
    consumer                                    supported
    consumer_max_num_tasks                      65536
    consumer_max_buf_size                       2097152
    producer                                    supported
    producer_max_num_tasks                      65536
    producer_max_buf_size                       2097152
comm_channel
    max_service_name_len                        120
    max_message_size                            4080
    max_send_queue_size                         8192
    max_recv_queue_size                         8192
    service_max_num_connections                 512
compress
    task_compress_deflate                       unsupported
    task_compress_deflate_get_max_buf_size      0
    task_compress_deflate_get_max_buf_list_len  0
    task_decompress_deflate                     supported
    task_decompress_deflate_get_max_buf_size    2097152
    task_decompress_deflate_get_max_buf_list_len 128
    task_decompress_lz4                         supported
    task_decompress_lz4_get_max_buf_size        2097152
    task_decompress_lz4_get_max_buf_list_len    128
    max_num_tasks                               65536
dma
    task_memcpy                                 supported
    max_buf_size                                2097152
    max_buf_list_len                            64
    max_num_tasks                               65536
dpa
    dpa                                         supported
    max_threads_per_kernel                      128
    kernel_max_run_time                         12
erasure_coding
    task_galois_mul                             supported
    task_create                                 supported
    task_update                                 supported
    task_recover                                supported
    max_block_size                              1048576
    max_buf_list_len                            128
eth
    rxq_cyclic_cpu                              unsupported
    rxq_cyclic_gpu                              supported
    rxq_managed_mempool_cpu                     unsupported
    rxq_managed_mempool_gpu                     supported
    rxq_regular_cpu                             unsupported
    rxq_regular_gpu                             supported
    rxq_max_recv_buf_list_len                   32
    rxq_max_packet_size                         16384
    rxq_max_burst_size                          32768
    txq_regular_cpu                             unsupported
    txq_regular_gpu                             supported
    txq_max_send_buf_list_len                   48
    txq_max_lso_header_size                     256
    txq_txq_max_lso_msg_size                    262144
    txq_l3_chksum_offload                       supported
    txq_l4_chksum_offload                       supported
    txq_wait_on_time_type                       unsupported
flow_ct
    flow_ct                                     supported
ipsec
    task_sa_create                              supported
    task_sa_destroy                             supported
nvrd_transport
    task_write                                  supported
    rc_max_src_buf_list_len                     0
    dc_max_src_buf_list_len                     0
pcc
    pcc                                         unsupported
    pcc_np                                      unsupported
    min_num_threads                             0
    max_num_threads                             0
rdma
    task_send                                   supported
    task_send_imm                               supported
```

```
        task_read                                       supported
        task_write                                      supported
        task_write_imm                                  supported
        task_atomic_cmp_swp                             supported
        task_atomic_fetch_add                           supported
        task_receive                                    supported
        rc_transport_type                               supported
        dc_transport_type                               unsupported
        rc_task_receive_get_max_dst_buf_list_len        31
        dc_task_receive_get_max_dst_buf_list_len        0
        task_remote_net_sync_event_get                  supported
        task_remote_net_sync_event_notify_set           supported
        task_remote_net_sync_event_notify_add           supported
        max_send_queue_size                             32768
        max_recv_queue_size                             32768
        max_send_buf_list_len                           13
        max_message_size                                1073741824
    sha
        sha1                                            unsupported
        sha256                                          unsupported
        sha512                                          unsupported
        sha1_partial                                    unsupported
        sha256_partial                                  unsupported
        sha512_partial                                  unsupported
        max_list_num_elem                               0
        max_src_buf_size                                0
        sha1_min_dst_buf_size                           0
        sha256_min_dst_buf_size                         0
        sha512_min_dst_buf_size                         0
        sha1_partial_hash_block_size                    0
        sha256_partial_hash_block_size                  0
        sha512_partial_hash_block_size                  0
PCI: 0000:03:00.0
    common
        mmap_export_pci                                 supported
        mmap_create_from_export_pci                     supported
        hotplug_manager                                 unsupported
        rep_filter_all                                  unsupported
        rep_filter_net                                  unsupported
        rep_filter_emulated                             unsupported
    aes_gcm
        task_encrypt                                    supported
        task_encrypt_get_max_iv_len                     12
        task_encrypt_tag_96                             supported
        task_encrypt_tag_128                            supported
        task_encrypt_128b_key                           supported
        task_encrypt_256b_key                           supported
        task_encrypt_max_buf_size                       2097152
        task_encrypt_max_list_buf_num_elem              128
        task_decrypt                                    supported
        task_decrypt_get_max_iv_len                     12
        task_decrypt_tag_96                             supported
        task_decrypt_tag_128                            supported
        task_decrypt_128b_key                           supported
        task_decrypt_256b_key                           supported
        task_decrypt_max_buf_size                       2097152
        task_decrypt_max_list_buf_num_elem              128
        max_num_tasks                                   65536
    cc
        server                                          unsupported
        client                                          supported
        max_name_len                                    120
        max_msg_size                                    4080
        max_recv_queue_size                             8192
        max_send_tasks                                  8192
        max_clients                                     0
        consumer                                        supported
        consumer_max_num_tasks                          65536
        consumer_max_buf_size                           2097152
        producer                                        supported
        producer_max_num_tasks                          65536
        producer_max_buf_size                           2097152
    comm_channel
        max_service_name_len                            120
        max_message_size                                4080
        max_send_queue_size                             8192
        max_recv_queue_size                             8192
        service_max_num_connections                     0
    compress
        task_compress_deflate                           unsupported
        task_compress_deflate_get_max_buf_size          0
        task_compress_deflate_get_max_buf_list_len      0
        task_decompress_deflate                         supported
        task_decompress_deflate_get_max_buf_size        2097152
        task_decompress_deflate_get_max_buf_list_len    128
        task_decompress_lz4                             supported
        task_decompress_lz4_get_max_buf_size            2097152
        task_decompress_lz4_get_max_buf_list_len        128
        max_num_tasks                                   65536
    dma
        task_memcpy                                     supported
        max_buf_size                                    2097152
        max_buf_list_len                                64
        max_num_tasks                                   65536
    dpa
        dpa                                             supported
        max_threads_per_kernel                          128
        kernel_max_run_time                             12
    erasure_coding
        task_galois_mul                                 supported
        task_create                                     supported
        task_update                                     supported
        task_recover                                    supported
        max_block_size                                  1048576
        max_buf_list_len                                128
```

```
    eth
        rxq_cyclic_cpu                                  supported
        rxq_cyclic_gpu                                  supported
        rxq_managed_mempool_cpu                         supported
        rxq_managed_mempool_gpu                         supported
        rxq_regular_cpu                                 supported
        rxq_regular_gpu                                 supported
        rxq_max_recv_buf_list_len                       32
        rxq_max_packet_size                             16384
        rxq_max_burst_size                              32768
        txq_regular_cpu                                 supported
        txq_regular_gpu                                 supported
        txq_max_send_buf_list_len                       48
        txq_max_lso_header_size                         256
        txq_txq_max_lso_msg_size                        262144
        txq_l3_chksum_offload                           supported
        txq_l4_chksum_offload                           supported
        txq_wait_on_time_type                           unsupported
    flow_ct
        flow_ct                                         unsupported
    ipsec
        task_sa_create                                  unsupported
        task_sa_destroy                                 unsupported
    nvrd_transport
        task_write                                      supported
        rc_max_src_buf_list_len                         0
        dc_max_src_buf_list_len                         0
    pcc
        pcc                                             unsupported
        pcc_np                                          unsupported
        min_num_threads                                 0
        max_num_threads                                 0
    rdma
        task_send                                       supported
        task_send_imm                                   supported
        task_read                                       supported
        task_write                                      supported
        task_write_imm                                  supported
        task_atomic_cmp_swp                             supported
        task_atomic_fetch_add                           supported
        task_receive                                    supported
        rc_transport_type                               supported
        dc_transport_type                               unsupported
        rc_task_receive_get_max_dst_buf_list_len        31
        dc_task_receive_get_max_dst_buf_list_len        0
        task_remote_net_sync_event_get                  supported
        task_remote_net_sync_event_notify_set           supported
        task_remote_net_sync_event_notify_add           supported
        max_send_queue_size                             32768
        max_recv_queue_size                             32768
        max_send_buf_list_len                           13
        max_message_size                                1073741824
    sha
        sha1                                            unsupported
        sha256                                          unsupported
        sha512                                          unsupported
        sha1_partial                                    unsupported
        sha256_partial                                  unsupported
        sha512_partial                                  unsupported
        max_list_num_elem                               0
        max_src_buf_size                                0
        sha1_min_dst_buf_size                           0
        sha256_min_dst_buf_size                         0
        sha512_min_dst_buf_size                         0
        sha1_partial_hash_block_size                    0
        sha256_partial_hash_block_size                  0
        sha512_partial_hash_block_size                  0
PCI: 0000:03:00.1
    common
        mmap_export_pci                                 supported
        mmap_create_from_export_pci                     supported
        hotplug_manager                                 unsupported
        rep_filter_all                                  unsupported
        rep_filter_net                                  unsupported
        rep_filter_emulated                             unsupported
    aes_gcm
        task_encrypt                                    supported
        task_encrypt_get_max_iv_len                     12
        task_encrypt_tag_96                             supported
        task_encrypt_tag_128                            supported
        task_encrypt_128b_key                           supported
        task_encrypt_256b_key                           supported
        task_encrypt_max_buf_size                       2097152
        task_encrypt_max_list_buf_num_elem              128
        task_decrypt                                    supported
        task_decrypt_get_max_iv_len                     12
        task_decrypt_tag_96                             supported
        task_decrypt_tag_128                            supported
        task_decrypt_128b_key                           supported
        task_decrypt_256b_key                           supported
        task_decrypt_max_buf_size                       2097152
        task_decrypt_max_list_buf_num_elem              128
        max_num_tasks                                   65536
    cc
        server                                          unsupported
        client                                          supported
        max_name_len                                    120
        max_msg_size                                    4080
        max_recv_queue_size                             8192
        max_send_tasks                                  8192
        max_clients                                     0
        consumer                                        supported
        consumer_max_num_tasks                          65536
        consumer_max_buf_size                           2097152
        producer                                        supported
```

```
        producer_max_num_tasks                          65536
        producer_max_buf_size                           2097152
    comm_channel
        max_service_name_len                            120
        max_message_size                                4080
        max_send_queue_size                             8192
        max_recv_queue_size                             8192
        service_max_num_connections                     0
    compress
        task_compress_deflate                           unsupported
        task_compress_deflate_get_max_buf_size          0
        task_compress_deflate_get_max_buf_list_len      0
        task_decompress_deflate                         supported
        task_decompress_deflate_get_max_buf_size        2097152
        task_decompress_deflate_get_max_buf_list_len    128
        task_decompress_lz4                             supported
        task_decompress_lz4_get_max_buf_size            2097152
        task_decompress_lz4_get_max_buf_list_len        128
        max_num_tasks                                   65536
    dma
        task_memcpy                                     supported
        max_buf_size                                    2097152
        max_buf_list_len                                64
        max_num_tasks                                   65536
    dpa
        dpa                                             supported
        max_threads_per_kernel                          128
        kernel_max_run_time                             12
    erasure_coding
        task_galois_mul                                 supported
        task_create                                     supported
        task_update                                     supported
        task_recover                                    supported
        max_block_size                                  1048576
        max_buf_list_len                                128
    eth
        rxq_cyclic_cpu                                  supported
        rxq_cyclic_gpu                                  supported
        rxq_managed_mempool_cpu                         supported
        rxq_managed_mempool_gpu                         supported
        rxq_regular_cpu                                 supported
        rxq_regular_gpu                                 supported
        rxq_max_recv_buf_list_len                       32
        rxq_max_packet_size                             16384
        rxq_max_burst_size                              32768
        txq_regular_cpu                                 supported
        txq_regular_gpu                                 supported
        txq_max_send_buf_list_len                       48
        txq_max_lso_header_size                         256
        txq_txq_max_lso_msg_size                        262144
        txq_l3_chksum_offload                           supported
        txq_l4_chksum_offload                           supported
        txq_wait_on_time_type                           unsupported
    flow_ct
        flow_ct                                         unsupported
    ipsec
        task_sa_create                                  unsupported
        task_sa_destroy                                 unsupported
    nvrd_transport
        task_write                                      supported
        rc_max_src_buf_list_len                         0
        dc_max_src_buf_list_len                         0
    pcc
        pcc                                             unsupported
        pcc_np                                          unsupported
        min_num_threads                                 0
        max_num_threads                                 0
    rdma
        task_send                                       supported
        task_send_imm                                   supported
        task_read                                       supported
        task_write                                      supported
        task_write_imm                                  supported
        task_atomic_cmp_swp                             supported
        task_atomic_fetch_add                           supported
        task_receive                                    supported
        rc_transport_type                               supported
        dc_transport_type                               unsupported
        rc_task_receive_get_max_dst_buf_list_len        31
        dc_task_receive_get_max_dst_buf_list_len        0
        task_remote_net_sync_event_get                  supported
        task_remote_net_sync_event_notify_set           supported
        task_remote_net_sync_event_notify_add           supported
        max_send_queue_size                             32768
        max_recv_queue_size                             32768
        max_send_buf_list_len                           13
        max_message_size                                1073741824
    sha
        sha1                                            unsupported
        sha256                                          unsupported
        sha512                                          unsupported
        sha1_partial                                    unsupported
        sha256_partial                                  unsupported
        sha512_partial                                  unsupported
        max_list_num_elem                               0
        max_src_buf_size                                0
        sha1_min_dst_buf_size                           0
        sha256_min_dst_buf_size                         0
        sha512_min_dst_buf_size                         0
        sha1_partial_hash_block_size                    0
        sha256_partial_hash_block_size                  0
        sha512_partial_hash_block_size                  0
```

- To print the available logger names of DOCA libraries, run:

```
/opt/mellanox/doca/tools/doca_caps --list-loggers
```

Example output:

```
/opt/mellanox/doca/tools/doca_caps --list-loggers
    CORE
    ARGP
    AES_GCM
    DMA
    APSH
    COMCH
    COMPRESS
    DEVEMU
    DPA
    DPDK_BRIDGE
    ERASURE_CODING
    ETH
    FLOW
    FLOW::DRIVER
    FLOW::DRIVER::RUNTIME
    MGMT
    PCC
    RDMA
    SHA
    TELEMETRY
    UROM
```

7.3 DOCA Comm Channel Admin Tool

This document provides instructions on the usage of the DOCA Comm Channel Admin Tool.

7.3.1 Introduction

The Comm Channel Admin Tool is used to print a snapshot of DOCA Comch (comm channel) connections:

- On the BlueField Arm side, it includes DOCA Comch servers and their current connection information
- On the host side, it includes all active client connections and the server they are connected to
- Only client-to-server control channels are reported; fast path producer/consumer channels are not.

7.3.2 Prerequisites

The Comm Channel Admin Tool is for Linux only and requires an up-to-date BFB bundle or DOCA host packages of at least 2.7, which include in the Resource dump binary.

7.3.3 Description and Execution

The Comm Channel Admin Tool can be executed on the host or Arm CPUs. By default, the tool scans all available PCIe slots to detect supported DOCA devices and reports any Comch information available.

The tool can be run on BlueField Arm or x86 host using the following command:

```
/opt/mellanox/doca/tools/doca_comm_channel_admin
```

7.3.3.1 Sample Output from BlueField Arm

On the BlueField Arm side, any active DOCA Comch servers are be reported:

```
SERVERS:
+-------------------------------------------------------------------------------+
| Server name      | PID        | Connections   | PCIe          | Interface Name    |
+===============================================================================+
| comch1           | 1898009    | 2/512         | 0000:03:00.0  | p0                |
+-------------------------------------------------------------------------------+
| comch3           | 1898011    | 1/512         | 0000:03:00.0  | p0                |
+-------------------------------------------------------------------------------+
| comch6           | 1898014    | 3/512         | 0000:03:00.0  | p0                |
+-------------------------------------------------------------------------------+
| comch2           | 1898010    | 1/512         | 0000:03:00.0  | p0                |
+-------------------------------------------------------------------------------+
| comch7           | 1898015    | 1/512         | 0000:03:00.0  | p0                |
+-------------------------------------------------------------------------------+
| comch5           | 1898013    | 4/512         | 0000:03:00.0  | p0                |
+-------------------------------------------------------------------------------+
| comch8           | 1898016    | 2/512         | 0000:03:00.0  | p0                |
+-------------------------------------------------------------------------------+
| comch4           | 1898012    | 0/512         | 0000:03:00.0  | p0                |
+-------------------------------------------------------------------------------+
```

The following information is available:

- Server Name – the name assigned to the server
- PID – the Linux process ID of the application which created the server
- Connections – the number of connections active on the server out of the total allowed (e.g., 2/512 means 2 active connections of a maximum of 512)
- PCIe – the PCIe address of the device which the server has been detected on
- Interface Name – the interface name associated with the PCIe address

> ⚠ Connections may also be displayed on the BlueField Arm like on x86. This occurs if SF ports are detected here. The interface name associated with the PCIe address indicates the SF port.

7.3.3.2 Sample Output from x86

The x86 host cannot run DOCA Comch servers. Therefore, individual client connections are reported:

```
CONNECTIONS:
+-------------------------------------------------------------------+
| Server name      | PID        | PCIe          | Interface Name     |
+===================================================================+
| comch6           | 299693     | 0000:3b:00.0  | ens1f0np0          |
+-------------------------------------------------------------------+
| comch3           | 299688     | 0000:3b:00.0  | ens1f0np0          |
+-------------------------------------------------------------------+
| comch2           | 299687     | 0000:3b:00.0  | ens1f0np0          |
+-------------------------------------------------------------------+
| comch5           | 299689     | 0000:3b:00.0  | ens1f0np0          |
+-------------------------------------------------------------------+
| comch5           | 299692     | 0000:3b:00.0  | ens1f0np0          |
+-------------------------------------------------------------------+
| comch7           | 299696     | 0000:3b:00.0  | ens1f0np0          |
+-------------------------------------------------------------------+
| comch5           | 299690     | 0000:3b:00.0  | ens1f0np0          |
+-------------------------------------------------------------------+
| comch6           | 299694     | 0000:3b:00.0  | ens1f0np0          |
+-------------------------------------------------------------------+
| comch8           | 299697     | 0000:3b:00.0  | ens1f0np0          |
+-------------------------------------------------------------------+
| comch6           | 299695     | 0000:3b:00.0  | ens1f0np0          |
+-------------------------------------------------------------------+
| comch8           | 299698     | 0000:3b:00.0  | ens1f0np0          |
+-------------------------------------------------------------------+
| comch1           | 299686     | 0000:3b:00.0  | ens1f0np0          |
+-------------------------------------------------------------------+
| comch5           | 299691     | 0000:3b:00.0  | ens1f0np0          |
+-------------------------------------------------------------------+
| comch1           | 299685     | 0000:3b:00.0  | ens1f0np0          |
+-------------------------------------------------------------------+
```

The following information is available:

- Server Name – the name of the BlueField Arm server that a client has connected to
- PID – the Linux process ID of the application running a DOCA Comch client
- PCIe – the PCIe address of the BlueField networking platform which the destination server is running on
- Interface Name – the interface name associated with the PCIe address

7.4 DPA Tools

7.4.1 Introduction

DPA tools are a set of executables that enable the DPA application developer and the system administrator to manage and monitor DPA resources and to debug DPA applications.

7.4.2 List of Tools

Tool	CLI Name
DPACC Compiler	`dpacc`
DPA Resource Management	`dpa-resource-mgmt`
DPA GDB Server	`dpa-gdbserver`
DPA PS	`dpa-ps`
DPA Statistic	`dpa-statistics`
DPA Application Security Attributes to Blob	`dpa-app-attributes2blob`

7.4.3 DOCA DPACC Compiler

This document describes DOCA DPACC compiler and provides instructions about DPA toolchain setup and usage.

7.4.3.1 Introduction

DPACC is a high-level compiler for the DPA processor which compiles code targeted for the data-path accelerator (DPA) processor into a device executable and generates a DPA program.

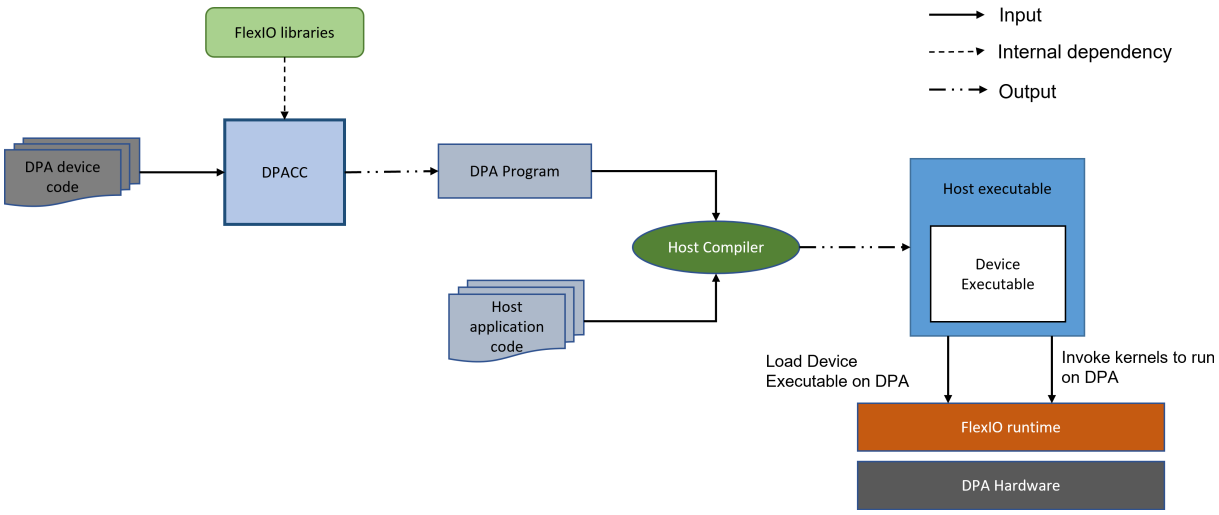
The DPA program is a host library with interfaces encapsulating the device executable. This DPA program is linked with the host application to generate a host executable. The host executable can invoke the DPA code through FlexIO runtime API.

DPACC uses DPA compiler (`dpa-clang`) to compile code targeted for DPA. `dpa-clang` is part of the DPA toolchain package which is an LLVM-based, cross-compiling, bare-metal toolchain. It provides Clang compiler, LLD linker targeting DPA architecture, and other utilities.

Glossary

Term	Definition
Device	DPA as present on the NVIDIA® BlueField® DPU
Host	CPU that launches the device code to run on the DPA
Device function	Any C function that runs on the DPA device
DPA global function	Device function that is the point of entry when offloading any work on DPA
Host compiler	Compiler used to compile the code targeting the host CPU
Device compiler	Compiler used to compile code targeting the DPA
Fatbinary	File that contains code for multiple target DPA architectures
DPA program	Host library that encapsulates the DPA device executable (`.elf`) and host stubs which are used to access the device executable

7.4.3.1.1 Offloading Work on DPA



To invoke a DPA function from a host, the following things are required:

- DPA device code – C programs, targeted to run on the DPA. The DPA device code may contain one or more entry functions.
- Host application code – the corresponding host application. Refer to DPA Subsystem for more details
- Runtime – FlexIO or DOCA DPA library provides the runtime

The generated DPA program, when linked with a host application results in a host executable which also contains the device executable. The host application oversees loading the device executable on the device.

7.4.3.1.2 DPACC Predefined Macros

DPACC predefines the following macros:

__DPA__	Defined when compiling device code file
__NV_DPA	Defined to the target DPA hardware identifier macros See Architecture Macros for more details.
__DPA_MAJOR__	Defined to the major version number of DPACC
__DPA_MINOR__	Defined to the minor version number of DPACC
__DPA_PATCH__	Defined to the patch version number of DPACC

7.4.3.1.3 Writing DPA Applications

DPA device code is a C code with some restrictions and special definitions.

FlexIO or DOCA-DPA APIs provide interfaces to DPA.

7.4.3.1.3.1 Language Support

The DPA is programmed using a subset of the C11 language standard. The compiler documents any constructs that are not available. Language constructs, where available, retain their standard definitions.

7.4.3.1.3.2 Restrictions on DPA Code

- Use of C thread local storage is not allowed for any variables
- Identifiers with `_dpacc/__dpacc` prefix are reserved by the compiler. Use of such identifiers may result in an error or undefined behavior
- DPA processor does not have native floating-point support; floating-point operations are disabled

7.4.3.1.3.3 DPA RPC Functions

A remote procedure call function is a synchronous call that triggers work in DPA and waits for its completion. These functions return a type `uint64_t` value. They are annotated with a `__dpa_rpc__` attribute.

7.4.3.1.3.4 DPA Global Functions

A DPA global function is an event handler device function referenced from the host code. These functions do not return anything. They are annotated with a `__dpa_global__` attribute.

For more information, refer to DPA Subsystem.

7.4.3.1.3.5 Characteristics of Annotated Functions

- Global functions must have `void` return type and RPC functions must have `uint64_t` return type
- Annotated functions cannot accept C pointers and arrays as arguments (for example, `void glob_func (int *ptr, int arr[])`)
- Annotated functions cannot accept a variable number of arguments
- Inline specifier is not allowed on annotated functions

7.4.3.1.3.6 Handling User-defined Data Types

User-defined data types, when used as global function arguments, require special handling. They must be annotated with a `__dpa_global__` attribute.

If the user-defined data type is `typedef` 'd, the `typedef` statement must be annotated with a `__dpa_global__` attribute along the data type itself.

7.4.3.1.3.7 Characteristics of Annotated Types

- They must have a copy of the definition in all translation units where they are used as global function arguments
- They cannot have pointers, variable length arrays, and flexible arrays as members
- Fixed-size arrays as C structure members are supported

- These characteristics apply recursively to any user-defined/ **typedef** 'd types that are members of an annotated type

DPACC processes all annotated functions along with annotated types and generates host and device interfaces to facilitate the function launch.

7.4.3.1.3.8 DPA Intrinsics

DPA features such as fences and processor-specific instructions are exposed via intrinsics by the DPA compiler. All intrinsics defined in the header file **dpaintrin.h** are guarded by the **DPA_INTRIN_VERSION_USED** macro. The current **DPA_INTRIN_VERSION** is **1.3**.

Example:

```
#define DPA_INTRIN_VERSION_USED (DPA_INTRIN_VERSION(1, 3))
#include <dpaintrin.h>
…
__dpa_thread_writeback_window();   // Fence for write barrier
```

For more information, refer to DPA Subsystem.

7.4.3.2 Prerequisites

Package	Instructions
Host compiler	Compiler specified through **hostcc** option. Both **gcc** and **clang** are supported. ⚠ Minimum supported version for **clang** as **hostcc** is **clang 3.8.0**.
Device compiler	The default device compiler is "DPA compiler". Installing the DPACC package also installs the DPA compiler binaries **dpa-clang**, **dpa-ar**, **dpa-nm** and **dpa-objdump**. ⚠ **dpa-clang** is the only supported device compiler.
FlexIO SDK and C library	Available as part of the DOCA software package. The DPA toolchain does not provide a C library and corresponding headers. Users are expected to use the C library for DPA from the FlexIO SDK.

7.4.3.3 Description

7.4.3.3.1 DPACC Inputs and Outputs

DPACC can produce DPA programs in a single command by accepting all source files as an input. DPACC also offers the flexibility of producing DPA object files or libraries from input files.

DPA object files contain both host-stub objects (DPACC-generated interfaces) and device objects. These DPA object files can later be given to DPACC as input to produce the DPA library.

Phase	Option Name	Default Output File Name
Compile input device code files to DPA object files	`--compile` or `-c`	`.dpa.o` appended to the name of each input source file
Compile and link the input device code files/ DPA object files, and produce a DPA program	No specific option	No default name, output file name must be specified
Compile and build DPA library from input device code files/DPA object files	`--gen-libs` or `-gen-libs`	No default name, output library name must be specified

DPACC can accept the following file types as input:

Input File Extension	File Type	Description
`.c`	C source file	DPA device code
`.dpa.o`	DPA object file	Object file generated by DPACC, containing both host and device objects
`.a`	DPA object archive	An archive of DPA object files. User can generate this archive from DPACC-generated DPA objects.

Based on the mode of operations, DPACC can generate the following output files:

Output File Type	Input Files
DPA object file	C source files
DPA program	C source files, DPA object files, and/or DPA object archives
DPA library (DPA host library and DPA device library)	C source files, DPA object files, and/or DPA object archives

The following provides the commands to generate different kinds of supported output file types for each input file type:

Input	Output	DPACC Command
C source file	DPA program	`dpacc -hostcc=<cc> -mcpu=<targets> in.c -o libprog.a`
	DPA object	`dpacc -hostcc=<cc> -mcpu=<targets> in.c -c`
	DPA library	`dpacc -hostcc=<cc> -mcpu=<targets> in.c -o lib<name> -gen-libs`
DPA object	DPA program	`dpacc -hostcc=<cc> -mcpu=<targets> in.dpa.o -o libprog.a`
	DPA library	`dpacc -hostcc=<cc> -mcpu=<targets> in.dpa.o -o lib<name> -gen-libs`
DPA object archive	DPA program	`dpacc -hostcc=<cc> -mcpu=<targets> in.a -o libprog.a`
	DPA library	`dpacc -hostcc=<cc> -mcpu=<targets> in.a -o lib<name> -gen-libs`

7.4.3.3.1.1 DPA Program

DPACC produces a DPA program in compile-and-link mode. A DPA program is a host library which contains:

- DPACC-generated host stubs which registering the DPA application and facilitate invoking an entry-point function from the host application
- Device executable, generated by DPACC by compiling and linking input DPA device code

DPA program library must be linked with the host application that contains appropriate runtime APIs to load the device executable onto the device.

A DPA program can contain device executables for multiple targets. All the target-specific device executables are encapsulated in a special fatbinary container format and this container is embedded as a section into the host object which is present in the host library.

7.4.3.3.1.2 DPA Object

DPACC produces DPA object files in compile-only mode. A DPA object is a host object file which has a similar layout to the DPA program where it contains DPACC-generated host stubs and device object which is generated by compiling input device code.

A DPA object can contain device objects for multiple targets, similar to a DPA program. All the target-specific device objects are encapsulated in a special fatbinary container format and this container is embedded as a section into the host object.

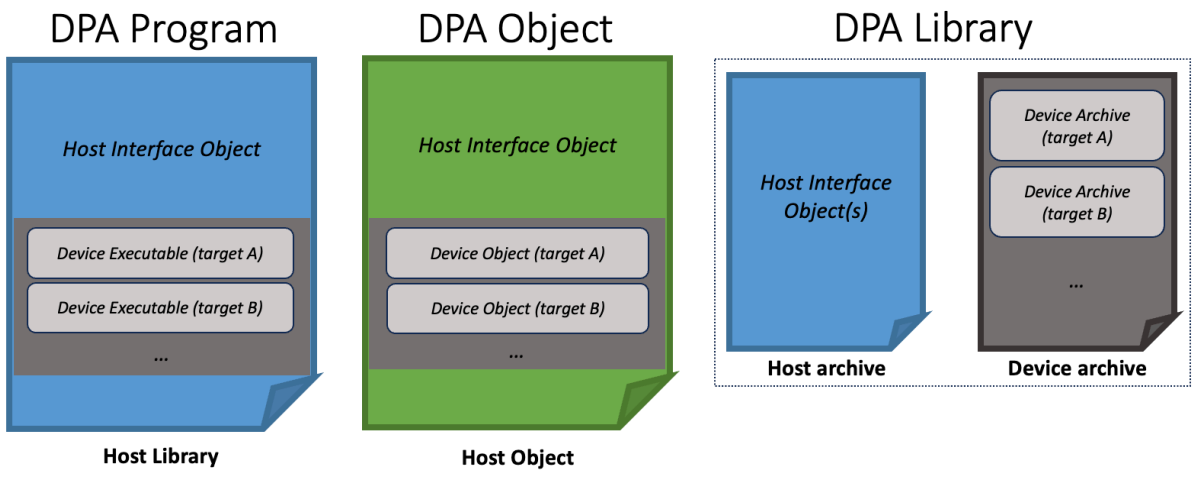
7.4.3.3.1.3 DPA Library

A DPA library is a collection of two separate libraries:

- DPA host library: contains host interface objects corresponding to the device objects in a DPA device library
- DPA device library: contains device objects generated by compiling the input device code files

The DPA device library is consumed by DPACC during DPA-program generation and the DPA host library can optionally be linked with other host code and be distributed as the host library. Both libraries are generated as static archives.

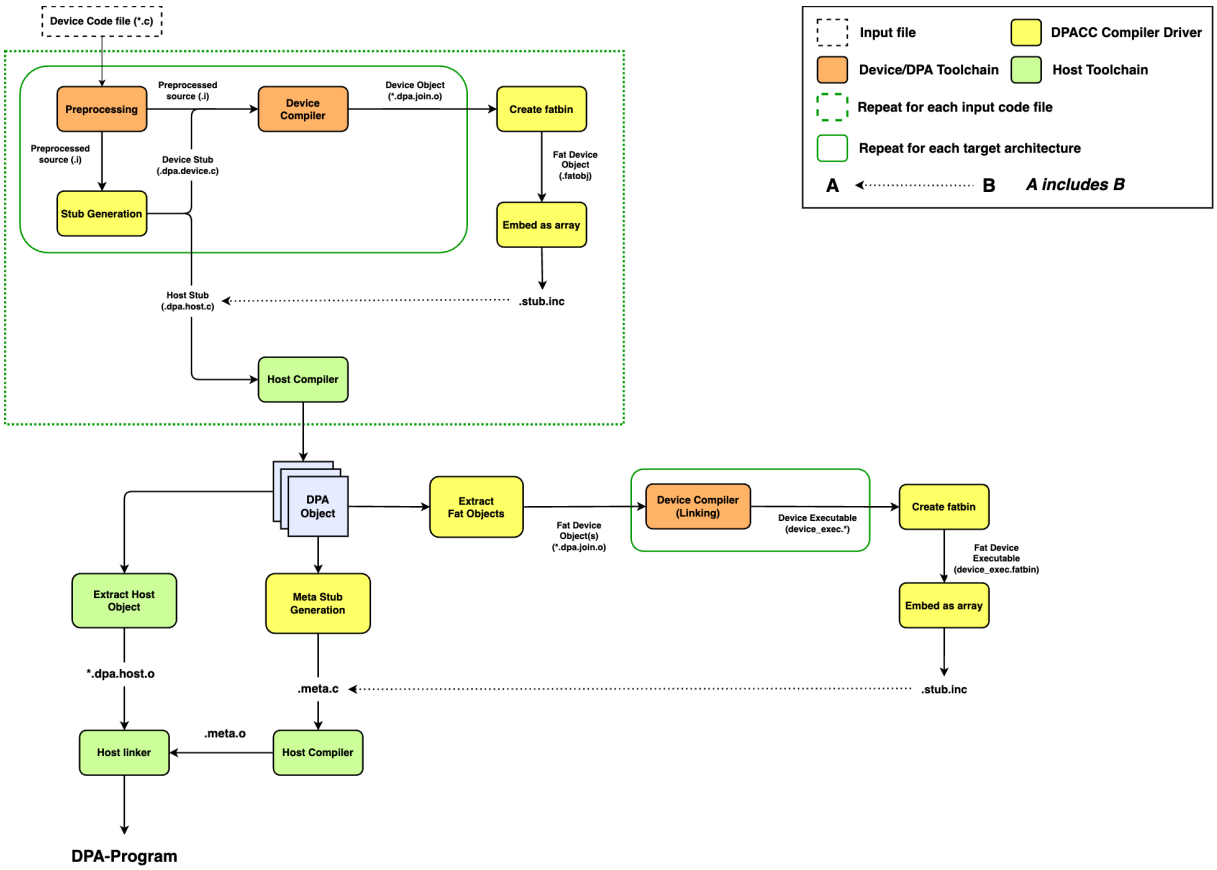
The host library is a static archive. The device library is a file of specialized fatbinary container format type which contains archives of device objects built for different targets.



The 'Device Archive' box in the preceding figure indicates the fatbinary container. The fatbinary container should be treated as an opaque object by the end-user and should not be manipulated in any way.

7.4.3.3.2 DPACC Trajectory

The following diagram illustrates DPACC compile-and-link mode trajectory.



7.4.3.3.3 Modes of Operation

In each of the below modes, DPACC accepts single or multiple target names through the `mcpu` option and builds the output such that all the mentioned targets are supported.

7.4.3.3.3.1 Compile-and-link Mode

This is a one-step mode that accepts C source files or DPA object files and produces the DPA program. Specifying the output library name is mandatory in this mode.

Example commands:

```
$ dpacc in1.c in2.c -o libapp1.a -hostcc=gcc -mcpu=nv-dpa-bf3                                  # Takes C sources to
produce libapp1.a library which supports a single target - nv-dpa-bf3
$ dpacc in3.dpa.o in4.dpa.o -o libapp2.a -hostcc=gcc -mcpu=nv-dpa-bf3,nv-dpa-cx8              # Takes DPA object files
to produce libapp2.a library which supports multiple targets - nv-dpa-bf3 and nv-dpa-cx8
```

```
$ dpacc in1.c in3.dpa.o -o libapp3.a -hostcc=gcc -mcpu=nv-dpa-bf3,nv-dpa-cx7,nv-dpa-cx8    # Takes C source and DPA
object to produce libapp3.a library which supports multiple targets - nv-dpa-bf3, nv-dpa-cx7 and nv-dpa-cx8
```

7.4.3.3.3.2 Compile-only Mode

This mode accepts C source code and produces `.dpa.o` object files. These files can be given to DPACC to produce the DPA program. The mode is invoked by the `--compile` or `-c` option.

To display the object file name, use the `--output-file` or `-o` option.

Example commands:

```
$ dpacc -c input1.c -hostcc=gcc -mcpu=nv-dpa-cx7                                        # Produces input1.dpa.o which
supports a single target - nv-dpa-cx7
$ dpacc -c input2.c -o in2.dpa.o -hostcc=gcc -mcpu=nv-dpa-cx8,nv-dpa-cx7                # Produces in2.dpa.o which
supports multiple targets - nv-dpa-cx7 and nv-dpa-cx8
$ dpacc -c input3.c input4.c -hostcc=gcc -mcpu=nv-dpa-bf3,nv-dpa-cx7,nv-dpa-cx8        # Produces input3.dpa.o and
input4.dpa.o which support multiple targets - nv-dpa-bf3, nv-dpa-cx7 and nv-dpa-cx8
```

7.4.3.3.3.3 Library Generation Mode

This mode accepts C source files or DPA object files and produces the DPA program. Specifying the output DPA library name is mandatory in this mode.

Example commands:

```
$ dpacc in1.c in2.c -o libdummy1 -hostcc=gcc -mcpu=nv-dpa-cx8 -gen-libs                                    # Takes
C sources to produce a DPA-Library (libdummy1_host.a and libdummy_device.a archives) which supports a single target
- nv-dpa-cx8
$ dpacc in3.dpa.o in4.dpa.o -o libdummy2 -hostcc=gcc -mcpu=nv-dpa-cx8,nv-dpa-bf3 -gen-libs                # Takes
DPA object files to produce a DPA-Library (libdummy2_host.a and libdummy2_device.a archives) which supports
multiple targets - nv-dpa-bf3 and nv-dpa-cx8
$ dpacc in1.c in3.dpa.o -o outdir/libdummy3 -hostcc=gcc -mcpu=nv-dpa-bf3,nv-dpa-cx7,nv-dpa-cx8 -gen-libs   # Takes
C source and DPA object to produce a DPA-Library (outdir/libdummy3_host.a and outdir/libdummy3_device.a archives)
which supports multiple targets - nv-dpa-bf3, nv-dpa-cx7 and nv-dpa-cx8
```

7.4.3.4 Execution

To execute DOCA DPACC compiler:

```
Usage: dpacc <list-of-input-files> -hostcc=<path> -mcpu=<targets> [other options]
Helper Flags:
  -h, --help                          Print help information about DPACC
  -V, --version                       Print DPACC version information
  -v, --verbose                       List the compilation commands generated by this invocation while also
executing every command in verbose mode
  -dryrun, --dryrun                   Only list the compilation commands generated by DPACC, without executing them
  -keep, --keep                       Keep all intermediate files that are generated during internal compilation
steps in the current directory
  -keep-dir, --keep-dir               Keep all intermediate files that are generated during internal compilation
steps in the given directory
  -optf, --options-file <file>,...    Include command line options from the specified file
```

7.4.3.4.1 Mandatory Arguments

Flag	DPACC Mode	Description
List of one or more input files	All	List of C source files or DPA object file names. Specifying at least one input file is mandatory. A file with an unknown extension is treated as a DPA object file.

Flag	DPACC Mode	Description
`-mcpu` , `--mcpu <targets>`	All	Specify the list of target DPA hardwares for code generation. (See DPA Hardware Architectures for more details) Multiple target names can be specified through this option. Supported values: `nv-dpa-bf3,nv-dpa-cx7, nv-dpa-cx8, nv-dpa-cx9`
`-hostcc` , `--hostcc <path>`	All	Specify the host compiler. This is typically the native compiler present on the host system. ⚠ The host compiler used to link the host application with the DPA program must be link-compatible with the `hostcc` compiler provided here.
`-o` , `--output-file <file>`	Compile-and-link/ library generation	Specify name and location of the output file.

7.4.3.4.2 Commonly Used Arguments

✓ Use the `--help` option to display a list of all supported options.

Flag	Description
`-app-name` , `--app-name <name>`	Specify DPA application name for the DPA program. This option is required if multiple DPA programs are part of a host application because each DPA application must have a unique name. Default name is `__dpa_a_out` .
`-flto` , `--flto`	Enable link-time optimization (LTO) for device code. Specify this option during compilation along with an optimization level in `devicecc-options` .
`-devicecc-options` , `--devicecc-options <options>` ,...	Specify the list of options to pass to the device compiler.
`-devicelink-options` , `--devicelink-options <options>` ,...	Specify the list of options to pass during device linking stage.
`-device-libs` , `--device-libs '-L<path> -l<name>'` ,...	Specify the list of device libraries including their names (in `-l`) and their paths (in `-L`). FlexIO libraries are linked by default.
`-I` , `--common-include-path <path>` ,...	Specify include search paths common to host and device code compilation. FlexIO headers paths are included by DPACC by default.

Flag	Description
`-o` , `--output-file <file>`	Specify name and location of the output file. • Compile-only mode: name of the output DPA object file. If not specified, `.dpa.o` is generated for each `.c` file. • Compiler-and-link mode: name of the output DPA program. This is a mandatory option in compiler-and-link mode. • Library generation mode: name of the output library. This is a mandatory option for this mode. Output files `<name>_device.a` and `<name>_host.a` are generated.
`-hostcc-options` , `--hostcc-options <options>` ,...	Specify the list of options to pass to the host compiler.
`-gen-libs` , `--gen-libs`	Generate a DPA library from input files
`-ldoca_dpa` , `--ldoca_dpa`	Link with DOCA-DPA libraries

⚠ Using machine-dependent options through `-devicecc-options` to influence compiler code generation is not supported. Examples of unsupported options through `-devicecc-options` are `-mcpu, -march, -mabi` , etc.

⚠ The `devicecc-options` option allows passing any option to the device compiler. However, passing options that prevent compilation of the input file may lead to unexpected behavior (for example, `-devicecc-options="-version"` makes the device compiler print the version and not process input files).

⚠ Incompatible options that affect DPA global function argument sizes during DPACC invocation and host application compilation may lead to undefined behavior during execution (for example, passing `-hostcc-options="-fshort-enums"` to DPACC and missing this option when building the host application).

7.4.3.4.3 DPA Hardware Architectures

The following table indicates the DPA architectures, the associated values supported in the compiler through the `-mcpu` option, and the macros defined by the compiler to identify these architectures.

Hardware Name	Value	Macro
NVIDIA® ConnectX®-7	`nv-dpa-cx7`	`__NV_DPA_CX7`
NVIDIA® BlueField®-3	`nv-dpa-bf3`	`__NV_DPA_BF3`
NVIDIA® ConnectX®-8	`nv-dpa-cx8`	`__NV_DPA_CX8`
NVIDIA® ConnectX®-9	`nv-dpa-cx9`	`__NV_DPA_CX9`

Because ConnectX-7 and BlueField-3 share the same DPA hardware, the compiler treats `nv-dpa-cx7` as an alias of `nv-dpa-bf3`.

7.4.3.4.3.1 Link Compatibility

Only link-compatible relocatable objects can be linked together; attempting to link incompatible objects will result in an error. Additionally, the toolchain version of the linker must match the toolchain version of the compiler used to produce the objects.

If architecture B is newer than A and they are link-compatible, objects built for target A can be linked to build an application for target B. However, the inverse (i.e., linking objects built for target B into an application for target A) is not valid.

For example, BlueField-3/ConnectX-7 and ConnectX-8 are link-compatible. This means objects built for BlueField-3/ConnectX-7 can be linked together to build an application for ConnectX-8.

Object Target ↓ \ Executable Target →	nv-dpa-bf3 / nv-dpa-cx7	nv-dpa-cx8	nv-dpa-cx9
nv-dpa-bf3 / nv-dpa-cx7	Compatible	Compatible	Compatible
nv-dpa-cx8	Incompatible	Compatible	Compatible
nv-dpa-cx9	Incompatible	Incompatible	Compatible

7.4.3.4.4 Architecture Macros

The compiler defines identifier macros for each version of DPA hardware, as listed in the DPA Hardware Architectures table. Each identifier macro has a unique integer value strictly greater than those of older DPA CPU models. Known aliases, such as BlueField-3 DPA and ConnectX-7 DPA, share the same integer value.

The macro `__NV_DPA` is defined as the value of the current compilation target. This can be used to write device code specific to a DPA hardware generation:

```
#if __NV_DPA == __NV_DPA_BF3
// Code for BlueField-3 here
#elif __NV_DPA > __NV_DPA_BF3
// Code for devices after BlueField-3 here
#endif
```

⚠ The ordering established by the hardware version macros does not imply an ordering of features supported by the hardware. It is the user's responsibility to ensure that features used in code specific to a DPA version are actually supported by that hardware.

7.4.3.4.5 LTO Usage Guidelines

7.4.3.4.5.1 Restrictions

- Only the default linker script is supported with LTO
- Using options `-fPIC` / `-fpic` / `-shared` / `-mcmodel=large` through `-devicecc-options` is not supported when LTO is enabled
- Fat bitcode objects containing both LLVM bitcode and ELF representation are not supported
- Thin LTO is not supported

7.4.3.4.5.2 Compatibility

During compilation, LLVM generates the object as bitcode intermediate representation (IR) when LTO is enabled instead of ELF representation. The bitcode IR generated by the DPA compiler is only guaranteed to be compatible within the same version. The toolchain version of the compiler which builds the objects involved in link-time optimization (enabled with `-flto`) and the toolchain version of the linker which performs LTO must be the same.

7.4.3.4.6 Optimized memcpy

7.4.3.4.6.1 `__dpa_memcpy` Builtin

DPACC supports the `__dpa_memcpy` builtin, an inline implementation of the standard `memcpy` operation. It is designed to minimize memory operations when the copy size is known at compile time. `__dpa_memcpy` accounts for the alignment of both source and destination base pointers, producing optimized code even when the source and destination have mismatched alignments. The compiler now automatically promotes standard `memcpy` calls to `__dpa_memcpy` whenever it yields better performance.

The builtin signature is the same as the standard library `memcpy` function.

Usage Notes

To achieve optimal code generation, developers should provide alignment information for the source and destination buffers using one of the following methods:

- `aligned` attribute: Use when declaring buffers or variables.
- `assume_aligned` attribute: Use if a function returns an aligned buffer.
- `__builtin_assume_aligned()` builtin: Use if the underlying buffer of a pointer is known to be aligned.

Usage Example

The following example demonstrates using **__builtin_assume_aligned()** to inform the compiler of 8-byte alignment, even when accessing offsets:

```
#define LENGTH 24

void aligned_parameters(uint8_t* dst, uint8_t* src) {
    // Inform compiler that dst and src base pointers are 8-byte aligned
    dst = __builtin_assume_aligned(dst, 8);
    src = __builtin_assume_aligned(src, 8);

    // Copy with offsets (even if offsets result in mismatched alignment)
    __dpa_memcpy(dst + 50, src + 60, LENGTH);
}
```

7.4.3.4.7 Deprecated Features

- The **-ldpa** option which links with DOCA-DPA libraries is deprecated and will be removed in future releases. Use the **-ldoca_dpa** option instead.

7.4.3.4.8 Examples

This section provides some common use cases of DPACC and showcases the **dpacc** command.

7.4.3.4.8.1 Building Libraries

This example shows how to build DPA libraries using DPACC. Libraries for DPA typically contain two archives, one for the host and one for the device.

```
dpacc input.c -hostcc=gcc -mcpu=nv-dpa-bf3 -o lib<name> -gen-libs -hostcc-options="-fPIC"
```

This command generates the output files **lib<name>_host.a** and **lib<name>_device.a**.

The host stub archive can be linked with other host code to generate a shared/static host library.

- Generating a static host library:

```
ar x lib<name>_host.a                  # Extract objects to generate *.o
ar cr lib<name>.a <*src.host.o> *.o    # Generate final static archive with all objects
```

- Generating a shared host library:

```
gcc -shared -o lib<name>.so <*src.host.o> -Wl,-whole-archive -l<name>_host -Wl,-no-whole-archive       #
Link the generated archive to build a shared library
```

7.4.3.4.8.2 Linking with DPA Device Library

The DPA device library generated by DPACC using **-gen-libs** as part of a DPA library can be consumed by DPACC using the **-device-libs** option. All such libraries are only linked statically.

```
dpacc input.c -hostcc=gcc -mcpu=nv-dpa-bf3 -o libInput.a -device-libs="-L <path-to-library> -l<libName>"
```

7.4.3.4.8.3 Enabling Link-time Optimizations

Link-time optimizations can be enabled using **-flto** along with an optimization level specified for device compilation.

```
dpacc input1.c -hostcc=gcc -mcpu=nv-dpa-bf3 -c -flto -devicecc-options="-O2"
dpacc input2.c -hostcc=gcc -mcpu=nv-dpa-bf3 -c -flto -devicecc-options="-O2"
dpacc -mcpu=nv-dpa-bf3 input1.dpa.o input2.dpa.o -hostcc=gcc -o libInput.a
```

7.4.3.4.8.4 Including Headers

This example includes headers for device compilation using **devicecc-options** and host compilation using **hostcc-options**. You may also specify headers for any compilation on both the host and device side using the **-I** option.

```
dpacc input.c -hostcc=gcc -mcpu=nv-dpa-bf3 -o libInput.a -I <common-headers-path> -devicecc-options="-I <device-
headers-path>" -hostcc-options="-I <host-headers-path>"
```

7.4.3.4.8.5 Providing Non-default Security Attributes Blob

By default, the **dpacc** compiler embeds a security attributes blob provided by FlexIO. To override this behavior, **dpacc** provides the **--dpa-proc-attr** option.

This flag allows you to specify the path to a custom, user-provided attributes blob. When used, this blob will be embedded into the compiled application instead of the default one.

For detailed instructions on creating the attributes file and converting it into a blob, please refer to the "DPA Application Attributes to Blob" documentation.

The following command demonstrates the use of this flag to specify a custom attributes blob:

```
dpacc dev_code.c -hostcc=gcc -mcpu=nv-dpa-bf3 -o libapp.a --dpa-proc-attr /path/to/attributes/blob
```

7.4.3.4.8.6 Linking FlexIO Libraries Dynamically

DPACC provides options that control how the **libflexio-al** and **libflexio-al-cp** libraries are linked during device linking.

Default behavior:

- **libflexio-al** : Linked statically
- **libflexio-al-cp** : Linked dynamically

This behavior can be controlled using the **--link-flexio-al** and **--link-flexio-al-cp** options.

To override the default and link the **libflexio-al-cp** library statically, use the following:

```
dpacc dev_code.c -hostcc=gcc -mcpu=nv-dpa-bf3 -o libapp.a --link-flexio-al-cp=static
```

7.4.3.4.8.7 Dump Targets Supported by a Fatbinary File

The targets supported by a fatbinary file can be dumped using **dpa-fatbin**. This tool can also dump the targets supported by a DPA library from the device archive.

```
dpa-fatbin --list libfoo_device.a
dpa-fatbin --list device_exec.fatbin
```

7.4.3.4.8.8 Dump Target of Device ELF file

The target for which a device ELF file is built can be dumped using **dpa-objdump**.

```
dpa-objdump --file-headers foo.o
```

7.4.3.4.8.9 Generating Output as Source Code

DPACC provides the option **-src-output** to generate output as host-source code. This source can be compiled by the host compiler to generate functionally equivalent output which DPACC would have generated directly.

This example shows how to build various outputs of DPACC as source using this option and how to compile the generated source.

DPA Program Source

Generate DPA program source to DPACC:

```
dpacc input.c -hostcc=gcc -mcpu=nv-dpa-bf3 -o libfoo.c -src-output
```

Compile the generated source using the host compiler to generate an object and build an archive with this object. The macro **__DPACC_SRC_TARGET__** must be defined when building this object to remove unnecessary code when building from source.

```
$ gcc libfoo.c -c -I /opt/mellanox/flexio/include -Wno-attributes -Wno-pedantic -Wno-unused-parameter -Wno-return-
type -Wno-implicit-function-declaration -D__DPACC_SRC_TARGET__
$ ar cr libfoo.a libfoo.o
```

DPA Library Source

Generate DPA library source passing this option to DPACC

```
dpacc input.c -hostcc=gcc -mcpu=nv-dpa-bf3 -o libfoo -gen-libs -src-output
```

This generates the device archive **libfoo_device.a** and host code files **libfoo.lib.c**, **input.dpa.c**. The host archive of DPA library is generated by compiling these sources and building an archive. The **__DPACC_SRC_TARGET__** macro needs to be defined in this case to remove unnecessary code.

```
$ gcc libfoo.lib.c input.dpa.c -c -I /opt/mellanox/flexio/include -Wno-attributes -Wno-pedantic -Wno-unused-
parameter -Wno-return-type -Wno-implicit-function-declaration -D__DPACC_SRC_TARGET__
$ ar cr libfoo_host.a libfoo.lib.o input.dpa.o
```

DPA Object Source

Generate DPA object source passing this option to DPACC:

```
dpacc input.c -hostcc=gcc -mcpu=nv-dpa-bf3 -c -src-output
```

This generates a single file, **input.dpa.c**. Compile the host file to generate an object.

```
gcc input.dpa.c -c -I /opt/mellanox/flexio/include -Wno-attributes -Wno-pedantic -Wno-unused-parameter -Wno-return-type -Wno-implicit-function-declaration
```

7.4.3.4.9 DPA Compiler Usage

The DPA-Compiler is an LLVM-based compiler used internally by DPACC to compile and link device code.

7.4.3.4.9.1 Passing Options

User-specified options can be passed to the underlying compiler and linker using the following DPACC flags:

- **--devicecc-options**: Passes options to the compiler.
- **--devicelink-options**: Passes options to the linker.

7.4.3.4.9.2 Inlining Control

The compiler supports the **-finline-level=<n>** option to control inlining aggressiveness, where **<n>** is an integer between 0 and 100. This option is supported even during Link Time Optimization (LTO).

- Value 100 – Maximizes inlining. All "viable" functions (e.g., those without **noinline** attributes) are inlined.
- Value 0 – Minimizes inlining. The compiler inlines only functions marked with the **always_inline** attribute or those where inlining is unconditionally beneficial under the current cost model.

7.4.3.4.9.3 Additional Resources

Refer to the following resources for additional options:

- Clang Compiler User Manual
- Clang Command Line Argument Reference
- Target-dependent Compilation Options
- LLD Command Line Reference

⚠ Important Notes

- Invoking the compiler, assembler, or linker directly (bypassing DPACC) may lead to unexpected errors
- Linker options are processed through the compiler driver (**dpa-clang**).

- Unlike GNU `ld` (which replaces the configuration), the LLD linker script is additive (i.e., it is honored in addition to the default configuration). You may need specific options to override default behaviors.
- Enabling optional standard library extensions by defining the `__STDC_WANT_LIB_EXT1__` macro is not supported.

7.4.3.4.9.4 dpacc-extract Command Line Options

`dpacc-extract` extracts a device executable out of a DPA program or a host executable containing DPA program(s).

To execute `dpacc-extract`:

```
Usage  : dpacc-extract <input-file> -o=<output-file> [other options]
Options
=======

--app-name                (-app-name)
        Specify name of the DPA application to extract.
        This option is mandatory if more than one application is present in the input file.

--dpa-proc-attr           (-pa)
        Extract DPA process attributes blob.

--help                    (-h)
        Print this help information on this tool.

--info                    (-i)
        List apps and supported targets.

--mcpu                    (-mcpu)
        Specify name of the device for which the application is to be extracted.
        This option is mandatory if the application is built for multiple devices.

--options-file <file>,...  (-optf)
        Include command line options from specified file.

--output-file             (-o)
        Specify name of the output file.

--version                 (-V)
        Print version information on this tool.
```

Mandatory arguments:

Flag	Description
Input file	DPA program or host executable containing DPA program. Specifying one input file is mandatory.
`-o`, `--output-file <file>`	Specify name and location of the output device executable.
`-app-name`, `--app-name <name>`	Specify name of the DPA application to extract. Mandatory if input file has multiple DPA apps.
`-mcpu`, `--mcpu <target>`	Specify name of the device for which the application is to be extracted. Mandatory if there are multiple target variants for an app.

7.4.3.4.9.5 Objdump Command Line Options

The `dpa-objdump` utility prints the contents of object files and final linked images named on the command line.

For more information, refer to the Objdump command line reference.

7.4.3.4.9.6 Archiver Command Line Options

`dpa-ar` is a Unix ar-compatible archiver.

For more information, refer to the Archiver command line reference.

7.4.3.4.9.7 NM Tool Command Line Options

The `dpa-nm` utility lists the names of symbols from object files and archives.

For more information, refer to the NM tool command line reference.

7.4.3.4.9.8 Miscellaneous Notes

- Objects produced by LLD are not compatible with those generated by any other linker.
- Ensure that there is at least one reference to the device entry point function in the DPA program from the host application so that during linking, the DPA program is not silently discarded by the host linker because it is not referenced.

7.4.3.5 Release Notes

7.4.3.5.1 Changes in 2.3.0 Release

Title	Description
Bug fixes	Fix __dpa_remote_atomic_load getting hoisted outside loops Fix CX8 macro typo in dpaintrin.h header

7.4.4 Single Point of Resource Distribution

This document describes the Single Point of Resource Distribution (SPRD) system, including the `dpa-resource-mgmt` tool, new Flex IO SDK APIs for consuming SPRD outputs, and the DPA Execution Unit (EU) management tool, `dpaeumgmt`.

> ⚠ Execution Unit partitioning is planned for support in a future release.

7.4.4.1 Introduction

DPA Flex IO applications may run simultaneously across multiple VHCAs, with multiple applications per VHCA. This can lead to contention for shared resources—such as Execution Units (EUs). Historically, applications independently allocated EUs based on general recommendations, which could result in performance degradation or even application crashes due to over-commitment.

To address this, the SPRD system provides a centralized infrastructure to orchestrate DPA resource allocation across applications. It is designed primarily for administrators managing DPUs in scalable environments and is managed using the `dpa-resource-mgmt` tool.

SPRD enables consistent resource planning at scale, for example, deploying the same configuration across a cluster of DPUs.

7.4.4.1.1 Key Characteristics

- Centralized EU resource management via `dpa-resource-mgmt`.
- Supports BlueField-3 and newer platforms.
- Operates over the `dpaeumgmt` EU management infrastructure.
- Host applications can reside on:
  - The BlueField Arm core (BF-ARM)
  - One or more external hosts (e.g., multi-host server configurations)
- Supports multiple instances of the same application running across different hosts.
- Non-SPRD-aware applications continue to run unaffected.

7.4.4.1.2 Integration Requirements

To fully adopt SPRD, the following updates are required:

- New APIs must be added to the FlexIO SDK.
- Existing tests and applications that use EU resources must be updated to consume allocations provided via the SPRD tool and APIs.

7.4.4.1.3 Glossary

This table introduces important terms necessary to understand this document:

Term	Definition
DPA	Data-path accelerator; an auxiliary processor designed to accelerate data-path operations.
DPA partition manager	PCIe device function capable of controlling the system's EUs. On NVIDIA® BlueField®-3, this is the ECPF. It is associated with the default partition. The application is named `dpaeumgmt`.
DOCA	SDK for DPU.
EU (Execution Unit)	Hardware execution unit; a logical DPA processing unit.
EU group	A collection/subset of EUs created using `dpaeumgmt`. EU groups belong to an EU partition and can only consist of EUs from that partition.
EU object	Refers to either an EU partition or an EU group.
EU partition	An isolated pool of EUs created using `dpaeumgmt`. A vHCA must be associated with a partition to use DPA resources and execute software threads.
EU affinity	Method for pairing a DPA thread with a DPA EU. Supported types: • none – use any available EU • strict – use a specific EU by ID • group – use an EU from a specified group

Term	Definition
MFT	NVIDIA Firmware Tools.
SPRD	Single Point of Resource Distribution (`dpa-resource-mgmt`).
vHCA	Virtual Host Channel Adapter; a virtualized representation of a physical network interface.

7.4.4.2 Preparation and Description

To work with SPRD, ensure the following prerequisites are met:

7.4.4.2.1 System Requirements

To work with SPRD, the following requirements must be met:

- A functioning system with DOCA installed
- Python 3.6 or higher
- An NVIDIA® BlueField®-3 or newer device
- MFT package installed and running, version 4.32.0-94 or higher
- Firmware (FW) version XX.45.0322 or higher
- `dpa-resource-mgmt` package installed and running, version 25.04.160 or higher

7.4.4.2.2 EUs and Cores

The hardware consists of several cores, each containing a set number of EUs. Some EUs or entire cores may be reserved by the firmware and are therefore unavailable to both `dpa-resource-mgmt` and user applications.

7.4.4.2.3 Partition Concept

A partition is a logical, continuous array of EUs starting at index zero, with a fixed size. Partitions are assigned to one or more vHCAs, and only applications on those vHCAs can use the EUs in the assigned partition.

Partition types:

- Default partition (ROOT):
  - Created at boot on root devices (ECPF in DPU mode, PF in NIC mode)
  - Operates over all free EUs initially
  - Uses absolute EU indices
- Sub-partitions:
  - Created and deleted at runtime
  - Pull EUs from the default partition (which then loses access to them)
  - EUs in a sub-partition do not need to be contiguous, but must not overlap with other sub-partitions
  - Use relative EU indices starting from 0

When a sub-partition is deleted, its EUs are returned to the default partition.

7.4.4.2.4 Group Concept

A group is a firmware-managed list of EUs. When EU affinity is set to "group", the firmware selects an EU from the assigned group.

- The maximum number of groups per sub-partition is defined when the partition is created
- The default partition (ROOT) has its limit defined by the **max_num_dpa_groups** capability (currently 15)
- Every partition also has an implicit hidden group, used when EU affinity is set to **none**

The total number of groups allowed on a device must be ≤ **max_num_dpa_groups** (15), and is calculated as:

We don't have a way to export this macro.

Where:

- P – number of sub-partitions (excluding the default)
- RG – number of groups in the default (ROOT) partition
- G_i – number of groups in each sub-partition **i**

> Example limitation: Even if no groups are defined in any partition, the number of sub-partitions cannot exceed 15, due to the group tracking mechanism.

7.4.4.3 SPRD

7.4.4.3.1 dpa-resource-mgmt Utility

The primary tool provided by SPRD is **dpa-resource-mgmt**, a Python 3-based CLI utility used by system administrators to configure and manage DPA resource allocations. It interfaces with the lower-level **dpaeumgmt** utility to perform hardware-level execution unit (EU) operations.

This utility takes as input:

- The InfiniBand (ibv) device name
- A YAML configuration file describing partition and application layout
- An optional output path where generated application configuration files will be saved (defaults to current working directory)

Each output file corresponds to a defined partition and contains the required configuration for applications that will run within that partition.

7.4.4.3.1.1 Operating Modes

config – Configure Partitions and Generate Application Files

Creates partitions and EU groups as defined in the YAML file, and generates corresponding application configuration files.

Options:

```
-d, --device DEVICE        ibv device name (required)
-f, --filename FILENAME    YAML configuration file (required)
-o, --output OUTPUT        Output directory for generated files (optional; default: current path)
-v, --verbose              Print underlying dpaeumgmt commands (optional)
```

query – Query Resource Status

Displays resource-related information such as chip-level capabilities, vHCA data, or DPA resource allocation. Output can be directed to a file and formatted in YAML or human-readable format.

Options:

```
-d, --device DEVICE        ibv device name (required)
-t, --type {chip,vhca,resources}
                           Type of information to display (required)
-f, --filename FILENAME    Output file (optional; default: stdout)
-y, --yaml-format          Output in YAML format (optional; default: human-readable)
-v, --verbose              Show executed commands (optional)
```

clear – Clean Up Existing Configuration

Destroys all partitions and groups associated with the given device.

Options:

```
-d, --device DEVICE        ibv device name (required)
-v, --verbose              Show executed commands (optional)
```

version – Print Tool Version

Displays the currently installed version of **dpa-resource-mgmt**.

7.4.4.3.1.2 Back-end

The **dpa-resource-mgmt** utility operates by interfacing with the **dpaeumgmt** tool to interact directly with the hardware. It performs the following key functions:

- System discovery – Retrieves detailed hardware and topology information from the chip, including:
  - Number of cores
  - Number of available EUs
  - vHCA identifiers and their parent-child relationships
- Configuration parsing and validation:
  - Reads the YAML configuration file provided via the **--filename** option
  - Validates the structure and content of the file
  - If any issue is detected (e.g., syntax error, invalid partition configuration), the utility exits with a descriptive error message
- Pre-configuration checks:
  - Checks for existing active partitions, running processes, and configured EU groups
  - If such elements are found, the utility prompts the administrator to remove them manually, then exits without making changes
- Partition setup – If no conflicts are found, the utility proceeds to:
  - Create EU partitions
  - Configure EU groups

- Generate partition-specific application configuration files, which are saved in the output directory (one file per partition)

This structured backend workflow ensures consistent and safe allocation of DPA resources across different applications and hosts.

7.4.4.3.1.3 Examples of Running dpa-resource-mgmt

Query VHCA Information

```
$ dpa-resource-mgmt query -t vhca -d mlx5_0
    VHCA_ID = 0x1
    FUNCTION_TYPE = ECPF
    PCI_BDF = 80:00.0
    VPORT_ID = 0x0

            VHCA_ID = 0x2
            FUNCTION_TYPE = PF
            PCI_BDF = 80:00.1
            VPORT_ID = 0x0

            VHCA_ID = 0x3
            FUNCTION_TYPE = PF
            PCI_BDF = 80:00.2
            VPORT_ID = 0x0

            VHCA_ID = 0x4
            FUNCTION_TYPE = PF
            PCI_BDF = 80:00.3
            VPORT_ID = 0x0

...
```

Query DPA Chip Information

```
$ dpa-resource-mgmt query -t chip -d mlx5_0
Maximal number of DPA EU per group (max_num_dpa_eu_per_group): 190
Maximal number of active DPA EU (max_num_dpa_eu): 190
Maximal number of DPA EU partitions (max_num_dpa_eu_partitions): 15
Maximal number of VHCAs associated with a single partition (max_num_partition_vhca_id): 16
Maximal number of EU groups could be created (max_num_dpa_groups): 15
Heap maximal size (heap_max_size): 1073741824
Maximal number of EU groups by FirmWare (max_eu_groups): 16
Maximal number of processes per VHCA (max_process_per_vhca): 64
Maximal number of DPA processes (max_processes): 64
Maximal number of DPA threads (max_threads): 40000
Maximal number of DPA threads per process (max_threads_per_process): 20000
Number of cores (num_cores): 12
DPA thread stack size (thread_stack_size): 16384
Cores:
        Core ID 0
        EUs number 16
        EUs IDs 0-15
        EUs indexes 0-15

        Core ID 1
        EUs number 16
        EUs IDs 16-31
        EUs indexes 0-15

        Core ID 2
        EUs number 16
        EUs IDs 32-47
        EUs indexes 0-15

...
```

Query DPA Resource Information

```
$ dpa-resource-mgmt query -t resources -d mlx5_0
Free EUs in ROOT partition 0-3,9-15,30,31,40-47,67-79,112-159,169
Cores available for use in ROOT partition 0-2,4,7-9,14
Free EUs in ROOT partition with groups 0-3,9-15,30,31,40-47,67-79,112-159,162-189
Cores available for use ROOT partition with groups 0-2,4,7-9,14,15

    PARTITIONS:

        PARTITION_1:
        partition_id: 1
        vhca_id: 5
        Range of EUs: 4-8,16-29,32-39
        Cores in use: 0-2
        GROUPS:
```

```
            name: EU1
            group_id: 1
            abs_EUs: 16,17,20-22,32
            Relative EUs: 5,6,9-11,19
            Cores in use: 1,2

            name: EU1_1
            group_id: 2
            abs_EUs: 26-28
            Relative EUs: 26-28
            Cores in use: 1

        PARTITION_2:
        partition_id: 2
        vhca_id: 2
        Range of EUs: 48-66
        Cores in use: 3,4
        GROUPS:

            name: EU2
            group_id: 1
            abs_EUs: 48-51,64
            Relative EUs: 0-3,16
            Cores in use: 3,4

        PARTITION_3:
        partition_id: 3
        vhca_id: 3,4,9-13
        Range of EUs: 80-111,160,161
        Cores in use: 5,6,14
        GROUPS:

            name: EU3
            group_id: 1
            abs_EUs: 80-96
            Relative EUs: 0-16
            Cores in use: 5,6

    ROOT_PARTITION_GROUPS:

        name: RG0
        group_id: 1
        abs_EUs: 170-189
        Cores in use: 14,15

        name: RG1
        group_id: 2
        abs_EUs: 162-168
        Cores in use: 14

Core's information:
    Core #0, with 16 EUs, has 11 EUs available: 0-3,9-15.
    Core #1, with 16 EUs, has 2 EUs available: 30,31.
    Core #2, with 16 EUs, has 8 EUs available: 40-47.
    Core #3, with 16 EUs, is fully utilized.
    Core #4, with 16 EUs, has 13 EUs available: 67-79.
    Core #5, with 16 EUs, is fully utilized.
    Core #6, with 16 EUs, is fully utilized.
    Core #7, with 16 EUs, is fully available.
    Core #8, with 16 EUs, is fully available.
    Core #9, with 16 EUs, is fully available.
    Core #14, with 16 EUs, has one EU available: 169.
    Core #15, with 14 EUs, is fully utilized.

VHCA information:
    10 unutilized VHCAs: 6-8,14-20
```

Clear Previously Allocated Configuration

```
$ dpa-resource-mgmt clear -d mlx5_0
Successfully destroyed all partitions and groups.
```

Config

```
$ dpa-resource-mgmt config -f config.yaml -d mlx5_0 -v
sudo /opt/mellanox/doca/tools/dpaeumgmt version
sudo /opt/mellanox/doca/tools/dpaeumgmt info status --dpa_device mlx5_0
sudo /opt/mellanox/doca/tools/dpaeumgmt info vhca --dpa_device mlx5_0
sudo /opt/mellanox/doca/tools/dpaeumgmt info chip --dpa_device mlx5_0
sudo /opt/mellanox/doca/tools/dpaeumgmt partition info --dpa_device mlx5_0
sudo /opt/mellanox/doca/tools/dpaeumgmt eu_group info --dpa_device mlx5_0
sudo /opt/mellanox/doca/tools/dpaeumgmt partition create --dpa_device mlx5_0 --vhca_list 5 --range_eus
4-8,16-29,32-39 --max_num_eu_group 2
sudo /opt/mellanox/doca/tools/dpaeumgmt partition create --dpa_device mlx5_0 --vhca_list 2 --range_eus 48-66 --
max_num_eu_group 1
sudo /opt/mellanox/doca/tools/dpaeumgmt partition create --dpa_device mlx5_0 --vhca_list 3,4,9-13 --range_eus
80-111,160,161 --max_num_eu_group 1
sudo /opt/mellanox/doca/tools/dpaeumgmt eu_group create --dpa_device mlx5_0 --range 5,6,9-11,19 --id_partition 1 --
name_group EU1
sudo /opt/mellanox/doca/tools/dpaeumgmt eu_group create --dpa_device mlx5_0 --range 15-17 --id_partition 1 --
name_group EU1_1
sudo /opt/mellanox/doca/tools/dpaeumgmt eu_group create --dpa_device mlx5_0 --range 0-3,16 --id_partition 2 --
name_group EU2
```

```
sudo /opt/mellanox/doca/tools/dpaeumgmt eu_group create --dpa_device mlx5_0 --range 0-16 --id_partition 3 --
name_group EU3
sudo /opt/mellanox/doca/tools/dpaeumgmt eu_group create --dpa_device mlx5_0 --range 170-188 --name_group RG0
sudo /opt/mellanox/doca/tools/dpaeumgmt eu_group create --dpa_device mlx5_0 --range 162-168 --name_group RG1
```

7.4.4.3.2 Format of Input YAML File

The input YAML file is divided into two main parts:

1. Partition definitions
2. Application definitions (not covered in this document)

These parts are separated by the standard YAML `---` divider.

7.4.4.3.2.1 Partition Definitions

This section defines how execution units (EUs) are assigned to partitions and groups.

Example:

```
version: 25.04

partitions:
  PARTITION_1:
    vhca_id: 1
    EU_allocation:
      - abs_EUs: 4-8
      - core: 1
        num_EUs: 14
      - core: 2
        num_EUs: 8
    groups:
      - name: EU1
        EUs:
          - abs_EUs: 16,17,20-22
          - core: 2
            num_EUs: 1
      - name: EU1_1
        EUs:
          - abs_EUs: 26-28

  PARTITION_2:
    vhca_id: 2
    EU_allocation:
      - core: 3
        num_EUs: 16
      - core: 4
        num_EUs: 3
    groups:
      - name: EU2
        EUs:
          - core: 3
            num_EUs: 4
          - core: 4
            num_EUs: 1

  PARTITION_3:
    vhca_id: [3, 4, 9, 10-13]
    EU_allocation:
      - core: 5
        num_EUs: 16
      - core: 6
        num_EUs: 16
    groups:
      - name: EU3
        EUs:
          - core: 5
            num_EUs: 16
          - core: 6
            num_EUs: 1

root_partition_groups:
  - name: RG0
    EUs:
      - abs_EUs: 170-188
  - name: RG1
    EUs:
      - core: 10
        num_EUs: 7
```

7.4.4.3.2.2 Partition Configuration Schema (Input YAML Format)

```
version: "string"  # Version identifier (e.g., "25.04")
# REQUIRED: version must be defined

partitions:  # Defines different partitions such as ARM, HOSTx86, etc.
  # Can be an empty dictionary: {}
  "<partition_name>":  # Dynamic key, replace with actual partition names
    vhca_id:  # Virtual Host Channel Adapter (VHCA) ID
      one_of:
        - type: "integer"  # Single VHCA ID (e.g., 1, 2)
        - type: "array"    # Multiple VHCA IDs (e.g., [3, 4, 9, 10-13])
          items: "integer"

    EU_allocation:  # List of Execution Unit (EU) allocations
      # REQUIRED: EU_allocation must be defined
      - abs_EUs: "string"  # Absolute EU range, e.g., "4-8" or "16,17,20-22"
      - core: "integer"    # Core ID to which EUs are allocated
        num_EUs: "integer"  # Number of EUs allocated

    groups:  # Defines groups of EUs within each partition
      - name: "string"  # Name of the EU group (e.g., "EU1", "EU2")
        EUs:  # List of Execution Units assigned to the group. Should be either abs_EUs or core with num_EUs.
          # REQUIRED: EUs must be defined
          - abs_EUs: "string"  # Absolute EU assignments (e.g., "26-28")
          - core: "integer"    # Core ID for EU allocation
            num_EUs: "integer"  # Number of EUs assigned to the core

root_partition_groups:  # Defines global root partition groups
  - name: "string"  # Name of the root partition group (e.g., "RG0")
    # REQUIRED: name must be defined
    EUs:  # List of Execution Units assigned to the group. Should be either abs_EUs or core with num_EUs.
      # REQUIRED: EUs must be defined
      - abs_EUs: "string"  # Absolute EU assignments (e.g., "170-188")
      - core: "integer"    # Core ID for EU allocation
        num_EUs: "integer"  # Number of EUs assigned to the core
```

7.4.4.3.2.3 Schema Description

Field	Description
`version`	Required. Specifies the configuration version (e.g., `"25.04"`).
`partitions`	Defines custom partitions and their associated resources. Can be empty (`{}`).
`partitions.<name>.vhca_id`	Required. Specifies the VHCA ID(s) to associate with the partition. Accepts a single number, list, or range. Example: `2`, `[3, 4]`, or `10-13`.
`partitions.<name>.EU_allocation`	Required. Defines EUs to assign to the partition. Supports: • `abs_EUs` – Comma-separated numbers or ranges (e.g., `7-9,11`) • `core` / `num_EUs` – Allocates a number of EUs from a specified core
`partitions.<name>.groups`	Defines named EU groups within a partition. Each group must include: • `name` – Group name (e.g., `EU1`) • `EUs` – Similar format to `EU_allocation`
`root_partition_groups`	(Optional) Defines EU groups within the ROOT partition. Uses the same format as `groups`. Each must have a unique `name` and valid `EUs`.

7.4.4.3.2.4 Notes on Field Behavior

`abs_EUs` vs. `core` / `num_EUs`

- Use `abs_EUs` for precise EU selection.
- Use `core` + `num_EUs` for abstracted allocation based on availability.
- Cannot combine both methods within a single mapping item.

Default Partition Behavior

- EUs not explicitly assigned to custom partitions remain in the default ROOT partition.
- The default partition is auto-created on boot and operates over unallocated EUs.

Partition Constraints

- Sub-partitions may not overlap in EU assignment.
- On sub-partition deletion, its EUs are returned to the ROOT partition.
- Each partition has its own relative EU indexing starting at 0.

Group Constraints

- The number of allowed groups is limited by the `max_num_dpa_groups` capability (currently 15).
- Hidden default groups are internally used for affinity mode `none`.

7.4.4.3.2.5 Application Definitions (YAML: DPA_APPS)

This section describes how applications are mapped to partitions and which execution units (EUs) or groups they use. EU values are defined in absolute terms for clarity but are converted to relative indices in output files.

Example:

```
DPA_APPS:
  APP1:
    - partition: PARTITION_1
      affinity_EUs: [35, 37-39]
      groups: [EU1, EU1_1]
      affinity_core:
        - core: 1
          num_EUs: 4
    - partition: PARTITION_2
      affinity_EUs: [61, 62]
      groups: EU2
    - partition: ROOT
      groups: RG0
      affinity_EUs: [190, 191]
      affinity_core:
        - core: 0
          num_EUs: 2
  APP2:
    - partition: PARTITION_1
      affinity_EUs: [63]
      groups: EU2
  APP3:
    partition: PARTITION_3
    affinity_EUs: [97, 99]
  APP4:
    partition: PARTITION_3
    affinity_EUs: 98
    groups: EU3
```

7.4.4.3.2.6 Application-to-Partition Mapping Schema (Input YAML Format)

```
DPA_APPS:  # List of applications with their partition mappings
  "<app_name>":  # Dynamic key, replace with actual application names (e.g., APP1, APP2)
    one_of:
      - type: array  # If an application has multiple partitions
        items:
          partition: "string"  # Name of the partition (e.g., "PARTITION_1", "PARTITION_2", "ROOT")
          # REQUIRED: partition must be defined

          affinity_cores:  # List of core/num_EUs pairs assigned to this app
            - core: "integer"
              num_EUs: "integer"

          affinity_EUs:  # List of Execution Units (EUs) assigned to this app
            one_of:
              - type: "array"  # List of EUs (e.g., [35, 37-39])
                items: integer
              - type: "integer"  # Single EU value (e.g., 98)
```

```
            - type: "string"  # Absolute EU range, e.g., "4-8" or "16,17,20-22"

        groups:  # Groups assigned to this app
          one_of:
            - type: array  # Multiple groups (e.g., ["EU1", "EU1_1"])
              items: string
            - type: string  # Single group (e.g., "EU2")

    - type: object  # If an application has only one partition
      properties:
        partition: "string"
        affinity_cores:
          - core: "integer"
            num_EUs: "integer"
        affinity_EUs:
          one_of:
            - type: array
              items: integer
            - type: integer
        groups:
          one_of:
            - type: array
              items: string
            - type: string
```

7.4.4.3.2.7 Schema Overview

Field	Description
`DPA_APPS`	Root node for defining applications
`<app_name>`	Application name (e.g., `APP1`, `APP2`)
`partition`	Partition assigned to the application (`PARTITION_1`, `PARTITION_2`, or `ROOT`)
`affinity_EUs`	Absolute EU values assigned to the app. Accepts: single int, list, ranges (e.g., `[35, 37-39]`), string (e.g. `35, 37-39`)
`affinity_core`	Alternative to `affinity_EUs`; assign EUs by number per core (e.g., `core: 1`, `num_EUs: 4`)
`groups`	Named EU groups the app uses. Accepts: single string or list (e.g., `["EU1", "EU2"]`)

7.4.4.3.2.8 Notes

`affinity_EUs` vs `affinity_cores`

Method	Description
`affinity_EUs`	Offers precise control using explicit EU numbers or ranges.
`affinity_cores`	Uses a core ID and number of EUs to assign. Abstracts allocation and balances load.

Using `affinity_cores` is recommended when the partition itself uses `core/num_EUs` style allocation.

Mixing `affinity_EUs` with Groups

Although supported, it's not recommended to mix strict `affinity_EUs` with group-based allocation (`groups`). Doing so may introduce conflicts during EU assignment but is not prohibited.

Using the `ROOT` Partition

Apps may be assigned to the `ROOT` partition. In this case:

- You must reference one of the `root_partition_groups`.
- You may use `affinity_EUs` or `affinity_core` — but only with EUs/cores not already allocated by sub-partitions.

7.4.4.3.2.9 Known Limitation

`dpa-resource-mgmt` does not validate whether a `vhca_id` belongs to a root PF in NIC mode or a VF. Errors will only occur during backend validation (via `dpaeumgmt`).

7.4.4.3.3 Output YAML files

After a successful run of `dpa-resource-mgmt`, the tool generates a YAML file for each partition defined in the input file. Each output file is named after its corresponding partition (e.g., `PARTITION_1.yaml`) and contains the list of applications assigned to that partition, along with their resolved EU and group mappings.

7.4.4.3.3.1 Example Output YAML file

```
version: 25.04
DPA_APPS:
  - name: APP1
    number_of_groups: 2
    # EU1, EU1_1
    groups: [1, 2]
    # absolute EUs: [16, 17, 18, 19, 20, 21, 32, 33, 35, 37, 38, 39]
    number_of_affinity_EUs: 12
    affinity_EUs: [5, 6, 7, 8, 9, 10, 19, 20, 22, 24, 25, 26]

  - name: APP2
    # absolute EUs: [16, 17, 22, 23, 24, 25, 26, 27, 34, 36]
    number_of_affinity_EUs: 10
    affinity_EUs: [5, 6, 11, 12, 13, 14, 15, 16, 21, 23]

  - name: APP3
    # absolute EUs: [16, 18, 19, 20, 21, 28, 29, 32, 33]
    number_of_affinity_EUs: 9
    affinity_EUs: [5, 7, 8, 9, 10, 17, 18, 19, 20]
```

7.4.4.3.3.2 Format Reference

Field	Description
`version`	**Required.** Specifies the YAML schema version (e.g., `"25.04"`).
`DPA_APPS`	**Required.** List of applications assigned to the partition.
`name`	**Required.** Application name (e.g., `APP1`).
`number_of_affinity_EUs`	Optional. Number of affinity EUs assigned to the app.
`affinity_EUs`	Optional. List of **relative EU IDs** (converted from absolute).
`number_of_groups`	Optional. Number of groups assigned to the app.
`groups`	Optional. List of group indices (integer values); original group names are included as comments.

⚠ The affinity_EUs values are relative to the partition in which the app is declared. The tool calculates them based on absolute values provided in the input YAML.

7.4.4.3.3.3 Format of output YAML file

```
version: "string"  # Version identifier (e.g., "25.04")
        # REQUIRED: version must be defined

DPA_APPS:  # List of applications with their attributes
  - name: "string"  # Name of the application (e.g., "APP3", "APP4")
         # REQUIRED: name must be defined

    #  (Optional) Number of affinity Execution Units (EUs) assigned to the application
    number_of_affinity_EUs: "integer"

    #  (Optional) List of affinity EU identifiers
    affinity_EUs:
      type: array
      items: integer  # Example: [17, 19] or [18]

    # (Optional) Number of groups assigned to the application
    number_of_groups: "integer"

    # (Optional) List of group identifiers
    groups:
      type: array
      items: integer  # Example: [1] (corresponds to "EU3" in the comment)
```

7.4.4.3.3.4 Key Behaviors

- Relative EU IDs – EUs are converted to partition-relative indices; this improves portability and compatibility across deployments
- Group Mapping – Group names from the input YAML are mapped to numeric indices; the mapping is shown as comments for traceability
- Partition Isolation – Each output file corresponds to a single partition, containing only the apps assigned to that partition

7.4.4.4 SPRD Configuration and Management Flow

All operations are performed using **`dpa-resource-mgmt -d mlx5_X`**, where **`mlx5_X`** is the root device name.

7.4.4.4.1 Gather Application Requirements

The system administrator receives the following details from users and developers:

- A list of applications to be used
- A list of devices on which the applications will run
- Resource requirements for each application (e.g., number of EUs, group needs)

7.4.4.4.2 Map Devices to VHCAs

Run the following command to determine the mapping between devices and VHCAs using the **`pci_bdf`** field:

```
dpa-resource-mgmt query -d mlx5_X -t vhca
```

You can also run **`sudo mst status -v`** from the MFT package to obtain device details.

7.4.4.4.3 Inspect Core and EU Layout

Run the following command to retrieve chip-level information such as core counts, EU layout, and absolute EU IDs:

```
dpa-resource-mgmt query -d mlx5_X -t chip
```

7.4.4.4.4 Create Partition Configuration

Using the gathered data, create a YAML configuration file with the following:

- Partition definitions.
- EU allocation (either by core/num_EUs or absolute EU IDs).
- EU groups within partitions (optional).
- Root partition groups (optional).

7.4.4.4.5 Define Application Assignments

Add the application mapping section to the YAML configuration file, based on:

- Application names
- Resource requirements
- Target partitions and group assignments

7.4.4.4.6 Validate Configuration

Run the following command to validate the YAML configuration:

```
dpa-resource-mgmt config -d mlx5_X -f <config.yaml>
```

7.4.4.4.7 Handle Conflicts (If Any)

If the tool returns an error due to existing partitions, groups, or active processes:

1. Clear the existing configuration:

   ```
   dpa-resource-mgmt clear -d mlx5_X
   ```

2. If clearing does not resolve the issue, perform a full chip reset

7.4.4.4.8 Review the Resulting Setup

Once configuration is successful, analyze the resulting setup:

```
dpa-resource-mgmt query -d mlx5_X -t resources
```

Use this step to verify if EU and group allocations are optimal.

7.4.4.4.9 Iterate (Optional)

If changes are required, modify the YAML file and revalidate configuration.

7.4.4.4.10 Distribute Output Files

After finalizing the configuration:

- Retrieve the output YAML files generated by **dpa-resource-mgmt**
- Distribute these files to the respective hosts where applications will run
- Notify users and developers to begin using their assigned resources

7.4.4.4.11 Revise as Needed

If any issues arise or new requirements are introduced, repeat the process from the start.

7.4.4.5 Flex IO SDK API for SPRD Support

To integrate SPRD with a host application, the typical workflow is:

1. Read a binary resource file generated by the **dpa-resource-mgmt** utility.
2. Pass the contents to **flexio_resources_get()** to populate a **flexio_res** structure.
3. Use the accessor functions to obtain execution unit (EU) or execution group (EUG) information.
4. Use this data to configure affinity for creating a **flexio_event_handler**.
5. After use, release the allocated structure using **flexio_resources_destroy()**.

7.4.4.5.1 Core API Functions

7.4.4.5.1.1 Main Functions

- **flexio_resources_get**

```
/**
 * Read resources for a given key from an SPRD resource buffer.
 *
 * @param[in] key      Application name as defined in the SPRD YAML output.
 * @param[in] buf      Binary buffer from dpa-resource-mgmt output.
 * @param[in] buf_len  Buffer size in bytes.
 * @param[out] res     Output pointer to a populated flexio_res structure.
 *
 * @return flexio_status enum value.
 */
flexio_status flexio_resources_get(const char *key, const void *buf, size_t buf_len,
                                   struct flexio_res **res);
```

- **flexio_resources_destroy**

```
/**
 * Free memory allocated for a flexio_res structure.
 *
 * @param[in] res  Pointer to the structure to destroy.
 *
 * @return flexio_status enum value.
 */
flexio_status flexio_resources_destroy(struct flexio_res *res);
```

7.4.4.5.1.2 Accessor Functions

- Get application name:

```
/**
 * @brief Get application name from a Flex IO resources struct.
 *
 * This function gets the application name from a Flex IO resources struct.
 *
 * @param[in] res - A pointer to a Flex IO resources struct.
 *
 * @return application name.
 */
FLEXIO_EXPERIMENTAL char *flexio_resources_get_app_name(struct flexio_resource *res);
```

- Number of execution groups:

```
/**
 * @brief Get number of execution groups from a Flex IO resources struct.
 *
 * This function gets the number of execution groups from a Flex IO resources struct.
 *
 * @param[in] res - A pointer to a Flex IO resources struct.
 *
 * @return number of execution groups.
 */
FLEXIO_EXPERIMENTAL int flexio_resources_get_eugs_num(struct flexio_res *res);
```

- Execution group list:

```
/**
 * @brief Get execution unit groups list from a Flex IO resources struct.
 *
 * This function gets the execution unit groups list from a Flex IO resources struct.
 *
 * @param[in] res - A pointer to a Flex IO resources struct.
 *
 * @return execution unit groups list.
 */
FLEXIO_EXPERIMENTAL uint32_t *flexio_resources_get_eugs(struct flexio_res *res);
```

- Number of execution units:

```
/**
 * @brief Get execution units number from a Flex IO resources struct.
 *
 * This function gets the execution units number from a Flex IO resources struct.
 *
 * @param[in] res - A pointer to a Flex IO resources struct.
 *
 * @return execution units number.
 */
FLEXIO_EXPERIMENTAL int flexio_resources_get_eus_num(struct flexio_res *res);
```

- Execution unit list:

```
/**
 * @brief Get execution units list from a Flex IO resources struct.
 *
 * This function gets the execution units list from a Flex IO resources struct.
 *
 * @param[in] res - A pointer to a Flex IO resources struct.
 *
 * @return execution units list.
 */
FLEXIO_EXPERIMENTAL uint32_t *flexio_resources_get_eus(struct flexio_res *res);
```

7.4.4.5.2 Example of Using EU Resources

```
file = fopen(filename, "rb");
if (!file) {
        printf("Cannot open file!n");
        return ERROR;
}

fseek(file, 0, SEEK_END);
buf_len = ftell(file);
```

```
if (buf_len == 0) {
        printf("Empty file!\n");
        return ERROR;
}
fseek(file, 0, SEEK_SET);

buf = calloc(sizeof(char), buf_len);
if (!buf) {
        printf("Cannot allocate\n");
        return ERROR;
}

if (buf_len != fread(buf, sizeof(char), buf_len, file)) {
        printf("Incorrect read\n");
        return ERROR;
}

fclose(file);

/* ..............  */

ret = flexio_resources_get(key, buf, buf_len, &resourses);

if (ret != FLEXIO_STATUS_SUCCESS) {
        printf("Failed to read resources for key %s\n", key);
        return ERROR;
}

free(buf);

/* ..............  */

attr.affinity.type = FLEXIO_AFFINITY_GROUP;
attr.affinity.id = flexio_resourses_get_eugs(resourses)[0];
ret = flexio_event_handler_create(process, &attr, &eug_handle_id);
if (ret != FLEXIO_STATUS_SUCCESS) {
        printf("Failed to create event_handler");
        return ERROR;
}

attr.affinity.type = FLEXIO_AFFINITY_STRICT;
attr.affinity.id = flexio_resourses_get_eus(resourses)[0];
ret = flexio_event_handler_create(process, &attr, &eu_handle_id);
if (ret != FLEXIO_STATUS_SUCCESS) {
        printf("Failed to create event_handler");
        return ERROR;
}

/* ..............  */

ret = flexio_resources_destroy(resources);
if (ret != FLEXIO_STATUS_SUCCESS) {
        printf("Failed to free resources");
        return ERROR;
}
```

7.4.4.6 Back-End Tool - dpaeumgmt

The **dpaeumgmt** tool enables management of DPA Execution Units (EUs), the fundamental compute resource of the Data Path Accelerator (DPA). It can run either on the host or on a target DPU and provides control over EU allocation, ensuring optimal resource distribution before using the DOCA FlexIO SDK API.

Without explicit EU allocation, a DPA software thread cannot execute, as it would lack access to pipeline or compute time resources.

Key use cases:

- Strict affinity execution – Run a DPA thread on a specifically assigned EU. **dpaeumgmt** can query the highest allowed EU ID for this purpose.
- Group Affinity Execution – Define EU groups. Once a group is created and assigned a unique ID, DPA threads can use group affinity to select an available EU from within the group.
- EU Partition Management – Manage EU partitions to logically separate compute resources across different virtual host channel adapters (VHCAs).

⚠ A DPA thread can only run if a suitable EU (according to its affinity type) is available.

7.4.4.6.1 Execution Unit Objects

At system boot, a default EU partition is automatically created. This partition initially owns all system EUs and is managed exclusively by the DPA partition manager function.

- When a DPA thread runs with `none` affinity, an EU is selected from the default partition's pool—excluding any EUs already allocated to groups.
- If no free EUs remain in the default pool, the thread fails to run.

7.4.4.6.2 dpaeumgmt CLI Reference

7.4.4.6.2.1 General Commands

Command	Description
`dpaeumgmt -h`	Print basic usage information
`dpaeumgmt help`	Show detailed help for all commands
`dpaeumgmt version`	Print the version of the tool

7.4.4.6.2.2 EU Group Commands

Command	Description
`dpaeumgmt eu_group info -d <device>`	Show system limits and max values for EU groups
`dpaeumgmt eu_group create -d <device> -n <name> -r <range>`	Create an EU group with a name and EU range
`dpaeumgmt eu_group destroy -d <device> -g <id>`	Destroy an EU group by ID
`dpaeumgmt eu_group destroy -d <device> -n <name>`	Destroy an EU group by name
`dpaeumgmt eu_group query -d <device>`	Query all EU groups on the device
`dpaeumgmt eu_group query -d <device> -n <name>`	Query a specific EU group by name
`dpaeumgmt eu_group query -d <device> -g <id>`	Query a specific EU group by ID
`dpaeumgmt eu_group apply -d <device> --file_groups <file>`	Replace EU groups with ones specified in the input JSON file

7.4.4.6.2.3 EU Partition Commands

Command	Description
`dpaeumgmt partition info -d <device>`	Show partition-related system limits
`dpaeumgmt partition create -d <device> -v <vhca_list> -r <range> -m <max_groups>`	Create a new partition with specified VHCA IDs and EU range

Command	Description
`dpaeumgmt partition destroy -d <device> -p <partition_id>`	Destroy a partition by ID
`dpaeumgmt partition query -d <device>`	Query all partitions
`dpaeumgmt partition query -d <device> -p <partition_id>`	Query a specific partition

7.4.4.6.2.4 Info Commands

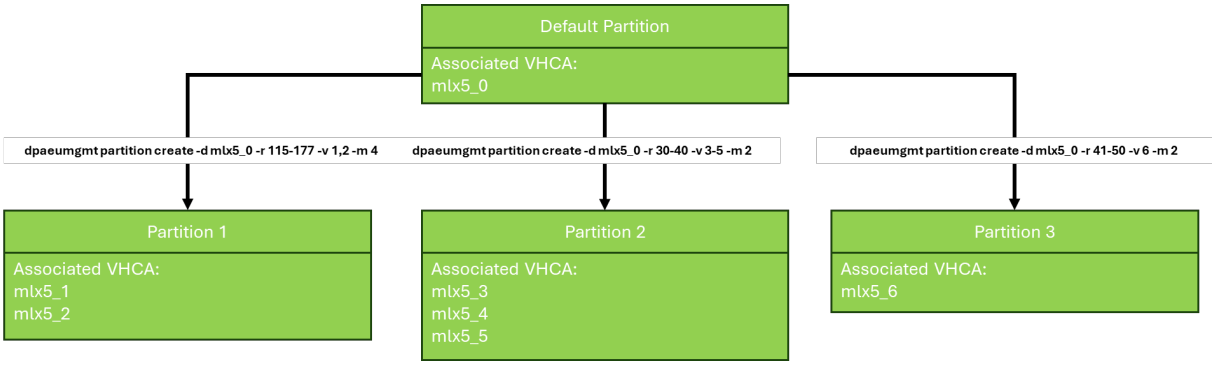
Command	Description
`dpaeumgmt info vhca -d <device>`	Show vHCA mapping and hierarchy
`dpaeumgmt info vhca -d <device> -y <yaml_file>`	Save vHCA info to YAML file
`dpaeumgmt info chip -d <device>`	Show chip-level resource info (cores, EUs, etc.)
`dpaeumgmt info chip -d <device> -y <yaml_file>`	Save chip info to YAML file
`dpaeumgmt info status -d <device>`	Show current state: partitions, root groups, processes
`dpaeumgmt info status -d <device> -y <yaml_file>`	Save status to YAML file

7.4.4.6.2.5 Common Flags and Options

Flag	Description
`-d` , `--dpa_device`	The device interface name (e.g., mlx5_0)
`-r` , `--range_eus`	EU range for allocation (e.g., "4-8,16,22")
`-v` , `--vhca_list`	Comma-separated list or range of VHCA IDs (e.g., "1,2,3", "4-6")
`-m` , `--max_num_eu_group`	Max number of groups in a partition
`-n` , `--name_group`	Group name (string up to 15 characters)
`-g` , `--id_group`	Group ID (integer)
`-p` , `--id_partition`	Partition ID (integer)
`-f` , `--file_groups`	Path to JSON file describing EU groups
`-y` , `--yaml_file`	Path to YAML file to write output

7.4.4.6.3 vHCAs and Partitions

The following diagram illustrates the ownership and control of a partition by a vHCA and also which vHCAs have claim to (i.e., can use) a partition.



7.4.4.6.4 Known Limitations of dpaeumgmt

Limitation	Description
Tool must run before DPA process creation	dpaeumgmt must be executed before any DPA process is started. Fails if run over an active process.
EU group names must be unique	Duplicate group names within the same partition are not allowed.
Partition creation consumes group quota	Each new partition reserves one default group and up to 'max_num' groups specified at creation.
Non-ECPF vHCAs require partitions	vHCAs other than ECPF must have a configured partition before using EU resources.
Only default partition sees real EU IDs	Other partitions use virtual EU numbers starting from 0.
Group IDs are virtual and scoped per partition	Same group ID can exist in multiple partitions with different semantics.
vHCA ID overlap not allowed	A vHCA cannot belong to multiple partitions.
vHCA IDs not discoverable via dpaeumgmt	User must know vHCA IDs in advance, e.g., via 'mst status -v'.
Partition deletion blocked by existing EU objects	All EU groups must be removed before destroying a partition.
EU assignment is opaque	No way to query which EU is selected at runtime.
vHCA scope is isolated	Each vHCA only sees its partition's groups, EUs, and limits.
Group ID assignment is not guaranteed	IDs are dynamically assigned and not persistent.
Default groups are not user-managed	System-created default groups cannot be edited or deleted.
EU ID range bounded	Valid IDs are between 0 and (max EU ID - 1), retrievable via 'info chip'.
No VF support	Virtual Functions are not supported by dpaeumgmt.
Partitions created only from DPA partition manager	Partition creation must be initiated from the manager function (e.g., ECPF).
Max 16 hardware EU groups	System limit of 16 concurrent EU group entities.

7.4.5 DOCA DPA GDB Server Tool

This document describes the DPA GDB Server tool.

> The DPA GDB Server Tool is currently supported at beta level.

7.4.5.1 Introduction

The DPA GDB Server tool (**dpa-gdbserver**) enables debugging FlexIO DEV programs.

DEV programs for debugging are selected using a token (8-byte value) provided by the FlexIO process owner.

> Any GDB, familiar with RISC-V architecture, can be used for the debug. Refer to this page for information how to work with GDB.

7.4.5.1.1 Glossary

Term	Description
PUD	Process under debug. DEV-side processes intended for debug.
EU	Execution unit (similar to hardware CPU core)
DPA	Data path accelerator
RPC	Remote process communication. Mechanism used in FlexIO to run DEV-side code instantly. Runtime is limited to 6 seconds.
HOST	x86 or aarch64 Linux OS which manages dev-side code (i.e., DEV)
DEV	RISC-V code, loaded by HOST into the DPA's device. Triggered to run by different types of interrupts. DEV side is directly connected to ConnectX adapter card.
GDB	GNU Project debugger. Allows users to monitor another program while it executes.
GDBSERVER	Tool for remote debug programs
RTOS	Real-time operation system running on RISC-V core. Manages handling of interrupts and calls to DEV user processes routines.
RSP	Remote serial protocol. Used for interaction between GDB and GDBSERVER.

7.4.5.1.2 Package Contents

The **dpa-gdbserver** package contains the **dpa-gdbserver** executable, which runs on the DPA to facilitate debugging.

This package does not contain the host-side debugger, **gdb-multiarch** , or an equivalent multi-architecture GDB application. Users must obtain and install a GDB version that supports the RISC-V architecture (such as **gdb-multiarch**) for their specific environment to interact with the server.

7.4.5.1.3 Known Limitations

- DPA GDB technology does not catch fatal errors. Therefore, if a fatal error occurs, core dump (created by **flexio_coredump_create()**) should be used.
- DPA GDB technology does not support Outbox access. GDB users cannot write to Doorbell or to Window configuration areas.
- DPA GDB technology does not support Window access. Read/write to Window memory does not work properly.

7.4.5.2 DPA-specific Notes

7.4.5.2.1 Token

The process under debug (PUD) can expose a debugging token. Every external process, using this token, get full access to the process with given token. To not show it constantly (e.g., for security reasons), users can modify their host application temporary. See **flexio_process_udbg_token_get()** .

7.4.5.2.2 Connection on Application Launch

If the code which needs debugging begins to run immediately after launch, the user should modify the host application to stop upon start to give the user time to run **dpa-gdbserver** . One possible way of doing this is to place function **getchar()** immediately after process creation.

7.4.5.2.3 Dummy Thread Concept

Something to consider with DPA debugging is that a PUD does not have a running thread all time (e.g., the process's thread may exist but be waiting for incoming packets). In a regular Linux application, this scenario is not possible and GDB does not support such cases.

Therefore, when no thread is running, **dpa-gdbserver** reports a dummy thread:

gdb

```
(gdb) info thread
  Id   Target Id                                Frame
* 1    Thread 1.805378433 (Dummy Flexio thread) 0x0800000000000000 in ?? ()
(gdb)
```

In this case user can inspect memory, create breakpoints, and give the **continue** command.

Commands like **step** , **next** , and **stepi** can not be executed for the Dummy thread.

7.4.5.2.4 Watchdog Issues

The RTOS has a watchdog timer that limits DEV code interrupt processes to 100 seconds. This timer is stopped when the user connects to DEV with GDB. Therefore users will have no time limitation for debugging.

7.4.5.3 Tool TCP Port and Execution Unit (EU)

By default, **dpa-gdbserver** uses TCP port 1981 and runs on EU 29. If this conflicts with another application (or if other instances of **dpa-gdbserver** are running), users should change the defaults as follows:

Bash

```
$> dpa-gdbserver mlx5_0 -T <token> -s <port> -E <eu_id>
```

7.4.5.4 Debugging

7.4.5.4.1 Preparation for Debug

Modify your FlexIO application if needed. Make sure the HOST code prints **udbg_token** and waits for GDB connection if needed:

C code. Host side. diff

```
+   uint64_t udbg_token;

    flexio_process_create(..., &flexio_process);

+   udbg_token = flexio_process_udbg_token_get(flexio_process);
+   if (udbg_token)
+       printf("Process created. Use token >>> %#lx <<< for debug\n", udbg_token);

+   printf("Stop point for waiting of GDB connection. Press Enter to continue..."); /* Usually you don't need this
stop point */
+   fflush(stdout);
+   getchar();
```

Extract the DPA application from the FlexIO application. For example:

Bash

```
$> dpacc-extract cc-host/app/host/flexio_app_name -o flexio_app_name.rv5
```

7.4.5.4.2 Start Debugging

1. Run your FlexIO application. It should expose the debug token:

   Bash

   ```
   $> flexio_app_name mlx5_0
   Process created. Use token >>> 0xd6278388ce4e682c <<< for debug
   ```

2. Run **dpa-gdbserver** with the debug token received:

Bash

```
$> dpa-gdbserver mlx5_0 -T 0xd6278388ce4e682c
Registered on device mlx5_0
Listening for GDB connection on port 1981
```

3. Run any GDB with RISC-V support. For example, **gdb-multiarch**:

Bash

```
$> gdb-multiarch -q flexio_app_name.rv5
Reading symbols from flexio_app_name.rv5...
(gdb)
```

4. Connect to the gdbserver using proper TCP port and hostname, if needed:

gdb

```
(gdb) target remote :1981
Remote debugging using :1981
0x0800000000000000 in ?? ()
```

7.4.5.4.3 DPA-specific Debugging Techniques

7.4.5.4.3.1 Easy Example of Transitioning from Dummy to Real Thread

Transitioning between the dummy thread and a real thread is not standard practice for debugging under GDB. In an ideal situation, the user would know exactly the entry points for all their routines and can set breakpoints for all of them. Then the user may run the **continue** command:

gdb

```
(gdb) target remote :1981
Remote debugging using :1981
0x0800000000000000 in ?? ()
(gdb) info threads
  Id   Target Id                                 Frame
* 1    Thread 1.805378433 (Dummy Flexio thread) 0x0800000000000000 in ?? ()
(gdb) b foo
Breakpoint 1 at 0x400000b2: file ../tests/path/hello.c, line 58.
(gdb) b bar
Breakpoint 2 at 0x40000518: file ../tests/path/hallo.c, line 113.
(gdb) continue
Continuing.
```

Initiate interrupts for your DEV program (depends your task), and GDB should catch a breakpoint and now the real thread of the PUD appear instead of the dummy:

gdb

```
(gdb) continue
Continuing.
(gdb) [New Thread 1.2]
[New Thread 1.130]
[New Thread 1.258]
[New Thread 1.386]
[Switching to Thread 1.2]

Thread 2 hit Breakpoint 1, foo(thread_arg=9008)
    at ../tests/path/hello.c:58
58              struct host_data *hdata = NULL;
```

```
(gdb) info threads
  Id   Target Id                                    Frame
* 2    Thread 1.2 (Process 0 thread 0x1 GVMI 0 EU 0)     foo (arg=9008) at ../tests/path/hello.c:58
  3    Thread 1.130 (Process 0 thread 0x81 GVMI 0 EU 1)  foo (arg=9264) at ../tests/path/hello.c:58
  4    Thread 1.258 (Process 0 thread 0x101 GVMI 0 EU 2) foo (arg=9648) at ../tests/path/hello.c:58
  5    Thread 1.386 (Process 0 thread 0x181 GVMI 0 EU 3) foo (arg=9904) at ../tests/path/hello.c:58
(gdb)
```

From this point, you may examine memory and trace your code as usual.

7.4.5.4.3.2 Complicated Example of Transitioning from Dummy to Real Thread

In a more complicated situation, the interrupt happens after GDB connection. In this case, the real thread should start running but cannot because the PUD is in HALT state. The user can type the command **info threads**, see new thread instead of the old dummy, and then switch to the new thread manually:

gdb

```
1   (gdb) target remote :1981
2   Remote debugging using :1981
3   0x0800000000000000 in ?? ()
4   (gdb) info threads
5     Id   Target Id                                  Frame
6   * 1    Thread 1.805378433 (Dummy Flexio thread) 0x0800000000000000 in ?? ()
7   (gdb) info threads
8   [New Thread 1.32769]
9     Id   Target Id                                        Frame
10    2    Thread 1.32769 (Process 0 thread 0x8000 GVMI 0 EU 3) bar (arg=0xc0, len=0)
11      at /path/lib/src/stub.c:167
12
13  The current thread <Thread ID 1> has terminated.  See `help thread'.
14  (gdb) thread 2
15  [Switching to thread 2 (Thread 1.32769)]
16  #0  bar (arg=0xc0, len=0)
17      at /path/lib/src/stub.c:167
18  167     {
19  (gdb) bt
20  #0  bar (arg=0xc0, len=0)
21      at /path/lib/src/stub.c:167
22  #1  0x000000004000017a in foo (thread_arg=3221)
23      at ../path/dev/hello.c:182
24  #2  0x0000000000000000 in ?? ()
25  Backtrace stopped: frame did not save the PC
26  (gdb)
```

> ⚠ The same command **info threads** in lines 4 and 7 gives different results. This happens because the interrupt occurs between the instances and the real code begins to run.

The user must switch to the new thread manually (see line 14). After this, they can trace/debug the flow as usual (i.e., using the commands **step**, **next**, **stepi**).

7.4.5.4.3.3 Finishing Real Thread without Finishing PUD

Every interrupt handler at some point finishes its way and returns the CPU resources to RTOS. The most common way to do this is to call function **flexio_dev_thread_reschedule()**. The command **next** on this function will have the same effect as the command **continue**:

gdb

```
205             __dpa_thread_fence(__DPA_MEMORY, __DPA_W, __DPA_W);
(gdb) next
206             flexio_dev_cq_arm(dtctx, app_ctx.rq_cq_ctx.cq_idx, app_ctx.rq_cq_ctx.cq_number);
(gdb) next
208             if ((dev_errno = flexio_dev_get_and_rst_errno(dtctx))) {
(gdb) next
213             print_sim_str("Nothing to do. Wait for next duar\n", 0);
(gdb) next
```

```
214             flexio_dev_thread_reschedule();
(gdb) next
```

GDB waits until the user types **^C** or a breakpoint is reached after the next interrupt occurred.

7.4.5.5 Error Reporting

The DPA GDB server tool has been validated with **gdb-multiarch** (version 9.2) and with GDB version 12.1 from RISC-V tool chain.

The GDB server should support all commands described in GDB RSP (remote serial protocol) for GDB stubs. But only the most common GDB commands are supported.

Should a dpa-gdbserver bug occur, please provide the following data:

- Used GDB (name and version)
- Commands sequence to reproduce the issue
- DPA GDB server tool console output
- DPA GDB server tool log directory content (see next part for details)
- Optional – output data printed when **dpa-gdbserver** is run in verbose mode

7.4.5.5.1 Tool Log Directory

For every run, a temporary directory is created with the template **/tmp/flexio_gdbs.XXXXXX** .

To locate the latest one, run the following command:

Bash

```
$> ls -ldtr /tmp/flexio_gdbs.* | tail
```

7.4.5.5.2 Verbosity Level of gdbserver

By default, **dpa-gdbserver** does not print any log information to screen. Adding - **v** option to command line increases verbosity level, printing additional info to **dpa-gdbserver** terminal display. Verbosity level is incremented according to number of 'v' in command line switch (i.e. **-vv** , **-vvv** etc.).

One **-v** shows the RSP exchange. This is a textual protocol, so users can read and understand requests from GDB and answers from the GDB server:

gdbserver.log -v

```
<<<<< "qTStatus"
>>>>> ""
```

```
<<<<< "?"
>>>>> "S05"
<<<<< "qfThreadInfo"
>>>>> "mp01.30011981"
<<<<< "qsThreadInfo"
>>>>> "l"
<<<<< "qAttached:1"
>>>>> "1"
<<<<< "Hc-1"
>>>>> "OK"
<<<<< "qC"
>>>>> "QCp01.30011981"
```

In the examples, `<<<<<` and `>>>>>` are used to indicate data received from GDB and transmitted to GDB, respectively.

When running with a higher verbosity level (e.g., run `dpa-gdbserver` with option `-vv` or higher), the exchange with the RTOS module is shown:

`gdbserver.log -vv`

```
<<<<< "qfThreadInfo"
/  2/dgdbs_handler - cmd 0x5
/  2/dgdbs_handler - retval 0x4
>>>>> "mp01.30011981"
<<<<< "qsThreadInfo"
/  2/dgdbs_handler - cmd 0x5
/  2/dgdbs_handler - retval 0x5
>>>>> "l"
<<<<< "m800000000000000,4"
/  2/dgdbs_handler - cmd 0xc
/  2/dgdbs_handler - retval 0x9
>>>>> "E0a"
<<<<< "m7ffffffffffffc,4"
/  2/dgdbs_handler - cmd 0xc
/  2/dgdbs_handler - retval 0x9
>>>>> "E0a"
<<<<< "qSymbol::"
>>>>> "OK"
```

Lines beginning with `/ #/` provide the number of internal RTOS threads printed from the DEV side.

7.4.5.6 Useful Info Regarding Work with GDB

This section provides useful information about commands and methods which can help users when performing DPA debug. This is not related to the `dpa-gdbserver` itself. But this is about remote debugging and FlexIO sources.

7.4.5.6.1 Command "directory"

GDB can run on a different host from the one where compilation was done. For example, users may have compiled and run their application on `host1` and run their instance of GDB on `host2`. In this case, users will see the error message `../xxx/yyy/zzz/your_file.c: No such file or directory`. To solve this problem, copy sources to the host running GDB (`host2` in the example). Make sure to save the original code hierarchy. Use GDB command `directory` to inform where the sources are to GDB:

gdb on host2

```
host2~$> gdb-multiarch -q /tmp/my_riscv.elf
Reading symbols from /tmp/my_riscv.elf...
(gdb) b foo
Breakpoint 1 at 0x4000016c: file ../xxx/yyy/zzz/my_file.c, line 182.
(gdb) target remote host1:1981
Remote debugging using host1:1981
0x0800000000000000 in ?? ()
(gdb) c
Continuing.
[New Thread 1.32769]
[Switching to Thread 1.32769]

Thread 2 hit Breakpoint 1, foo (thread_arg=5728) at ../xxx/yyy/zzz/my_file.c:182
182     ../xxx/yyy/zzz/my_file.c: No such file or directory.
(gdb) directory /tmp/apps/
Source directories searched: /tmp/apps:$cdir:$cwd
(gdb) list
179             struct flexio_dev_thread_ctx *dtctx;
180             uint64_t dev_errno;
181
182             print_sim_str("=====> NET event handler started\n", 0);
183
184             flexio_dev_print("Hello GDB user\n");
185
```

Pay attention to the exact path reported by GDB. The argument for the command **directory** should point to the start point for this path. For example, if GDB looks for **../xxx/yyy/zzz** and you placed the sources in local directory **/tmp/copy_of_worktree**, then the command should be **(gdb) directory /tmp/copy_of_worktree/xxx/** and not **(gdb) directory /tmp/copy_of_worktree/**.

Sometimes, the ***.elf** file provides a global path from the root. In this case, use the command **set substitute-path <from> <to>**. For example, if the file **`/foo/bar/baz.c`** was moved to **`/mnt/cross/baz.c`**, then the command **(gdb) set substitute-path /foo/bar /mnt/cross** instructs GDB to replace **/foo/bar** with **`/mnt/cross`**, which allows GDB to find the file **`baz.c`** even though it was moved.

See this page of GDB documentation for more examples of specifying source directories.

7.4.5.6.2 Core Dump Usage

If the code runs into a fatal error even though the host side of your project is implemented correctly, a core dump is saved which allows analyzing the core. It should point exactly to where the fatal error occurred. The command **backtrace** can be used to examine the memory and its registers. Change the frame to see local variables of every function on the backtrace list:

gdb

```
$> gdb-multiarch -q -c crash_demo.558184.core /tmp/my_riscv.elf
Reading symbols from /tmp/my_riscv.elf...

[New LWP 1]
#0  0x000000004000126e in read_test (line=153, ptr=0x30) at /xxx/yyy/zzz/my_file.c:109
109             val = *(volatile uint64_t *)ptr;
(gdb) bt
#0  0x000000004000126e in read_test (line=153, ptr=0x30) at /xxx/yyy/zzz/my_file.c:109
#1  0x000000004000031a in tlb_miss_test (op_code=1) at /xxx/yyy/zzz/my_file.c:153
#2  0x0000000040000144 in test_thread_err_events_entry_point (h2d_daddr=3221258560) at /xxx/yyy/zzz/my_file.c:588
#3  0x00000000400013fc in _dpacc_flexio_dev_arg_unpack_test_err_events_dev_test_thread_err_events_entry_point
(argbuf=0xc0008228, func=0x400000b0 <test_thread_err_events_entry_point>)
    at /tmp/dpacc_xExkvE/test_err_events_dev.dpa.device.c:67
#4  0x0000000040001680 in flexio_hw_rpc (host_arg=3221258752) at /local_home/www/flexio-sdk/libflexio-dev/src/
flexio_dev_entry_point.c:75
#5  0x0000000000000000 in ?? ()
Backtrace stopped: frame did not save the PC
```

```
(gdb) frame 4
#4  0x0000000040001680 in flexio_hw_rpc (host_arg=3221258752) at /local_home/igorle/flexio-sdk/libflexio-dev/src/
flexio_dev_entry_point.c:75
75                      retval = unpack_cb(&data_from_host->func_params.arg_buf,
(gdb) p /x *data_from_host
$2 = {poll_lkey = 0x1ff2b1, window_id = 0x3, poll_haddr = 0x55dc0f40b900, entry_point = 0x400013d8, func_params =
{func_wo_pack = 0x0, dev_func_entry = 0x400000b0, arg_buf = 0xc0008140}}
(gdb)
```

7.4.5.6.3 Debug of Optimized Code

Usually highly optimized code is compiled and run.

Two types of mistakes in code can be considered:

- Logical errors
- Optimization-related errors

Logical errors (e.g., using `&` instead of `&&`) are reproduced on the non-optimized version of the code. Optimization related errors (e.g., forgetting volatile classification, non-usage of memory barriers) only impact optimization. Non-optimized code is much easier for tracing with GDB, because every C instruction is translated directly to assembly code.

It is good practice to check if an issue can be reproduced on non-optimized code. That helps observe the application flow:

Bash

```
$> build.sh -O 0
```

For tracing this code, using GDB commands `next` and `step` should be sufficient.

But if an issue can only be reproduced on on optimized code, you should start debugging it. This would require reading disassembly code and using the GDB command `stepi` because it becomes a challenge to understand exactly which C-code line executed.

7.4.5.6.4 Disassembly of Advanced RISC-V Commands

DPA core runs on a RISC-V CPU with an extended instruction set. The GDB may not be familiar with some of those instructions. Therefore, `asm` view mode shows numbers instead of disassembly. In this case it is recommended to disassemble your RISC-V binary code manually. Use the `dpa-objdump` utility with the additional option `--mcpu=nv-dpa-bf3`.

bash

```
$> dpa-objdump -sSdxl --mcpu=nv-dpa-bf3 my_riscv.elf > my_riscv.asm
```

The following screenshot shows the difference:

```
4000057a: 03 35 84 fe   ld  a0, -24(s0)
4000057e: 08 65         ld  a0, 8(a0)
40000580: 1355856b
40000584: e2 60         ld  ra, 24(sp)
40000586: 42 64         ld  s0, 16(sp)
40000588: 05 61         addi    sp, sp, 32
4000058a: 82 80         ret
```

```
4000057a: 03 35 84 fe   ld  a0, -24(s0)
4000057e: 08 65         ld  a0, 8(a0)
40000580: 13 55 85 6b   rev8    a0, a0
40000584: e2 60         ld  ra, 24(sp)
40000586: 42 64         ld  s0, 16(sp)
40000588: 05 61         addi    sp, sp, 32
4000058a: 82 80         ret
```

7.4.6 DOCA DPA PS Tool

7.4.6.1 Introduction

DOCA **dpa-ps** is a CLI tool which allows users to monitor running DPA processes and threads. The tool presents sorted lists of the currently running DPA processes and threads.

The process ID output of the **dpa-ps** tool may be used as the input parameter for the **dpa-statistics** tool.

This tool is supported for NVIDIA® BlueField®-3 only.

7.4.6.2 Command Flags and Arguments

The following table lists the flags for the **dpa-ps** tool.

Short Option	Long Option	Description
-h	**--help**	Help information
-d	**--device**	Device interface name (MST/RDMA)
-p	**--process-id**	Hexadecimal process ID for filtering
-t	**--threads**	Show threads info for each process
-i	**--suppress-header-info**	Suppress print header info

Arguments for the flags must be used within quotes (if more than one) and without extra spaces.

7.4.6.3 Example

```
$ sudo ./dpa-ps -d mlx5_0 -t
ProcessID     Process Name
    ThreadID      Thread Name
0             PROCESS_0
    5             EH_0_5_5
    6             EH_0_5_5
```

```
1               PROCESS_1
    3               EH_1_3_3
    4               EH_1_4_4
2               PROCESS_2
3               PROCESS_3
    0               EH_3_0_0
    1               EH_3_1_1
    2               EH_3_2_2
4               PROCESS_4
```

7.4.6.4 Known Limitations

- The **dpa-ps** and **dpa-statistics** tools cannot be run at the same time on the same device

7.4.7 DOCA DPA Statistics Tool

7.4.7.1 Introduction

DOCA **dpa-statistics** is a CLI tool which allows users to monitor and obtain statistics on thread execution per running DPA process and thread. The tool is used to expose information about the running DPA processes and threads and to collect statistics on DPA thread performance.

The tool presents performance information for running DPA threads, including the number of cycles and instructions executed in a time period. The tool enables initiating and stopping collection of statistics and displaying the data collected per thread.

> The process ID output of the **dpa-ps** tool may be used as the input parameter for the **dpa-statistics** tool.

> This tool is supported for NVIDIA® BlueField®-3 only.

7.4.7.2 Collecting Performance Statistics Data

The command **collect** works on four mutually exclusive modes:

- Enable mode – start collecting performance data
- Disable mode – stop collecting performance data
- Timeout mode – start collecting, wait with a timeout, stop collect and print info. User could break the wait with Ctrl-C command and then the timeout will be canceled and tool will disable statistics collection and prints the info with the actual time of the collect operation.
- Infinite mode – no special flags. Same as timeout mode but with infinite timeout. The tool awaits the Ctrl-C command to stop.

The following table lists the **collect** command's flags and arguments:

Short Option	Long Option	Description
-h	**--help**	Help information
-d	**--device**	Device interface name (MST/RDMA)

Short Option	Long Option	Description
`-p`	`--process-id`	Hexadecimal process ID for filtering This flag indicates a specific command for the command to operate on. Otherwise, statistics are collected from all processes.
`-i`	`--suppress-header-info`	Suppress print header info
`-n`	`--enable`	Enable collect info
`-o`	`--disable`	Disable collect info
`-t`	`--timeout`	Enable collect, wait with timeout, disable collect and print info Timeout value is in milliseconds. Examples for inputting timeout value: • 45 – 45 milliseconds • 45.55 – 45 milliseconds and 550,000 nanoseconds • .0005 – 500 nanoseconds • 45m55n – 45 milliseconds and 55 nanoseconds • 66n – 66 nanoseconds
`-r`	`--reset`	Reset counters before operation starting collect operation

7.4.7.3 Presenting Statistics List

Presenting performance statistics is applicable after initiating data collection.

The following table lists the `show` command's flags and arguments:

Short Option	Long Option	Description
`-h`	`--help`	Help information
`-d`	`--device`	Device interface name (MST/RDMA)
`-p`	`--process-id`	Hexadecimal process ID for filtering
`-i`	`--suppress-header-info`	Suppress print header info

Output example:

```
$ sudo ./dpa-statistics show -d mlx5_0 -p 1
ProcessID    Process Name
    ThreadID    Cycles              Instruction         Time              Executions        Thread Name
1            PROCESS_1
    3           266268              18193               164               41                EH_1_0_0
    4           411571              32727               252               47                EH_1_1_1
```

Where:

- **ProcessID** – The **dpa_process_object_id** to which the thread belongs
- **Process Name** – The **dpa_process_name** to which the thread belongs

- **ThreadID** – DPA thread object ID
- **Cycles** – Total EU cycles the thread used
- **Instruction** – Total number of instructions the thread executed
- **Time** – Total time in ticks the thread was active
- **Executions** – Total number of thread invocations
- **Thread Name** – The **dpa_thread_name**

7.4.7.3.1 Examples

- Example of **collect** in infinite mode for process 0 with suppress header info:

```
$ sudo ./dpa-statistics collect -d mlx5_0 -p 0 -i
...^C
Data collected for 4606 milliseconds 0 nanoseconds
0           PROCESS_0
    5           223964            13754             140               31          EH_0_5_5
    6           190130            13754             114               31          EH_0_6_6
```

- Example of **collect** in timeout mode with a timeout of 1 second and half a millisecond.

```
$ sudo ./dpa-statistics collect -d mlx5_0 -t 1000.500
Data collected for 1000 milliseconds 500000 nanoseconds
ProcessID    Process Name
    ThreadID    Cycles            Instruction       Time              Executions        Thread Name
0           PROCESS_0
    5           223964            13754             140               31                EH_0_5_5
    6           190130            13754             114               31                EH_0_6_6
1           PROCESS_1
    3           266268            18193             164               41                EH_1_3_3
    4           411571            32727             252               47                EH_1_4_4
2           PROCESS_2
3           PROCESS_3
    0           223205            13754             137               31                EH_3_0_0
    1           189896            13754             113               31                EH_3_1_1
    2           191796            13754             117               31                EH_3_2_2
4           PROCESS_4
```

- Example of enabling statistics collection with reset of counters.

```
$ sudo ./dpa-statistics collect -d mlx5_0 -n -r
```

- Example of disabling statistics collection.

```
$ sudo ./dpa-statistics collect -d mlx5_0 -o
```

7.4.7.4 Known Limitations

- Reading large statistics counter blocks takes a long time
- The **dpa-ps** and **dpa-statistics** tools cannot be run at the same time on the same device

7.4.8 DPA Application Security Attributes to Blob

7.4.8.1 Overview

dpa-app-attributes2blob is a tool for converting between FlexIO attributes YAML files and binary blob files, and vice versa. It is typically used to generate binary configuration blobs from human-readable YAML attributes, or to reverse the process for inspection and debugging.

7.4.8.2 Features

- Converts attributes YAML files to binary blobs (default mode)
- Converts binary blobs back to attributes YAML files
- Supports a nested **Feature_list** structure
- Validates input files and provides clear error messages
- Sets default values to **false** for unspecified fields
- Provides detailed feature descriptions in generated YAML files
- Supports error modes for blob-to-YAML conversion
- Configurable blob size for YAML-to-binary conversion
- Comprehensive bit validation in binary blobs

7.4.8.3 Usage

```
dpa-app-attributes2blob [--help] [--version] [mode] [error_mode] [blob_size] <input-file> <output-file>
```

Option	Description	Notes
--yaml2bin	Convert an attributes YAML file to a binary blob	Default
--bin2yaml	Convert a binary blob to an attributes YAML file	N/A
--restrictive	Treat invalid bits in a binary blob as errors	• Default • bin2yaml only
--notification	Treat invalid bits in a binary blob as warnings	bin2yaml only
--blob-size <size>	Specify the output size of the blob in bytes. The size must be a multiple of 4 and less than 1024.	• 8 bytes by default • yaml2bin only

7.4.8.3.1 Examples

- Convert YAML to binary blob:

```
dpa-app-attributes2blob --yaml2bin config.yaml output.bin
```

- Simple convert YAML to binary blob:

```
dpa-app-attributes2blob config.yaml output.bin
```

- Convert YAML to binary blob with custom size:

```
dpa-app-attributes2blob --yaml2bin --blob-size 16 config.yaml output.bin
```

- Convert binary blob to YAML:

```
dpa-app-attributes2blob --bin2yaml output.bin config.yaml
```

- Convert binary blob to YAML with warning mode:

```
dpa-app-attributes2blob --bin2yaml --notification output.bin config.yaml
```

7.4.8.4 Attributes YAML Format

- The attributes must have a top-level mapping containing only a **Feature_list** key.
- The **Feature_list** value must be a mapping containing feature flags.
- Field values must be boolean (**true** / **false** , **yes** / **no** , **on** / **off**).
- Fields not explicitly set default to **false** .
- Supported fields:
  - **Congestion_Control_Flow_Context_Access** : Enables the application to access the Congestion Control Flow Context.
  - **Programmable_Congestion_Control** : Enables the application to serve Programmable Congestion Control events.
  - **Generating_Empty_CQE** : Enables the application to generate an Empty CQE on a given CQ (e.g., for triggering other software entries).
  - **Device_Emulation** : Enables the application to emulate a PCIe device.

7.4.8.4.1 Example Attributes YAML

```
Feature_list:
  Congestion_Control_Flow_Context_Access: true
  Programmable_Congestion_Control: false
  Generating_Empty_CQE: true
  Device_Emulation: true
```

7.4.8.4.2 Generated YAML Output

When converting from binary to YAML, the tool generates descriptive comments:

```
# This file was generated by dpa-app-attributes2blob
# A dictionary of features enabled/disabled for the application
Feature_list:
  # Enables application to access Congestion Control Flow Context
  Congestion_Control_Flow_Context_Access: true
  # Disables application to serve Programmable Congestion Control events
  Programmable_Congestion_Control: false
  # Enables application to generate Empty CQE on a given CQ, e.g. for triggering other SW entries
  Generating_Empty_CQE: true
  # Enables application to emulate a PCI device
  Device_Emulation: true
```

7.4.8.5 Error Handling

- The tool checks for duplicate fields, unknown fields, invalid boolean values, and malformed YAML structure.
- Only one document and the specified nested structure are allowed in the YAML.
- Sequences and aliases/anchors are not supported.
- In blob-to-YAML mode, the tool validates that only valid feature bits are set in the binary blob.
- Invalid bits can be treated as errors (default) or warnings (using **--notification**).
- The tool provides detailed error messages showing which specific bits are invalid in the binary blob.

- Command-line validation ensures error modes are only used with `--bin2yaml` and size options are only used with `--yaml2bin`.

7.4.8.6 Binary Blob Format

- The binary blob is stored in big-endian byte order.
- Custom blob sizes can be specified using the `--blob-size` option.
- The blob size must be a multiple of 4 bytes and cannot exceed 1024 bytes.
- Only the bits corresponding to defined features are validated; other bits are checked to ensure they are not set to `1`.

7.5 DOCA Flow Tune Tool

DOCA Flow Tune is a powerful, one-stop-shop solution, providing visibility and analysis capabilities for DOCA Flow programs.

7.5.1 Introduction

> DOCA Flow Tune is supported at alpha level.

DOCA Flow Tune is a one-stop-shop solution which allows developers to visualize their traffic steering pipelines, have a live monitor of software Key Performance Indicators (KPIs) as well as hardware counters, and gain valuable performance insights about their DOCA-Flow-based program.

DOCA Flow Tune is especially useful for the following scenarios:

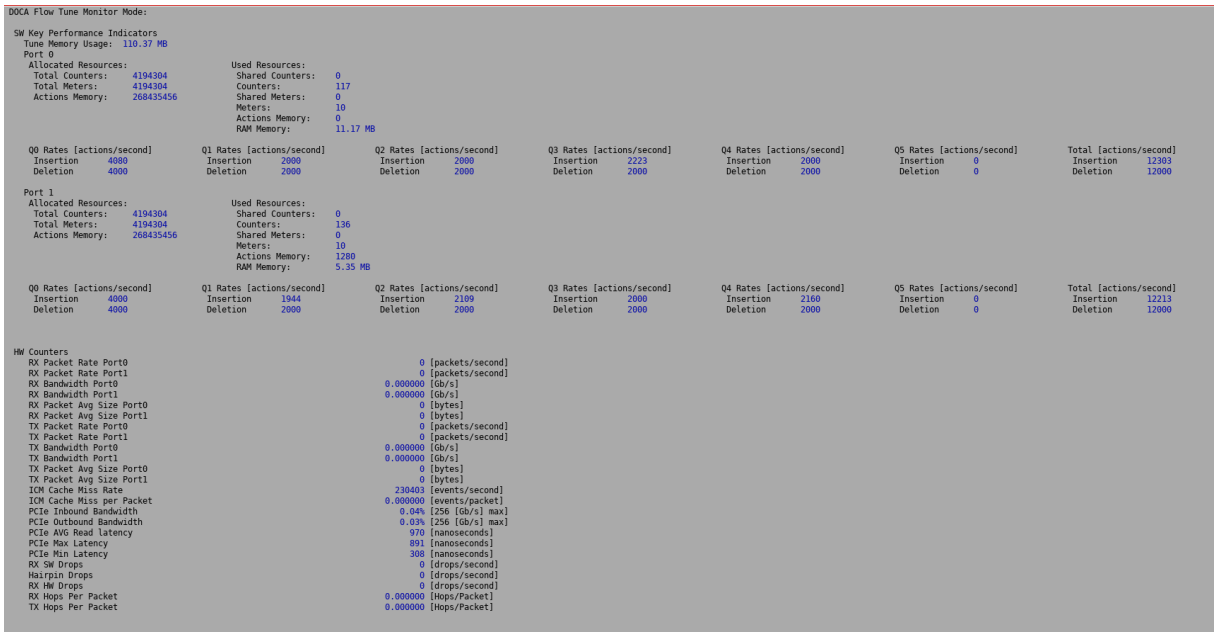
- Aiding developers during the development of their traffic steering pipeline by providing visualization of the pipeline, and later also performance insights about the designed pipeline
- Aiding developers in pre-production environments by providing live monitoring of the performance indicators of the program on both software and hardware levels, and helping detect possible bottlenecks/critical paths so they are addressed before deployment to production environments
- Aiding administrators monitor the program in production by providing live monitoring as well as high-rate hardware counters to be used when analyzing a possible deployment/ setup issue

The tool operates in three distinct modes, Monitor, Analyze, and Visualize, which are presented in the following subsections.

> Collecting, analyzing, and displaying information from the analyzed DOCA-Flow-based program requires the explicit activation of the DOCA Flow Tune server by the target program. For more information about that, refer to the DOCA Flow Tune Server programming guide.

7.5.1.1 Monitor Mode Overview

This mode collects and displays both hardware counters and software KPIs in real time (as extracted from the running DOCA Flow program and the underlying setup), providing a comprehensive view of the system's performance:



This information can also be exported to a CSV/JSON file for further analysis.

For more information about this mode, please refer to section "Monitor Mode".

For information about running DOCA Flow Tune in this mode, please refer to section "Monitor Command".

7.5.1.2 Analyze Mode Overview

The analyze mode supports the ability to dump the internal steering pipeline state to be used by Visualize mode.
It also supports the ability to trace a packet in the pipeline and dump the path in JSON format.

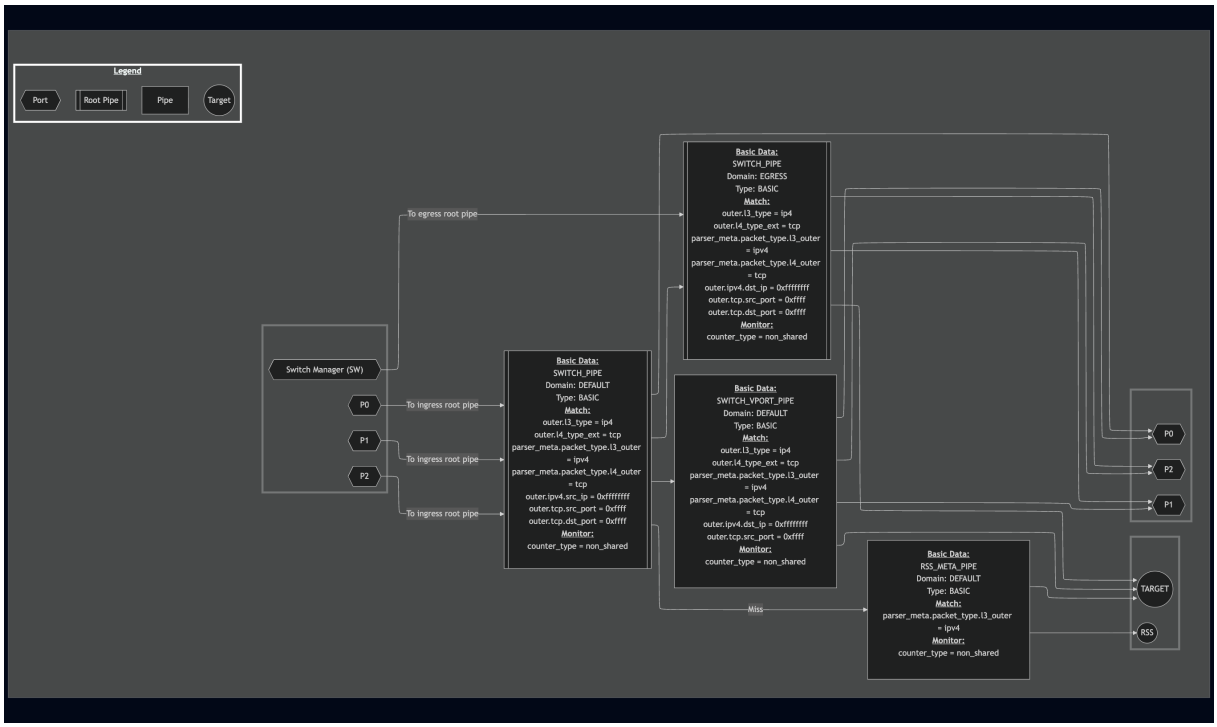
For more information about this mode, please refer to section "Analyze Mode".

For more information about running DOCA Flow Tune in this mode, please refer to section "Analyze Command".

7.5.1.3 Visualize Mode Overview

This mode allows users to produce a graphical representation of their steering pipeline (as built using the DOCA Flow API), allowing developers to quickly understand their program's pipeline, and compare it with their intended architecture.

The following diagram represents DOCA Flow Switch to Wire sample:



7.5.1.3.1 Mermaid Graph

The DOCA Flow Tune Mermaid graph visually represents the pipeline structure and metadata. It uses the following conventions:

- Shapes:
  - Square: Represents a pipe (root pipes denote with two additional lines inside the figure).
  - Hexagon: Represents a port.
  - Circle: Represent a pipeline target (drop, RSS, Kernel, etc).
  - Arrow: Indicates a possible connection between pipes and ports.
- Pipe Data Layers:
  Pipe metadata is organized into multiple visual layers. Layers can be stacked and selectively displayed when analyzing the pipeline.
  - Base Layer (default):
    - Includes general pipe attributes:
      - Name, label, type, domain, port id, hash algorithm
    - Match fields, action data and monitor configuration

        - Mask fields if provided.
    - Cost Layer:
        - Displays the minimum and maximum cost a packet can incur while traversing the pipe.
        - Overlays on top of the base layer.
    - Critical Path Layer:
        - Highlights the pipeline path with the highest total processing cost.
        - If multiple paths have the same cost, a single one is arbitrarily highlighted.
    - Resources Layer:
        - Shows resource utilization per pipe, including:
            - Number of counters
            - Number of meters
            - Action memory usage
            - RAM consumption
        - This layer is an overlay on the base layer.
    - Packet Trace Layer:
        - Given a PCAP packet and port ID, show the packet path in the pipeline in green.

For more information about this mode, refer to section "Visualize Mode".

For more information on running DOCA Flow Tune in this mode, refer to section "Visualize Command".

7.5.2 Dependencies

DOCA Flow Tune depends on the following DOCA SDK libraries:

- DOCA Flow
- DOCA Telemetry

7.5.3 Prerequisites

- DOCA 3.0.0 and higher.
- For optimal experience, it is recommended to comply with the prerequisites of all the listed dependencies, and especially with their recommended firmware versions.

7.5.4 Execution

To execute DOCA Flow Tune tool:

```
Usage: doca_flow_tune [Program Commands] [DOCA Flags] [Program Flags]

Program Commands:
  analyze                         Run Flow Tune in Analyze mode
  monitor                         Run Flow Tune in Monitor mode
  visualize                       Run Flow Tune in Visualize mode

DOCA Flags:
  -h, --help                      Print a help synopsis
  -v, --version                   Print program version information
```

```
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file
```

This usage printout can be printed to the command line interface (CLI) using the **-h** (or **--help**) option:

```
doca_flow_tune -h
```

The same applies for each of the tool's commands (and subcommands). For instance:

```
doca_flow_tune monitor -h
```

7.5.4.1 Monitor Command

The **monitor** command presents software KPIs and hardware counters. Each component offers various options, which can be specified in the configuration file under the **monitor** section, or through the CLI.

```
Usage: doca_flow_tune monitor [Program Commands] [DOCA Flags] [Program Flags]

Program Commands:
  background                        Collect SW key performance indicators and HW counters on the background

DOCA Flags:
  -h, --help                        Print a help synopsis
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma

Program Flags:
  --enable-dump                     Enable counters data dumping to file
  --disable-dump                    Disable counters data dumping to file
  --dump-file-name                  Counters dump file name to create
  --dump-file-format                Counters dump file format {json, csv}
  --hw-profile                      Register hardware profile {basic, full}
  --sw-profile                      Register software profile
  -f, --cfg-file                    JSON configuration file
```

Supported sub-commands:

- **background** – This subcommand allows performing file dumping without displaying the output on the screen. This is useful for scenarios where one wants to log counters without cluttering the terminal. It also supports high-rate dumping for hardware counters which may be activated using the **--high-rate** flag.

```
Usage: doca_flow_tune monitor background [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE,
20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma

Program Flags:
  --high-rate                       Enable dumping HW counters data to file in high rate
  --hw-profile                      Register hardware profile {basic, full}
  --sw-profile                      Register software profile
```

7.5.4.1.1 CLI Examples

- To launch the **monitor** command with a given configuration file:

```
doca_flow_tune monitor -f /tmp/flow_tune_cfg.json
```

- To launch the **monitor** command with both a given configuration file and a CLI parameter for specifying the desired hardware counters profile:

```
doca_flow_tune monitor -f /tmp/flow_tune_cfg.json --hw-profile basic
```

- To launch the **monitor** command with the **background** subcommand and the request to perform a high-rate collection and export for the hardware counters:

```
doca_flow_tune monitor -f /tmp/flow_tune_cfg.json background --high-rate
```

> ⚠ The tool silently creates and updates the **flow_tune{.csv, .json}** file.

7.5.4.2 Analyze Command

The **analyze** command runs a specified set of analysis methods over the target DOCA Flow program. The analysis supports the ability to export a JSON description of the steering pipeline, as is used by the **visualize** command, and could later be used for future analysis methods (both online or offline).
The **analyze** command also supports the ability to trace a packet path in the pipeline and output the path in JSON format.

```
Usage: doca_flow_tune analyze export [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  --file-name                       File name on which the pipeline information will be saved
  -f, --cfg-file                    JSON configuration file

------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------

Usage: doca_flow_tune analyze packet-trace [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma

Program Flags:
  --pcap-file                       Input PCAP file path
  --output-file                     Output file path
  --port-id                         Ingress port ID for packet trace
  -f, --cfg-file                    JSON configuration file
```

Supported subcommands:

- **export** – This command allows the tool to export a running DOCA Flow program's pipeline into a JSON file. This file is the main input for other features of the tool, such as the graphical visualization.
- **packet-trace** - This command allows the tool to trace a given packet in the DOCA Flow program's pipeline, and dump the path in JSON format.

7.5.4.2.1 CLI Examples

7.5.4.2.1.1 Export subcommand

- To launch the `analyze` command without a configuration file:

```
doca_flow_tune analyze export
```

  The JSON file is stored into its default path.
- To launch the **analyze** command with a given configuration file that specifies the desired values for all configurations necessary:

```
doca_flow_tune analyze export -f /tmp/flow_tune_cfg.json
```

- To launch the **analyze** command with a configuration file while also configuring the output path for the exported JSON file through the CLI:

```
doca_flow_tune analyze export -f /tmp/flow_tune_cfg.json --file-name my_program_pipeline_desc.json
```

  The exported pipeline is stored as **my_program_pipeline_desc.json** into the chosen/default output directory.

7.5.4.2.1.2 Packet Trace Subcommand

- To launch the **packet-trace** command without a configuration file:

```
doca_flow_tune analyze packet-trace --pcap-file /path/to/pcap/file --output-file /path/to/output/file
```

- To launch the **packet-trace** command with a given configuration file that specifies the desired values for all necessary configurations:

```
doca_flow_tune analyze packet-trace -f /tmp/flow_tune_cfg.json
```

- To launch the **packet-trace** command with a configuration file while configuring the output path for the packet trace JSON file through the CLI:

```
doca_flow_tune analyze packet-trace -f /tmp/flow_tune_cfg.json --output-file /path/to/output/file
```

7.5.4.3 Visualize Command

The **visualize** command visualizes the steering pipeline of a given DOCA Flow program. The command works on a given JSON file as input. This file can either be generated by the **analyze**

export command or queried dynamically from a running program, in which case the command would dump the pipeline from the program and then generate the visualization output file.

> The visualization output file is a Mermaid markdown format.

> This file can be fed to any of the widely available Mermaid visualization tools, as explained in depth in the corresponding section "Visualize Mode".

```
Usage: doca_flow_tune visualize [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                        Print a help synopsis
  -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL,
30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter                      Filter logs from specific modules, separated by comma
  -j, --json <path>                 Parse command line flags from an input json file

Program Flags:
  --pipeline-desc                   Input JSON file that represents the Flow application pipeline
  --file-name                       File name on which the visualization information will be saved
  -f, --cfg-file                    JSON configuration file
  --cost                            Show cost
  --critical-path                   Show critical path
  --resources                       Show resources
  --distribution                    Show distribution
  --tag                             Add mermaid tag (```mermaid) to the .md file
  --enhanced                        Enable enhanced graph coloring
```

7.5.4.3.1 CLI Examples

- Launching the **visualize** command without a configuration file triggers a live query of the pipeline from the currently running DOCA Flow program. In the absence of an input JSON file, the tool automatically generates and dumps a JSON representation of the active pipeline to be used for Mermaid diagram generation.

  ```
  doca_flow_tune visualize
  ```

  Example output:

  ```
  2025-04-09 15:06:44 - flow_tune - INFO - Pipeline description file not provided or doesn't exist, dumping
  pipeline description to /tmp/tmp_pipeline_desc.json
  2025-04-09 15:06:44 - flow_tune - INFO - Flow program pipeline information was exported to "/tmp/
  tmp_pipeline_desc.json"
  2025-04-09 15:06:44 - flow_tune - INFO - Mermaid graph exported to /tmp/flow_tune/flow_tune_pipeline_vis.md
  2025-04-09 15:06:44 - flow_tune - INFO - DOCA Flow Tune finished successfully
  ```

- Launching the **visualize** command with a given configuration file that specifies the desired values for all needed configurations:

  ```
  doca_flow_tune visualize -f /tmp/flow_tune_cfg.json
  ```

- To launch the **visualize** command with a configuration file while configuring the output path for the Mermaid file through the CLI and providing an offline pipeline file:

  ```
  doca_flow_tune visualize -f /tmp/flow_tune_cfg.json --file-name my_program_pipeline_viz.md --pipeline-desc
  my_program_pipeline_desc.json
  ```

  The exported Mermaid file is stored as **my_program_pipeline_viz.md** into the chosen/default output directory. Because the pipeline description file is explicitly provided, this command

could be used offline, as it would not need connection with the DOCA Flow program to visualize.

> ⚠ To launch the **visualize** command with the **--critical-path** or **--cost** flags, the **low_level_info** setting must be enabled (**true**):
>
> - When using a configuration file, the **low_level_info** field must be set to **true** in the **analyze** section of the provided configuration file. For example:
>
> ```
> doca_flow_tune visualize --cost -f /tmp/flow_tune_cfg.json
> # flow_tune_cfg.json must include: "low_level_info": true
> ```
>
> - When using an offline pipeline JSON file, the file must have been generated with **low_level_info** enabled at the time of the dump:
>
> ```
> doca_flow_tune visualize --cost --pipeline-desc my_program_pipeline_desc.json
> # my_program_pipeline_desc.json must have been dumped with "low_level_info": true
> ```

7.5.5 Configuration

DOCA Flow Tune has a configuration file which allows customizing various settings.

> ℹ The configuration file is divided into sections to simplify its usage.

7.5.5.1 Config File Default Values

If a configuration file is not provided, DOCA Flow Tune uses its default values for fields which are mandatory.

> ℹ A list of all default values can be seen in the appendix.

In Monitor mode, if a software KPIs or hardware counters query is not needed, removing the **hardware** or **software** fields from the configuration file disables the respective feature.

7.5.5.2 Custom Config File

Instead of using default configuration values, users can create a file of their own and provide a file path when running DOCA Flow Tune (**-f** / **--cfg-file**).

Once used, DOCA Flow Tune loads all provided values directly from the file, while the rest of the fields (if any) use their respective default values.

7.5.5.3 Overriding Config Values from CLI

Setting some of the fields in the configuration file is supported through CLI using the **--file-name** flag. If used, the values provided from the CLI would override the values of the fields from

the configuration file. This allows for easier configuration of common values without the need to create a new custom file or to modify an existing configuration file.

7.5.5.4 Common Configuration Values

Some sections of the configuration file are shared between multiple runtime modes of DOCA Flow Tune (i.e., Monitor, Analyze, Visualize) and generally have to do with the output/input file paths and interaction with the live DOCA Flow program.

flow_tune_cfg.json

```
{
...
    "outputs_directory": "/tmp/flow_tune/",
...
    "network": {
        "server_uds": "/tmp/tune_server.sock",
        "uds_directory": "/var/run/doca/flow_tune/"
    },
...
}
```

7.5.5.4.1 Output Directory

outputs_directory defines the main directory on which all output products are saved. This field does not have a default value. If no value is provided, DOCA Flow Tune files are saved at the following directories:

- Counters dump file {.json, .csv} – **/tmp/flow_tune/**
- Analyze export pipeline description file – **/tmp/flow_tune/**
- Pipeline visualization file – **/tmp/flow_tune/**

7.5.5.4.2 Connection to DOCA Flow Tune Server

Some features of DOCA Flow Tune work by interacting with a live DOCA Flow based program. This is enabled through a server that is running in the background as part of the DOCA Flow library, and requires all the following to be applied:

1. DOCA Flow based program should explicitly enable the server.

   More information is available in the relevant DOCA Flow Tune Server programming guide.

2. The DOCA-Flow-based program should run using the trace-enabled DOCA Flow library.

   More information is available in the "Debug and Trace Features" section of the DOCA Flow programming guide.

DOCA Flow Tune should be configured in a way to allow it to connect to the matching server. This can be done by modifying the following variables under the **network** section of the configuration file:

- **server_uds** – DOCA Tune Server Unix Domain Socket (UDS) path. Default value is **/tmp/tune_server.sock**.
- **uds_directory** – Directory on which all local UDSs are created. Default value is **/var/run/doca/flow_tune/**.

7.5.6 Monitor Mode

7.5.6.1 Hardware Counters

This table provides the supported hardware counters and their associated profiles.

Counter Name	Description	Unit	Profile Basic	Profile Full
RX Packet Rate	The number of received packets per second	pkt/sec	✓	✓
RX Bandwidth	The data transfer rate based on the number of packets received per second	Gb/s	✓	✓
RX Packet Average Size	The average size of received data packets	Bytes	✓	✓
TX Packet Rate	The number of packets transmitted per second	pkt/sec	✓	✓
TX Bandwidth	The data transfer rate based on the number of packets transmitted per second	Gb/s	✓	✓
TX Packet Average Size	The average size of transmitted data packets	Bytes	✓	✓
RX SW Drops	The number of packets dropped due to a lack of WQE for the associated QPs/RQs (excluding hairpin QPs/RQs)	drops/sec	✓	✓
Hairpin Drops	The number of packets dropped due to a lack of WQE for the associated hairpin QPs/RQs	drops/sec	✓	✓
RX HW Drops	The number of packets discarded due to no available data or descriptor buffers in the RX buffer	drops/sec	✓	✓
ICM Cache Miss Rate	The rate of data requests that miss in the ICM (interconnect context memory) cache	events/sec	✗	✓
ICM Cache Miss per Packet	The number of data requests that miss per packet	events/pkt	✗	✓
PCIe Inbound Bandwidth	The number of bits received from the PCIe toward the device per second.	Gb/s	✗	✓
PCIe Outbound Bandwidth	The number of bits transmitted from the device toward the PCIe per second	Gb/s	✗	✓
PCIe AVG Read latency	The average PCIe read latency for all read data	nsec	✗	✓
PCIe Max Latency	The maximum latency (in nanoseconds) for a single PCIe read from the device	nsec	✗	✓
PCIe Min Latency	The minimum latency (in nanoseconds) for a single PCIe read from the device	nsec	✗	✓

Counter Name	Description	Unit	Profile Basic	Profile Full
RX Hops Per Packet	The number of hops per packet in RX domain	hops/ pkt	✗	✓
TX Hops Per Packet	The number of hops per packet in TX domain	hops/ pkt	✗	✓

We don't have a way to export this macro.

7.5.6.2 Software Key Performance Indicators

The following table lists the supported software Key Performance Indicators (KPIs) and their associated profiling categories:

Key Performance Indicator	Description	Units	Profile
Insertion rate	Number of successful table entry insertions per queue per second	actions/ sec	`entries_ops_rates`
Deletion rate	Number of successful table entry deletions per queue per second	actions/ sec	`entries_ops_rates`
Action resources	Number of resources utilized by the program for all user-defined actions	N/A	`action_resources`

7.5.6.2.1 Action Resources Counters

The `action_resources` section displays the following per-port counters, including both the total allocated and currently used amounts:

- Resource counters:
  - Number of Shared Counters
  - Number of Counters
  - Number of Shared Meters
  - Number of Meters
  - Actions Memory
- Used memory resources:
  - RAM

7.5.6.3 Configuration

7.5.6.3.1 Counters Dumper

7.5.6.3.1.1 Hardware Counters

Counter Name	Units
Rate of RX packets on port 0	Packets per second
Rate of RX packets on port 1	Packets per second
RX bandwidth on port 0	Gb/s
RX bandwidth on port 1	Gb/s
Average RX packet size on port 0	Bytes
Average RX packet size on port 1	Bytes
Rate of TX packets on port 0	Packets per second
Rate of TX packets on port 1	Packets per second
TX bandwidth on port 0	Gb/s
TX bandwidth on port 1	Gb/s
Average TX packet size on port 0	Bytes
Average TX packet size on port 1	Bytes
ICMC misses rate	Events per second
ICMC misses per packet	Events per packet
The bandwidth of bytes received from PCIe toward the device	Gb/s
The bandwidth of bytes transmitted from the device toward PCIe	Gb/s
The average PCIe read latency	Nanoseconds
The total latency for all PCIe read from the device	Nanoseconds
The total number of PCIe packets	Events
The maximum latency for a single PCIe read from the device	Nanoseconds
The minimum latency for a single PCIe read from the device	Nanoseconds
RX software drops	Drops per second
Hairpin drops	Drops per second
RX hardware drops	Drops per second
RX Hops Per Packet	[Hops/Packet]
TX Hops Per Packet	[Hops/Packet]

7.5.6.3.1.2 Software KPI Counters

Counter Name	Units
Queue <i> Insertion Rate	Deletions per second

Counter Name	Units
Queue <i> Deletion Rate	Insertions per second
Total Insertion Rate	Insertions per second
Total Deletion Rate	Insertions per second
Total Allocated Counters	
Total Used Counters	
Total Used Shared Counters	
Total Allocated Meters	
Total Used Meters	
Total Used Shared Meters	
Total Allocated Actions memory	Bytes
Total Used Actions memory	Bytes
Used RAM Memory	Bytes

7.5.6.3.1.3 CSV Format

The CSV format stores two types of rows, specific to each counter module:

- *Hardware Counter Rows*

Module Name	HW Counter Name	Counter Value	Timestamp
HW	Hairpin drops	8	142623139459
HW	TX Hops Per Packet	197503959728	142623139459

  - Module Name – Hardware module name
  - HW CounterName – Hardware counter name
  - Counter Value – Counter value
  - Timestamp – Hardware timestamp

- *Software KPI Rows*

Module Name	Port ID	SW Counter Name	Counter Value	Timestamp
SW	0	Queue 0 Insertion Rate	34511	1727345744137828
SW	1	Queue 2 Insertion Rate	37050	1727345755137828
SW	0	Allocated Counters	4	1727345755137828
SW	0	Total Allocated Counters	3	1727345755137828

  - Module Name – Software module name
  - Port ID – Software port ID
  - SW KPI Name – KPI name
  - KPI Value – KPI value
  - Timestamp – Software timestamp

7.5.6.3.1.4 JSON Format

The JSON output consists of a root object where each key represents a timestamp. The value associated with each timestamp is a nested JSON object containing the performance counters and metrics recorded at that specific point in time.

JSON output example:

```
{
  "1711065501 67": {
    "Hairpin Drops Rate": 0.0,
    "ICMC Miss Rate": 0.0,
    "ICMC Miss per Packet": 18.944444444444443,
    "PCIe AVG Read Latency": 6028.0,
    "PCIe Inbound BW": 0.0,
    "PCIe Max Read Latency": 396.0,
    "PCIe Min Read Latency": 346.0,
    "PCIe Outbound BW": 0.0,
    "RX Bandwidth Port0": 0.0,
    "RX Bandwidth Port1": 0.0,
    "RX HW Drops Rate": 0.0,
    "RX Hops Per Packet": 0.0,
    "RX Packet Avg Size Port0": 0.0,
    "RX Packet Avg Size Port1": 0.0,
    "RX Packet Rate Port0": 0.0,
    "RX Packet Rate Port1": 0.0,
    "SW Drops Rate": 0.0,
    "TX Bandwidth Port0": 0.0,
    "TX Bandwidth Port1": 0.0,
    "TX Hops Per Packet": 0.0,
    "TX Packet Avg Size Port0": 0.0,
    "TX Packet Avg Size Port1": 0.0,
    "TX Packet Rate Port0": 0.0,
    "TX Packet Rate Port1": 0.0
  },
  ...
}
```

7.5.6.3.2 Configuration File

DOCA Flow Tune's configuration file consists of two main parts of relevance for Monitor mode:

- **dumper** dump object
- **monitor** configuration object

The following is an example for both sections:

flow_tune_cfg.json

```
{
...
     "dumper": {
        "enable": true,
        "high-rate": false,
        "file_name": "flow_tune.json",
        "stats_time": 1,
        "format_type": "json", // types: {csv, json}
        "max_files": 3,
        "max_size_bytes": 1000000
    },

...

     "monitor": {
        "screen_mode": "dark",  // modes: {light, dark}
        "hardware": {
            "pci_addresses": [
                "b1:00.0",
                "b1:00.1"
            ],
            "profile": "full"  // profiles: {basic, full}
        },
        "software": [
            {
                "flow_port_id": 0,
                "profiles": [
                    "entries_ops_rates"  // profiles: {entries_ops_rates, action_resources}
                ]
            },
            {
```

```
                "flow_port_id": 1,
                "profiles": [
                    "entries_ops_rates"
                ]
            }
        ]
    }
...
}
```

7.5.6.3.2.1 Dumper Configuration Section

The dumper allows exporting the hardware and software counters collected by the tool into a output file for further analysis or record keeping. This is particularly useful for logging performance metrics over time.

How to Enable Counters Dumper

To enable CSV dumping, modify the configuration in the JSON file as follows:

```
{
  "dumper": {
        "enable": true,
        "file_name": "flow_tune.csv",
        "format_type": "csv",
    },
}
```

To enable JSON dumping, modify the configuration in the JSON file as follows:

```
{
  "dumper": {
        "enable": true,
        "file_name": "flow_tune.json",
        "format_type": "json",
    },
}
```

The supported fields are:

- **enable** – Set to **true** to enable counters dumping or **false** to disable it. Default value is **false**.
- **file_name** – The name of the dump file where the data will be saved. By default when enabling the dumper, format type is **Json**.
- **format_type** – The dump file format **CSV** or **Json**.
- **max_size_bytes** – The maximum size (in bytes) of the dump file. Once this limit is reached, a new file is created based on the **max_files** setting.
- **max_files** – The maximum number of CSV files to keep. When this limit is reached, the oldest files are deleted.

The counters dumping can also be enabled or disabled from the CLI using the **--enable-dump** or **--disable-dump** flags, respectively. For example:

```
doca_flow_tune monitor -f /tmp/flow_tune_cfg.json --enable-dump
```

Additionally, the format type and filename can be updated by using the **--dump-file-name** and **--dump-file-format** flags, for example:

```
doca_flow_tune monitor -f /tmp/flow_tune_cfg.json --dump-file-name "counters_dump.csv" --dump-file-format "csv"
```

7.5.6.3.2.2 Monitor Configuration Section

Screen Mode

The Monitor module supports two screen modes: **dark** and **light**.

Hardware

The **hardware** section includes the **pci_addresses** and **profile** fields:

- The **pci_addresses** field expects an array of PCIe addresses for NIC ports. The tool uses these addresses to retrieve the corresponding NIC device and the desired port IDs.

  > ⚠ PCIe addresses must belong to the same device.

  > ℹ The tool supports up to two ports per device.

- The **profile** field expects to receive either a **basic** or **full** profile.
  - **basic** profile – includes packet- and port-related counters (i.e., Bandwidth, Packets Per Second, Average Packet Size, Packet Drops)
  - **full** profile – includes all the **basic** counters and adds additional debug counters (e.g., ICMC and PCIe counters)

    > ℹ For more information about the counters please refer to section "Hardware Counters".

The hardware counters profile can be set from the CLI by adding **--hw-profile**. For example:

```
doca_flow_tune monitor -f /tmp/flow_tune_cfg.json --hw-profile basic
```

Software

The **software** section includes the **flow_port_id** and **profiles** fields:

- **flow_port_id** field – expects a single DOCA Flow port identification number. Flow port ID should be set by the DOCA Flow program, by calling the **doca_flow_port_cfg_set_devargs()** API call with a proper port ID string.
- **profiles** field – accepts one or more of the supported profiling categories.
  - **entries_ops_rates** profile – includes both insertion and deletion rates KPIs
  - **action_resources** profile – provides data on allocated and in-use resources, including counters, meters, and actions memory.

The software profile can be set from the CLI by adding **--sw-profile**, for example:

```
doca_flow_tune monitor -f /tmp/flow_tune_cfg.json --sw-profile entries_ops_rates
```

7.5.7 Analyze Mode

Analyze mode gathers (and later analyzes) information in order to assist users to better understand and debug their DOCA-Flow-based program.

7.5.7.1 Pipeline Export

This command export an internal state of the DOCA-Flow-based program in a proprietary JSON format. This allows the tool to provide offline information about a given program which can be later be analyzed. One such example is the ability to visualize the pipeline of the target program without having said program run on real hardware.

While the pipeline export operation is meant to encode all relevant information for future analysis, the format itself is proprietary and is only meant to be consumed by other DOCA tools.

7.5.7.2 Packet Trace

This command parses a PCAP packet from a PCAP file, and simulates the packet's path in the DOCA Flow based-program pipeline and outputs the path in JSON format. This command is useful for debugging and discovering where do packets go in complex pipelines.

> ⚠ This sub-command does not support all pipe and action types. Consequently, results may be inaccurate for pipelines containing unsupported types.

7.5.7.2.1 Requirements:

- A PCAP file must be provided via the **`--pcap-file`** flag or in the configuration file. Note that if the PCAP file includes more than one packet, only the first packet will be traced.
- A port ID must be provided via the **`--port_id`** flag or in the configuration file. If no port ID was provided then the default is zero
- An output file path must be provided via the **`--output-file`** flag or in the configuration file.

7.5.7.2.2 Output data:

- A JSON file detailing the path that the packet took, including the match and action values.

7.5.7.2.3 Example:

Here we use the DOCA Flow Port FWD Sample, using a PCAP file representing this packet that should match the PORT_FWD_PIPE in the sample:

```
Ether()/IP(src='1.2.3.4', dst='8.8.8.8')/TCP(sport=1234, dport=80)/Raw(load=b'Sent to port 0, forwarded from port 1')
```

And then using the analyze command with the --packet-trace sub-command:

```
doca_flow_tune analyze packet-trace --port-id 0 --pcap-file flow_port_fwd.pcap --output-file /tmp/flow_port_fwd_trace.json
```

We would get this JSON file (at **/tmp/flow_port_fwd_trace.json**):


```
{
    "paths" :
    [
        [
            {
                "entry_id" : 0,
                "match" :
                {
                    "fields" :
                    [
                        "match.packet.outer.l3_type",
                        "match.packet.outer.l4_type_ext",
                        "match.packet.parser_meta.packet_type.l3_outer",
                        "match.packet.parser_meta.packet_type.l4_outer",
                        "match.packet.outer.ipv4.src_ip",
                        "match.packet.outer.ipv4.dst_ip",
                        "match.packet.outer.tcp.src_port",
                        "match.packet.outer.tcp.dst_port"
                    ],
                    "values" :
                    [
                        "DOCA_FLOW_L3_TYPE_IP4",
                        "DOCA_FLOW_L4_TYPE_EXT_TCP",
                        "DOCA_FLOW_L3_META_IPV4",
                        "DOCA_FLOW_L4_META_TCP",
                        "1.2.3.4",
                        "8.8.8.8",
                        "1234",
                        "80"
                    ]
                },
                "name" : "PORT_FWD_PIPE",
                "result" : "hit",
                "type" : "pipe"
            },
            {
                "name" : "port 1",
                "type" : "port"
            }
        ]
    ]
}
```


This JSON describes that there is only one path (because there is only one element in the "paths" array), and the path includes:

1. PORT_FWD_PIPE
    a. The packet matched on the pipe ("result": "hit").
        i. The entry that matched was the first entry ("entry_id": 0).
        ii. Includes the details on the fields that matched and their values.
2. Port 1

7.5.7.2.4 Know Issues:

- Actions of the following types are not supported:
    - Crypto/Crypto Encap
    - NAT64
    - Push VLAN
- Pipes of the following types are not supported:
    - LPM
    - CT
    - ACL

7.5.8 Visualize Mode

The **doca_flow_tune visualize** command generates a visual representation of a DOCA Flow pipeline using the Mermaid diagram format.

7.5.8.1 Supported Pipe Types

The following pipe types are supported in visualization mode:

- Basic
- Control
- Hash
- CT – Fully supported in the Base and Resources layers. For other layers, the visualization results may be inaccurate.
- Ordered List – Fully supported in the Base and Resources layers. For other layers, the visualization results may be inaccurate.

7.5.8.2 Viewing the Pipeline

Running the **visualize** command produces a Mermaid-format file representing the pipeline structure. Mermaid is a text-based diagramming language supported by many documentation tools and editors. One commonly used online editor is Mermaid Live, which can be used to render and inspect the generated pipeline diagram.

To view the pipeline:

1. Open Mermaid Live.
2. Copy the contents of the generated **.mmd** file.
3. Paste the contents into the editor to visualize the pipeline.

7.5.8.3 Visualization Layers

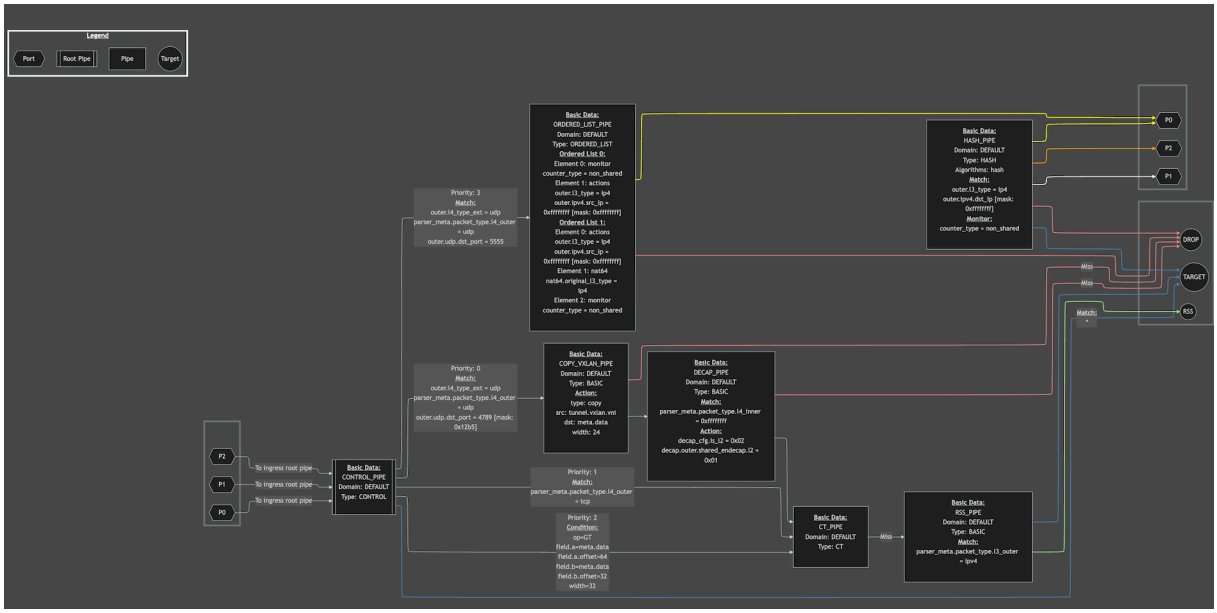
Each Mermaid diagram can include multiple layers, each offering a distinct view into pipeline characteristics. If no specific layer is selected, the base layer is used by default.

7.5.8.3.1 Base Layer

This layer visualizes the static configuration of the DOCA Flow pipeline, including:

- Pipe node data:
  - Name
  - Label
  - Port ID
  - Domain
  - Type
  - Mask configuration
  - Defined actions
  - Monitor attributes
- Pipe link (arrow) data:
  - For Control pipes: Priority, condition and matching values (e.g., ***** indicates match-all).
  - For other pipes: Miss links are marked accordingly.

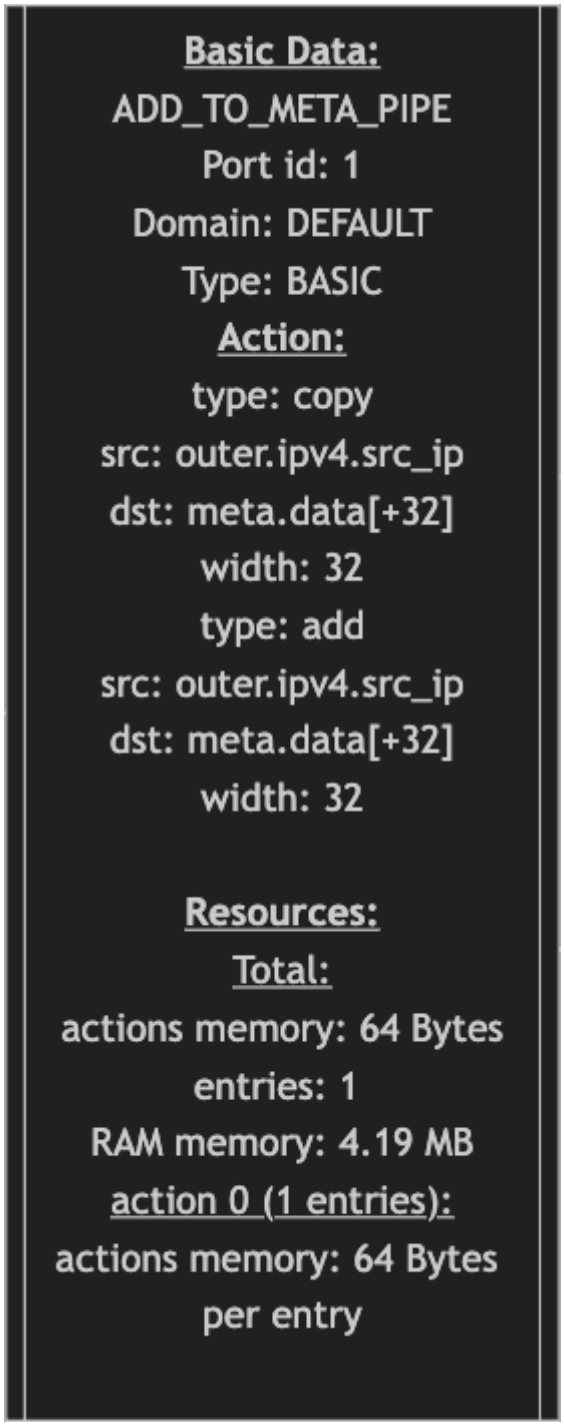
Using **--enhanced**, pipe links going into DROP/TARGET/RSS/PORT will be colored to ease the paths identification.



7.5.8.3.2 Resources Layer

Enabled via the `--resources` flag, this layer adds information about resource utilization:

- Pipe node data:
  - Total number of actions in use by entries (all pipe types)
  - Resource breakdown per action (non-Control pipes only)
- Resources visualized:
  - Counters
  - Meters
  - Actions memory
  - Entry count
  - RAM usage

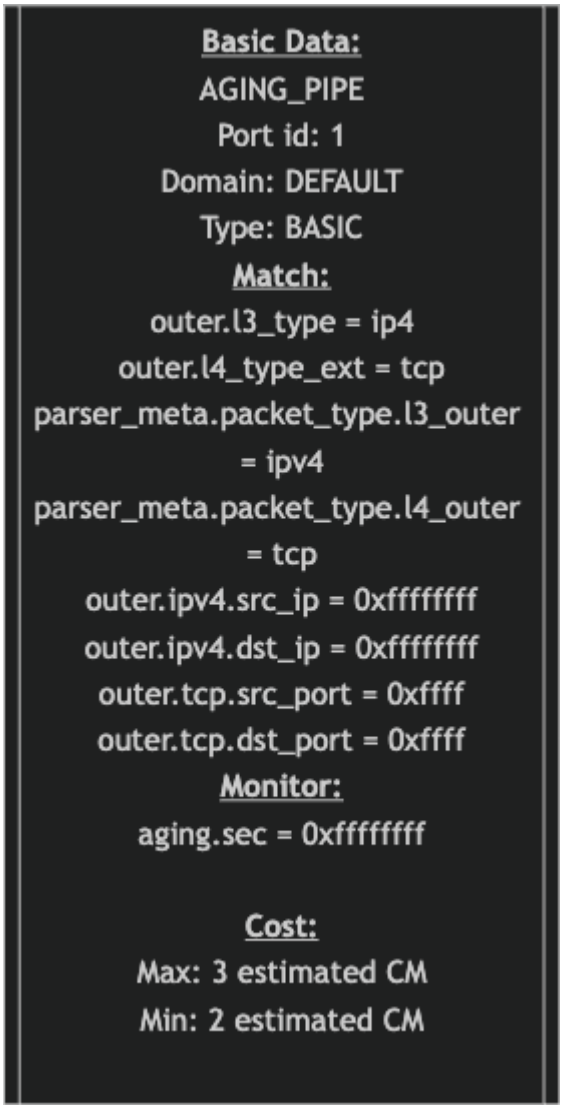


7.5.8.3.3 Cost Layer

Enabled via the `--cost` flag, this layer estimates the cache impact of processing packets through the pipeline:

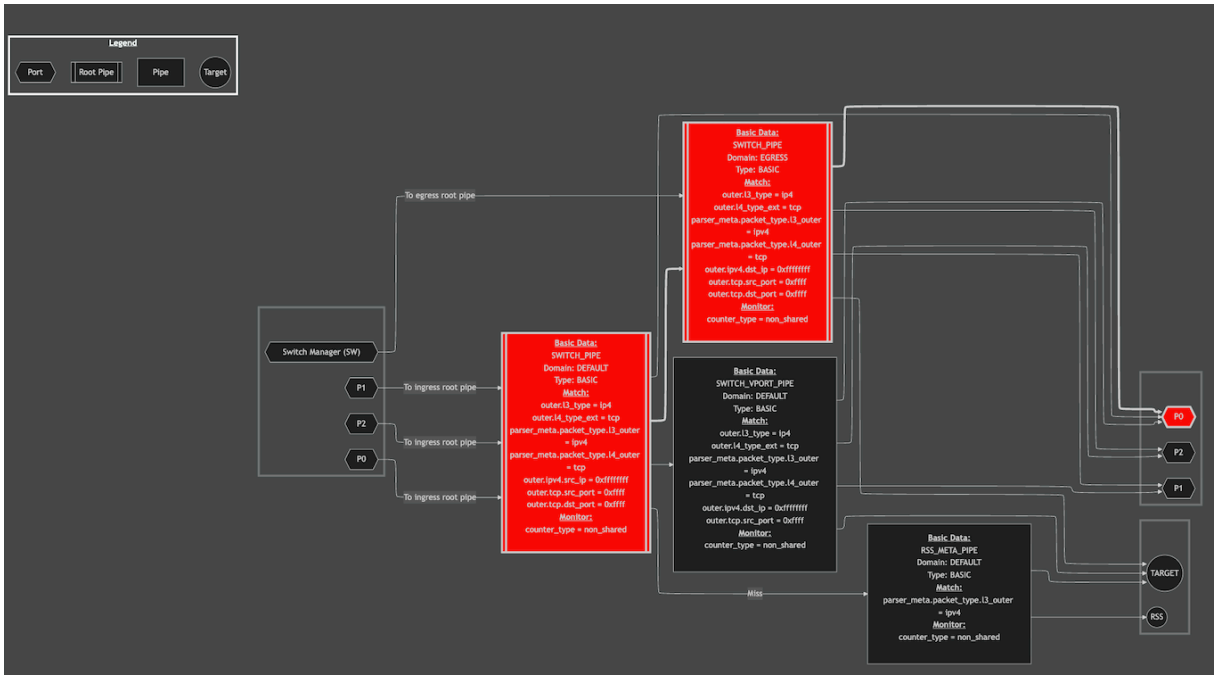
- Minimum and maximum estimated cache misses per pipe

- Helps identify high-cost areas in the pipeline



7.5.8.3.4 Critical Path Layer

Enabled via the `--critical-path` flag, this layer highlights the pipeline path with the highest estimated processing cost (i.e., the path with the most cache misses). If multiple paths are tied, one is selected arbitrarily.



7.5.8.3.5 Distribution Layer

The Distribution Layer is not supported in DOCA version 3.4. Refer to DOCA 3.3 for supported functionality.

Enabled via the **`--distribution`** flag, this layer simulates traffic flow through the pipeline using a PCAP input file.

This layer does not support all pipe types. Consequently, results may be inaccurate for pipelines containing unsupported types.

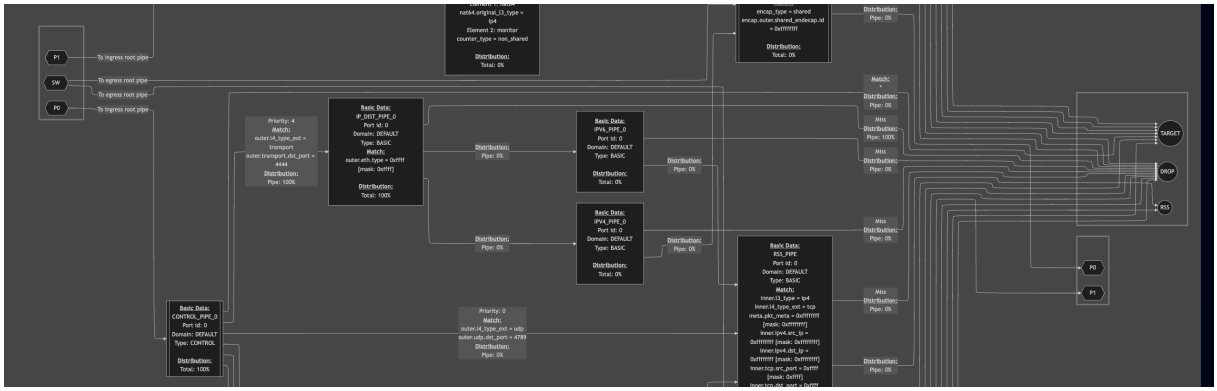
Requirements:

- A PCAP file must be provided via the **`--pcap-file`** flag or in the configuration file.
- The **`low_level_info`** parameter must be set to **`true`** in the configuration when performing offline analysis.

Output data:

- Pipe node data:
  - **`Distribution`** : Displays the percentage of *total* packets that passed through this specific pipe.
    - Example: **`Total: 62%`**
- Pipe link data:

- **Distribution** : Displays the percentage of packets from the *source* pipe that traversed this specific link.
    - Example: **Pipe: 78%**



In the partial diagram above, packets can be observed flowing along the following path: Port 0 → **CONTROL_PIPE_0** → **IP_DIST_PIPE_0** → DROP.

7.5.8.3.5.1 Known Issues

- Match values for the following fields are not supported:
    - **outer.l2_type**
    - **outer_l3_type**
    - **outer_l4_type**
    - **tun.gre.protocol**
    - **tun.key_present**
- The Distribution layer functions only when using supported match fields. Unsupported fields may result in inaccurate or incomplete packet flow analysis.

7.5.8.3.6 Packet Trace Level

Enabled via the **--packet-trace** flag, this layer simulates a packet's path through the pipeline.

> ⚠ This layer does not support all pipe and action types. Consequently, results may be inaccurate for pipelines containing unsupported types.

Requirements:

- A PCAP file must be provided via the **--pcap-file** flag or in the configuration file. *(Note: If the PCAP file includes more than one packet, only the first packet will be traced).*
- A port ID must be provided via the **--port-id** flag or in the configuration file. If no port ID is provided, the default is **0** .

Output data:

- The path (nodes and links) that the packet traversed will be colored green.

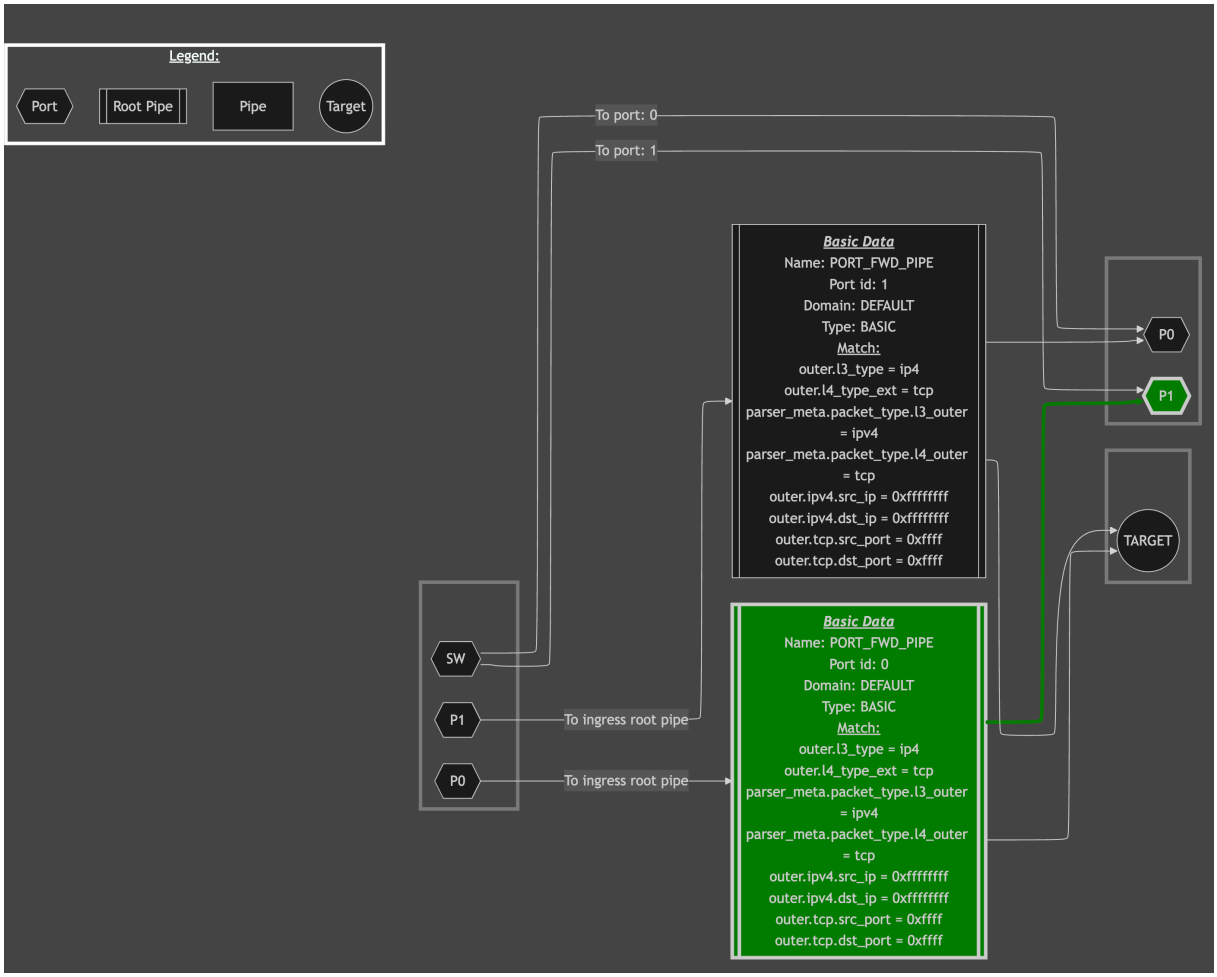
In this example, the DOCA Flow Port FWD Sample is used alongside a PCAP file representing a packet that should match the **PORT_FWD_PIPE**:

```
Ether()/IP(src='1.2.3.4', dst='8.8.8.8')/TCP(sport=1234, dport=80)/Raw(load=b'Sent to port 0, forwarded from port 1')
```

Use the **visualize** command with the **--packet-trace** flag:

```
doca_flow_tune visualize --packet-trace --port-id 0 --pcap-file flow_port_fwd.pcap
```

This generates the following Mermaid diagram:



7.5.8.3.6.1 Known Issues

- Actions of the following types are not supported:
  - Crypto/Crypto Encap
  - NAT64

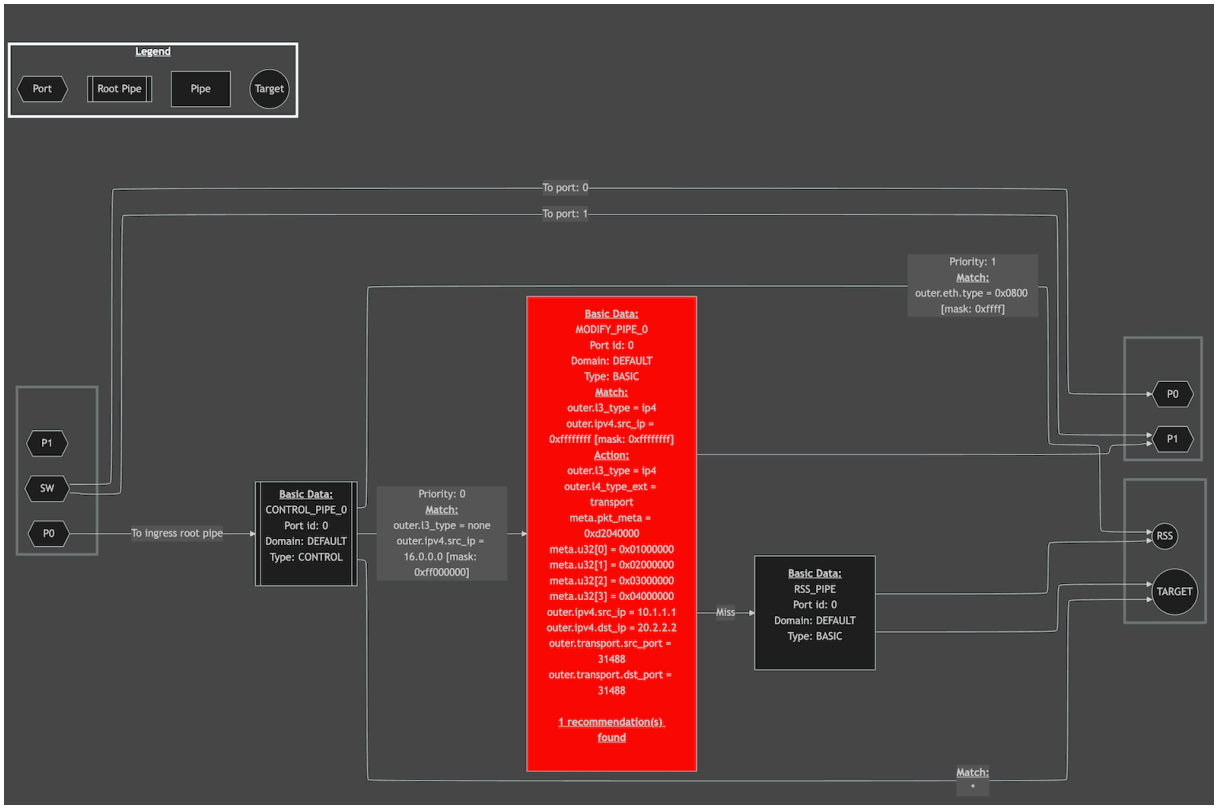
    - Push VLAN
- Pipes of the following types are not supported:
    - LPM
    - CT
    - ACL
- The Packet Trace layer functions properly only when using supported actions. Unsupported actions may result in inaccurate or incomplete packet flow analysis.

7.5.8.3.7 Recommendations Layer

When enabled using the `--recommendations` flag, this layer identifies and highlights pipeline components that can be optimized for better performance.

Supported insights:

- Action and action description split recommendations. This insight enables the tune tool to suggest splitting pipes that contain actions which can benefit from separation. By dividing these pipes into two, the system optimizes the steering processing path, improving overall performance.



7.5.9 Use Cases

7.5.9.1 Actions Memory Calculation

The DOCA Flow API requires the application to configure the maximum actions memory using the `doca_flow_port_cfg_set_actions_mem_size()` function. There are two strategies for determining the optimal value:

7.5.9.1.1 Measured Strategy

- Configure the application with the maximum supported actions memory.
- Run the application under expected load.
- Execute the Flow Tune tool in Monitor mode.
- Use the reported memory usage from the tool to determine the optimal actions memory size.
- Update the application configuration to match the measured usage, ensuring compliance with the DOCA Flow API.

7.5.9.1.2 Calculated Strategy

- Start with a minimal actions memory size in the application configuration.
- After pipe creation (but before adding entries), run the Flow Tune tool in Visualization mode with the Resources Layer enabled.
- For each pipe:
  - Multiply the expected number of entries by the reported actions memory per entry.
- Sum the memory requirements across all pipes to obtain the total actions memory needed.
- Set this value in the application using the appropriate DOCA Flow API.

7.5.9.2 Counter Observation

Use the tool in Monitor mode to observe hardware and software counters in real time. The tool provides per-second updates on:

- Entry insertion and deletion rates
- Memory and resource utilization
- Other performance metrics

This helps identify performance bottlenecks and track system behavior under load.

7.5.9.3 Program Pipeline Visualization

By running the tool in Visualization mode, the developer can:

- Generate a graphical representation of the DOCA Flow program pipeline
- Understand the packet processing path and logic
- Debug and validate flow construction
- Optimize the pipeline for performance and correctness

7.5.9.4 Identifying Bottle Neck Pipes

Running the tool in Visualization mode with the `--cost` and `--critical-path` flags enables detection of performance-critical paths:

- Highlights pipes with the highest cache miss estimates
- Marks the most critical path (i.e., the slowest or most resource-intensive)
- Helps developers focus optimization efforts on the slowest segments of the pipeline

7.5.10 Troubleshooting

7.5.10.1 Telemetry fwctl driver is not loaded

7.5.10.1.1 Error

When running the DOCA Flow Tune in Monitor mode, the following log messages are encountered at startup:

```
[DOCA][WRN][priv_doca_telemetry_fwctl.cpp:121][priv_doca_telemetry_fwctl_find_device_by_pci] Failed finding fwctl
device: Opening directory /sys/class/fwctl/ failed. Make sure you have the fwctl driver loaded
[DOCA][ERR][priv_doca_telemetry_fwctl.cpp:201][priv_doca_telemetry_fwctl_open_by_devinfo] devinfo 0x55c572286520:
Failed to open fwctl device: Failed to find matching fwctl device
```

7.5.10.1.2 Solution

The DOCA Telemetry SDK uses the `fwctl` driver to query the hardware counters, so it is essential to have it installed and loaded.

7.5.10.1.2.1 Step 1: Verify the Driver Installation

First, check if the driver is installed as follow:

- Debian/Ubuntu:

  ```
  $ sudo apt list --installed | grep fwctl
  ```

- RHEL:

  ```
  $ sudo yum list installed | grep fwctl
  ```

If the driver is not installed, install it by running the following commands:

- Debian/Ubuntu:

```
$ sudo apt search fwctl
>> <fwctl-package-name>/....

$ sudo apt install -y <fwctl-package-name>
```

- RHEL:

```
$ sudo yum search fwctl
>> <fwctl-package-name>/....

$ apt/yum install -y <fwctl-package-name>
```

7.5.10.1.2.2 Step 2: Check if the Driver is Loaded

After installing the driver, verify that it is loaded by executing:

```
$ sudo lsmod | grep fwctl
```

You should see output similar to:

```
> mlx5_fwctl 20480 0
> fwctl 16384 1 mlx5_fwctl
> mlx5_core 2134016 2 mlx5_fwctl,mlx5_ib
> mlx_compat 69632 14
 rdma_cm,ib_ipoib,mlxdevm,mlxfw,mlx5_fwctl,iw_cm,ib_umad,fwctl,ib_core,rdma_ucm,ib_uverbs,mlx5_ib,ib_cm,mlx5_core
```

If the driver is not loaded, load it by running:

```
$ sudo modprobe mlx5_fwctl
```

7.5.10.2 Mermaid visualization in Visual Studio Code

Visual Studio Code provides extensions to view Mermaid markdown format, these extensions can be used to view the Mermaid output from DOCA Flow Tune tool.

However, for these extension to work, DOCA Flow Tune tool should run with `--tag`.

7.5.10.3 Limited Feature Set – Could Not Detect Running DOCA Flow Program

7.5.10.3.1 Error

When running DOCA Flow Tune, the following log message is encountered at startup, followed by some features failing to work/load:

```
[DOCA][WRN][flow_tune.cpp:195][get_flow_app_data] Could not detect a running DOCA Flow program, some features will
be impacted
```

7.5.10.3.2 Solution

Some features of DOCA Flow Tune work by interacting with a live DOCA-Flow-based program. This is enabled through a server running in the background as part of the DOCA Flow library, and requires all of the following to be applied:

- DOCA-Flow-based program should explicitly enable the server. More information is available in the DOCA Flow Tune Server programming guide.
- DOCA-Flow-based program should run using the "trace enabled" DOCA Flow library. More information is available in the "Debug and Trace Features" section of the DOCA Flow programming guide.\

7.5.10.4 visualize --cost or --critical-path Not Working

7.5.10.4.1 Error

When running DOCA Flow Tune in **visualize** mode with the **--cost** or **--critical-path** flags, the following error may occur:

```
2025-04-14 12:24:20 - flow_tune - ERROR - Low level info need to be enabled to visualize cost or critical path
2025-04-14 12:24:20 - flow_tune - ERROR - Failed to export mermaid graph
2025-04-14 12:24:20 - flow_tune - ERROR - DOCA Flow Tune failed during "visualize" command: Invalid input
2025-04-14 12:24:20 - flow_tune - ERROR - DOCA Flow Tune finished with errors
```

7.5.10.4.2 Solution

The **--cost** and **--critical-path** options require low-level pipeline information, which is not enabled by default.

To resolve this issue, ensure that **"low_level_info": true** is specified in one of the following ways:

1. Option 1 – configuration file:

   ```
   doca_flow_tune visualize --cost -f /tmp/flow_tune_cfg.json
   ```

   Make sure the config file (**flow_tune_cfg.json**) includes:

   ```
   "analyze": {
     "low_level_info": true
   }
   ```

2. Option 2 – offline pipeline description:

   ```
   doca_flow_tune visualize --cost --pipeline-desc my_program_pipeline_desc.json
   ```

   Ensure that **my_program_pipeline_desc.json** was originally exported with **"low_level_info": true** enabled during analysis or dump.

7.5.11 Appendix – Configuration File Example

flow_tune_cfg.json

```
{
    "outputs_directory": "/tmp/flow_tune/",
    "logging": {
        "developer_log": "/var/log/doca/flow_tune/flow_tune_dev.log",
        "operational_log": "/var/log/doca/flow_tune/flow_tune.log"
    },
    "network": {
        "server_uds": "/tmp/tune_server.sock",
        "uds_directory": "/var/run/doca/flow_tune/"
    },
    "dumper": {
        "enable": false,
        "file_name": "flow_tune.csv",
        "format_type": "csv",
        "max_size_bytes": 1000000000,
        "max_files": 1
    },
    "analyze": {
        "file_name": "flow_tune_pipeline_desc.json",
        "low_level_info": false
    },
    "visualize": {
        "pipeline_desc_file": "/tmp/flow_tune/flow_tune_pipeline_desc.json",  // Non-mandatory field
        "file_name": "flow_tune_pipeline_vis.md",
        "pcap_file": "sample_pcap.pcap"
    },
    "monitor": {
        "screen_mode": "light",
        "hardware": {
            "pci_addresses": [
                "08:00.0",
                "08:00.1"
            ],
            "profile": "full"
        },
        "software": [
            {
                "flow_port_id": 0,
                "profiles": [
                    "entries_ops_rates",
                    "action_resources"
                ]
            },
            {
                "flow_port_id": 1,
                "profiles": [
                    "entries_ops_rates",
                    "action_resources"
                ]
            }
        ]
    }
}
```

Where:

- **outputs_directory** – Main directory on which all output products are saved. This field does not have a default value. If no value is provided, DOCA Flow Tune files are saved at the following directories:
  - Dumper file – **/tmp/flow_tune/**
  - Analyze export pipeline description file – **/tmp/flow_tune/**
  - Pipeline visualization file – **/tmp/flow_tune/**
- **logging** - Log files directory.
  - **developer_log** – developer level log file.
  - **operational_log** – user level log file.
- **network**
  - **server_uds** – DOCA Tune Server Unix Domain Socket (UDS) path. Default value is **/tmp/tune_server.sock**.

    - **uds_directory** – Directory on which all local UDS is created. Default value is **/var/run/doca/flow_tune/**.
- **dumper**
    - **enable** – true if information should be saved into a dump file. Default value is **false**.
    - **file_name** – Dump filename. Default value is **flow_tune.json**.
    - **format_type** – The dump file format **CSV** or **Json**. Default value is **Json**.
    - **max_size_bytes** – The file maximum size in bytes. When the limit is reached, a new file is created. Default value is **1Gb**..
    - **max_files** – Maximum CSV files to create. Default value is **1**.
- **analyze**
    - **file_name** – Flow program pipeline description filename. File is created under **outputs_directory** path. Default value is **flow_tune_pipeline_desc.json**.
    - **low_level_info** – Pipeline description JSON file to include low level information.
- **visualize**
    - **pipeline_desc_file** – Flow program pipeline description input file path. This file is the product of the **analyze export** command. If not provided, then the **visualize** command triggers a creation of it (a run of **analyze export**).
    - **file_name** – Flow program pipeline visualization filename. File is created under the **outputs_directory** path. Default value is **flow_tune_pipeline_vis.md**.
    - **pcap_file** – PCAP file to use when running distribution algorithm.
- **monitor**
    - **screen_mode** – Monitor command theme to be used. Default value is **light**.
    - **hardware**
        - **pci_addresses** – List of PCIe addresses which DOCA Flow Tune should inspect.
        - **profile** – Hardware profile to be used for each PCIe address given. Default value is **full**.
    - **software**
        - **flow_port_id** – Flow program port identification number which DOCA Flow Tune should inspect.
        - **profiles** – List of software profiles to be used for the specific port identification number given. Default value is **[entries_ops_rates]**.

7.6 DOCA Ngauge

This document provides instructions on the usage of the **ngauge** tool.

7.6.1 Introduction

ngauge tool is used to analyze, visualize, and debug network performance on a single node. The tool is designed for probing NIC hardware counters and storing the collected data in HDF5 format, along with relevant metadata, for subsequent processing. Additionally, the tool provides graphical progress updates and a measurement summary directly on the CLI, offering real-time insights into the measurement process.

Supported hardware: NVIDIA® BlueField®-3, NVIDIA® ConnectX®-7, and above.

ngauge relies on the **fwctl** driver and, therefore, cannot be run simultaneously with other tools or services that also utilize this driver.

7.6.2 Prerequisites

- BlueField-3 or ConnectX-7 and above with firmware version xx.43.1000 or higher
- **fwctl** driver installed on the host:

  On hosts running Linux kernel version 6.15 and above, there is no need to install **fwctl** manually as it is available by default, unless it was explicitly disabled during kernel configuration.

OS	Commands

Deb-based	a. Search for the package:
	`apt-cache search fwctl`
	b. Install the package:
	`sudo apt install <package-name>`

RPM-based	a. Search for the package:
	`dnf search fwctl`
	b. Install the package:
	`sudo dnf install <package-name>`

We don't have a way to export this macro.

7.6.2.1 Installing Ngauge

Install **ngauge** by running **sudo apt-get install ngauge** or **sudo dnf install ngauge** (on x86 or Arm 64 hosts).

On the DPU, the **ngauge** package is pre-installed, so the above step is not necessary.

7.6.3 Description

All the configurations for **ngauge** are defined in an input YAML file.

1. Copy a sample configuration file from **/usr/share/doc/ngauge/examples/settings**.

   > Use the one which best fits your scenario single/dual port, etc.

2. Specify the device to run on using its PCIe address. For example:

```
device: "0000:03:00.0"
```

3. Configure the output path and file prefix (both are mandatory):

```
output:
  path: /path/to/output/directory
  prefix: "ngauge_data_"
  silent: false
```

   - The output file is saved in the format **/path/to/output/directory/<prefix>_<DATE>_<TIME>.h5**.
   - The exact file name is printed after each run.
   - If the **silent** option is set to **true**, progress indications on the command line are suppressed (default: **false**).

4. Configure parameters for the application's runtime behavior:

```
params:
  mode: repetitive  # [repetitive, single]
  period_us: 1e2    # Sampling period in microseconds (e.g., "1e2" = 100 µs)
```

   > Numbers in decimal or scientific notation are accepted. In the example, **1e2** means 100 µs.

5. Define the counters to measure. The **id** (data ID) is the only mandatory field. Additional fields are optional:

```
counters:
  - id: 0x1020000100000000  # Data ID (mandatory)
    desc: RX bytes port 0   # Description (optional)
    unit: RX port           # Unit type (optional)
    accumulating: false     # Whether the counter accumulates values (optional)
    normalizer: time        # Normalizer, if present, must be either 'time', or a non-zero number, or one
of the configured Data IDs with an 'id/' prefix (optional)
    cutoff_min: 1           # Cutoff minimum: data below this value will not be recorded (optional)
    cutoff_max: 3e10        # Cutoff maximum: data above this value will not be recorded (optional)
```

   > All supported performance counters may be found under section "Supported Data IDs".

7.6.3.1 Parsing Output

A sample plugin named **simple-plot** is provided and installed under **/usr/share/doc/ngauge/examples/plugins**.

This plugin demonstrates how to open the output HDF5 file generated by **ngauge** and plot the data. While it focuses on plotting, the data can also be used for various types of analysis. This plugin is a basic demonstration and is not intended for advanced use.

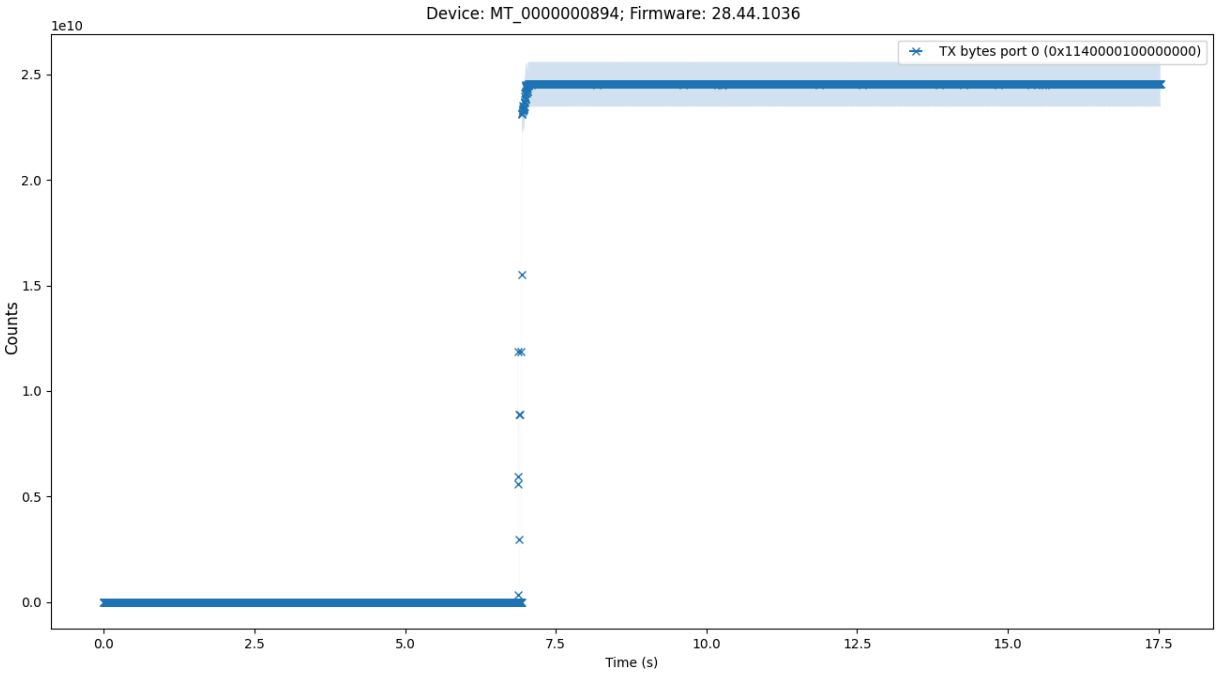
To plot the data from an **ngauge** output file, use the following command:

```
/usr/share/doc/ngauge/examples/plugins/simple_plot.py <ngauge output .h5 file> <counter ID> [<counter ID> ...]
```

Use the **-h** flag to view additional options, including plotting on a logarithmic scale and in the frequency domain.

If your output directory is **/tmp** (the default), you can always reference the most recent results without manually copying the file name by using the expression **"$(ls -1 /tmp/ngauge_data_*.h5 | tail -n1)"** .

The following is a simple plot example. The lighter area around the samples represents the worst-case measurement error.



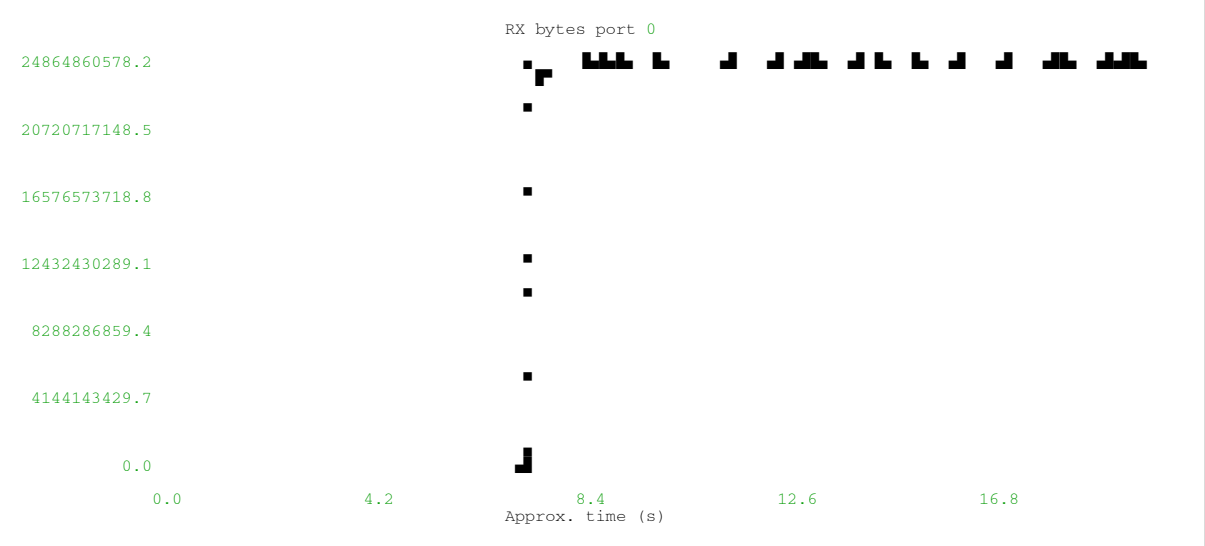
The GUI includes a zoom-in function that allows you to drill down for higher resolution.

An alternative plugin, **simple_text_plot.py** , generates text-based plots directly in the terminal. Although its resolution is significantly lower, it supports zooming into specific regions and proves especially useful when graphical output is unavailable or when network connectivity to the server is limited.

The usage syntax is similar to that of the graphical plotting plugin. Use the `-h` flag to view additional options, such as plotting in the frequency domain, specifying the start, end, or duration of the plotted interval, and more.

```
/usr/share/doc/ngauge/examples/plugins/simple_text_plot.py <ngauge output .h5 file> <counter ID> [<counter ID> ...]
```

Simple text plot example:



> The sample plugins are provided as examples and are not integral parts of the **ngauge** tool. Dependencies such as NumPy, H5py, Matplotlib, plotext, and others may need to be installed separately to run these plugins.

7.6.4 Execution

To run **ngauge**:

```
ngauge <configuration YAML file>
```

The output is saved as an HDF5 file (**.h5**) in the path specified in the configuration YAML.

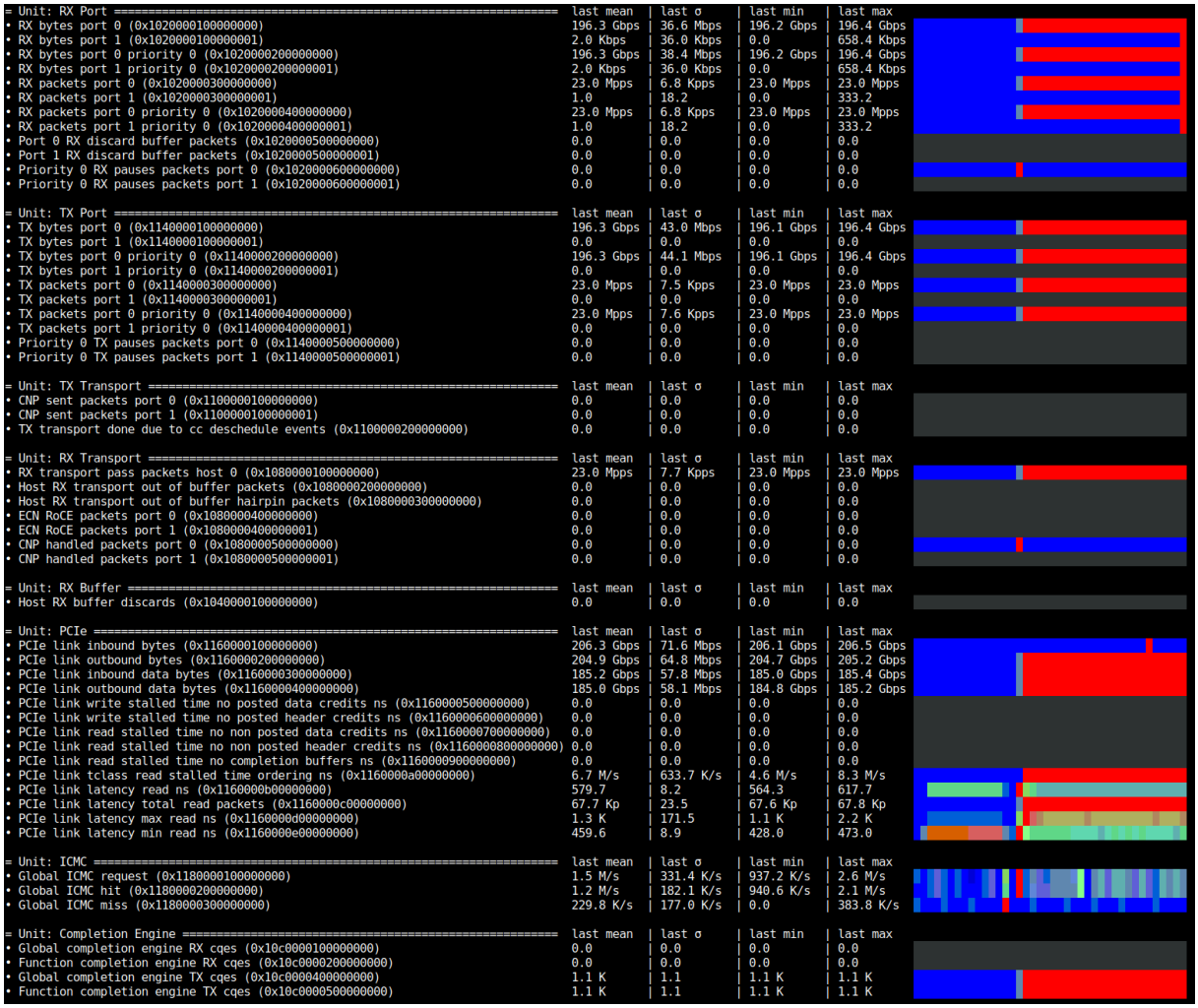
> To end a run before the full dataset is collected **Ctrl+C** (SIGINT) can be used. This is a normal and supported way to end the run, and all results collected up to that point will be saved as usual.

During the run, progress bars for each counter will be displayed. These bars provide visual feedback on the counter activity, with color coding to indicate value levels:

- Blue – Represents low values relative to other values of the same counter
- Red – Represents high values relative to other values of the same counter
- Intermediate colors (gradient) – Values between low and high, transitioning from blue to red
- Solid gray bars – Indicate no changes in the values of this counter during the run

Data is updated once per second, showing the mean, standard deviation (σ), minimum, and maximum values observed during the last second.

This visual representation helps track counter activity in real time, offering immediate insights into system behavior.



7.7 DOCA Perftest

This guide describes DOCA Perftest, an RDMA benchmarking tool designed for compute clusters that enables fine-tuned evaluation of bandwidth, message rate, and latency across various RDMA operations and complex multi-node scenarios.

7.7.1 Introduction

NVIDIA® DOCA Perftest is an RDMA benchmarking utility designed to evaluate performance across a wide range of compute and networking environments (from simple client-server tests to complex, distributed cluster scenarios).

It provides fine-grained benchmarking of bandwidth, message rate, and latency, while supporting diverse RDMA operations and configurations.

Key features:

- Comprehensive RDMA Benchmarks – Supports bandwidth, message rate, and latency testing.
- Unified RDMA Testing Tool – A single executable for all RDMA verbs, with rich configuration options and CUDA/GPUDirect RDMA integration.
- Cluster-Wide Benchmarking – Run distributed tests across multiple nodes, initiated from a single host, with aggregated performance results.
- Flexible Scenario Definition – Define complex multi-node, multi-test configurations via a JSON input file.
- Command-Line Simplicity – Quickly run local or point-to-point benchmarks directly from the CLI.
- Synchronized Execution – Ensures all benchmarks begin and end simultaneously for consistent results.

The DOCA Perftest utility simplifies evaluation and comparison of RDMA performance across applications and environments.

7.7.2 Comparison with Legacy Perftest

Unlike legacy RDMA benchmarking tools (e.g., **ib_write_bw**, **ib_send_lat**), DOCA Perftest is a native implementation designed for modern data centers. As opposed to a wrapper, it is a standalone product that replaces both the legacy tools and the custom orchestration scripts often required to run them at scale.

Architectural differences:

Feature	Legacy Perftest	DOCA Perftest
Scope	Point-to-Point (P2P) only	Single-node to Cluster-wide
Orchestration	Manual or third-party wrappers	Built-in (Single-host initiation)
Concurrency	Single-process per execution	Native multi-process/multi-core
Synchronization	Loose (Serial start)	Hardware-aligned (Synchronized start/stop)
Result Handling	Per-process manual extraction	Automatic cluster-wide aggregation

Benefits of migration to DOCA Perftest:

- Standard RDMA benchmarks require complex external scripts (Ansible, Bash, Python) to manage remote process launching, NUMA pinning, GPU selection and result parsing. DOCA Perftest handles these natively via the CLI or JSON scenario files.

- In large-scale clusters, measuring fabric congestion or incast/outcast scenarios requires all nodes to hit the network simultaneously. DOCA Perftest utilizes a centralized sync engine to ensure all processes begin and end traffic in a coordinated window, providing accuracy that is impossible to achieve with asynchronous legacy wrappers.
- While legacy tools require running multiple instances to saturate high-speed links (e.g., 200G/400G+), DOCA Perftest scales linearly across cores within a single execution using the `-N` or `-C` flags.
- Rather than collecting individual output files from dozens of servers, DOCA Perftest provides a unified report. This includes the full scenario definition, all raw results, and calculated aggregations per-device, per-node, and per-test.

7.7.3 Setup and Dependencies

DOCA Perftest is included in the DOCA Networking installation profile. It requires only a standard RDMA stack to run, with optional DOCA SDK components enabling advanced capabilities.

7.7.3.1 RDMA Core (libibverbs)

The only mandatory dependency is `libibverbs` (part of the `rdma-core` package). DOCA Perftest supports both the version bundled with the DOCA deployment and the upstream open-source release from rdma-core.

7.7.3.2 MPI (Multi-Node Scenarios)

Multi-node scenarios (JSON mode and `--traffic_pattern` CLI mode) require OpenMPI for orchestration.

DOCA Perftest relies on the OpenMPI package bundled with DOCA.

MPI is not required for simple point-to-point CLI benchmarks.

7.7.3.3 Optional DOCA SDK Components

Depending on the DOCA installation, additional capabilities are available:

- DOCA Verbs (`-r dv`) – The DOCA RDMA Verbs backend provides a high-performance alternative to standard `libibverbs` . Loaded on demand; if the package is not installed, DOCA Perftest operates normally with the default IBV driver.
- DOCA GPUNetIO (`--engine gpunetio`) – Offloads the RDMA data-path to GPU CUDA kernels using DOCA GPUNetIO. Requires the `doca-gpunetio` package, CUDA Toolkit, and a Volta+ GPU. For details, see GPUNetIO Engine (GPU-Initiated RDMA).

If an optional component is not installed and a user explicitly requests it (e.g., `-r dv` without DOCA Verbs), `doca_perftest` exits with a descriptive error message.

7.7.4 Point-To-Point Benchmarks

For simple benchmarks, run `doca_perftest` directly from the command line.

When invoked on the client, the utility automatically launches the corresponding server process (requires passwordless SSH) and selects optimal CPU cores on both systems based on NUMA affinity.

Example command:

```
# Run on client
doca_perftest -d mlx5_0 -n <server-host-name>
```

This is equivalent to running:

```
# On server
doca_perftest -d mlx5_0 -N 1 -c RC -v write -m bw -s 65536 -D 10

# On client
doca_perftest -d mlx5_0 -N 1 -c RC -v write -m bw -s 65536 -D 10 -n <server-host-name>
```

Parameter breakdown:

Parameter	Description
-d mlx5_0	Uses the device **mlx5_0**.
-N 1	Runs one process, automatically selecting an optimal core. (Use **-C core** to specify manually.)
-c RC	Uses a Reliable Connection (RC) transport.
-v write	Selects the Write verb for transmission.
-m bw	Measures bandwidth.
-s 65536	Sets message size to 65,536 bytes.
-D 10	Runs for 10 seconds.
-n server-host-name	(Client only) Specifies the remote target host.

For a full list of CLI arguments, run **doca_perftest -h** or **man doca_perftest**.

If passwordless SSH is not configured, you must manually run **doca_perftest** on both client and server, ensuring parameters match.

7.7.5 Scenario-Based Benchmarks

When a benchmark involves multiple devices, multiple nodes, or multiple test configurations, DOCA Perftest uses MPI to orchestrate execution across all participants. Tests are launched, synchronized, and results aggregated automatically on the initiating host.

Examples of scenarios that require orchestration:

- Multi-node – Traffic between two or more hosts across the network.

- Multi-device – Traffic across multiple NICs on the same host or across hosts (e.g., railed configurations with `mlx5_0` and `mlx5_1`).
- Multi-test – Several benchmark configurations running simultaneously within a single execution — for example, mixed message sizes ("elephant and mice"), latency under background load, or short-haul vs long-haul comparisons.

Scenario-based benchmarks can be configured in two ways:

- CLI mode (`--traffic_pattern`) – Define a single test directly from the command line. No configuration file required.
- JSON mode (`-f`) – Define one or more tests in a JSON configuration. Required for multi-test scenarios.

A test is a single benchmark configuration: one verb, one message size, one traffic pattern, and one set of participating nodes. CLI mode runs exactly one test; JSON mode can define multiple tests that execute in parallel, synchronized to start and stop together.

When to use which:

- Use CLI mode for single-test scenarios as it requires no configuration file and supports all traffic patterns, partitions, and standard parameters.
- Use JSON mode when you need multiple tests in one scenario, per-node parameter overrides, or integration with automation pipelines that generate configurations programmatically (see Automation and CI Integration).

7.7.5.1 CLI Scenario Mode

Use the `--traffic_pattern` flag to launch an orchestrated scenario directly from the command line:

```
# Basic: all-to-all across 12 nodes on mlx5_0
doca_perftest -d mlx5_0 -n my_node_0[1-12] --traffic_pattern all_to_all

# Advanced: railed all-to-all on two NICs, 10 QPs, 1KB messages, 60-second run
doca_perftest -d mlx5_[0,1] -n my_node_0[1-12] --traffic_pattern a2a --traffic_partition railed -q 10 -s 1024 -D 60
```

This mode of work builds a scenario configuration from CLI flags and feeds it into the standard MPI pipeline. All results, progress tracking, and aggregation work identically to JSON mode.

The advanced example runs an all-to-all pattern across 12 nodes using two NICs per node (`mlx5_0` and `mlx5_1`). The `railed` partition ensures each NIC only communicates with the same-named NIC on peer nodes (e.g., `mlx5_0` ↔ `mlx5_0`). The test uses 10 queue pairs, 1 KB messages, and runs for 60 seconds.

Key flags:

Flag	Description
`--traffic_pattern`	Required. Pattern: `all_to_all` (`a2a`), `bisection` (`b`), `ring` (`r`), `one_to_one` (`o2o`), `one_to_many` (`o2m`), `many_to_one` (`m2o`).

Flag	Description
`--traffic_partition`	Optional. Partition filter: `global`, `railed`, `cuda_railed`, `local`, `remote`, `cross_railed`, `remote_cross_railed`.
`-n`	Accepts hostlist syntax (e.g., `node[001-008]`).
`-d`	Accepts device hostlists (e.g., `mlx5_[0-1]`) for railed configurations.
`-G`	Accepts CUDA device hostlists (e.g., `[0-3]`); converted to integer in simple CLI mode.

All standard test parameters (`-v`, `-m`, `-s`, `-D`, `-q`, etc.) are supported alongside `--traffic_pattern`.

> ⚠ Important: The `--test_name`, `--iteration_sync`, and `--dual_unidir` flags are only valid in `--traffic_pattern` mode and will be rejected in simple CLI mode.

7.7.5.2 JSON Scenario Mode

For multi-test scenarios or advanced configurations, provide a JSON scenario definition with `-f` / `--input_file`:

- From file – Load from a JSON file. Scenario files can be version-controlled, shared across teams, and reused for regression testing.
- Inline string – Pass a raw JSON string directly (wrap in single quotes). Useful for automation pipelines where the configuration is generated programmatically, avoiding extra file I/O.

```
# From file
doca_perftest -f path_to_scenario_file.json

# From inline JSON string
doca_perftest -f '{"testNodes": [...], "trafficPattern": "ALL_TO_ALL"}'
```

JSON mode capabilities:

- Can be initiated from any node in the cluster (even non-participating ones).
- All benchmarks run simultaneously, with millisecond-level synchronization for benchmark start and stop across all nodes.
- Supports all traffic patterns and partitions.
- Fully compatible with all CLI parameters (JSON parameters inherit the same defaults).

Common multi-test scenarios:

- Mixed message sizes – Combine a large-message bandwidth test with a small-message message-rate test to evaluate how the fabric handles concurrent bulk and signaling traffic ("elephant and mice").
- Latency under load – Run bandwidth traffic as background noise while measuring latency on a separate set of connections, revealing how congestion affects tail latency.
- Short-haul vs long-haul – Define one test for intra-rack traffic (low-hop, optimized for latency) and another for inter-rack or cross-fabric traffic (different RDMA parameters tuned for distance).

- Message size sweep – Run tests at multiple message sizes (e.g., 64B, 4KB, 64KB, 1MB) in a single scenario to profile throughput across the full range.

> Example JSON configuration files are provided under: **/usr/share/doc/doca-perftest/examples/** . It is recommended to start by copying and modifying an existing example file.

7.7.5.3 Results and Output

By default, DOCA Perftest prints aggregated results to the terminal at the end of execution. This summary includes the total bandwidth, message rate, or latency statistics across all connections.

For detailed, machine-readable results, use the **outputFile** field (JSON mode) or the **-j** flag (CLI mode) to produce a structured JSON report. The JSON output provides full visibility into every level of the test hierarchy:

- Per-connection results – Raw results for each individual connection, broken down by process. This level of detail is critical for identifying anomalies: if a subset of connections underperforms due to congestion, routing issues, or hardware faults, the per-connection data reveals it (even when the aggregated summary appears healthy).
- Per-device and per-test aggregations – Results rolled up by NIC and by benchmark, making it easy to compare device-level performance across a cluster.
- Per bidirectional connection aggregation (**BIDIR** tests only) – For each bidirectional connection pair, the two per-direction results are combined into a single entry under **aggregatedResults.bidirConnections** (BW and message rate are summed across the two directions). The terminal report also includes a **"BIDIR Connections Summary"** section with the overall average, minimum, and maximum BW across all bidirectional pairs (the min/max rows identify which connection contributed each value). The per-direction results are also available in the top-level **connections** array.
- Full scenario configuration – The exact parameters used for execution, enabling reproducibility.
- Threshold pass/fail status (if configured) and QP histogram data (if enabled).

```
# JSON mode: add to scenario config
"outputFile": "results.json"

# CLI mode: save results to file
doca_perftest -d mlx5_0 -n server-name -j results.json

# CLI mode: default results path
doca_perftest -d mlx5_0 -n server-name -j
```

The JSON output schema is versioned. Schema version and DOCA Perftest version are included in the output under the **versionInfo** block.

For programmatic consumption of results via **stdout** , see Structured JSON Output to stdout.

7.7.6 Benchmark Results

7.7.6.1 Bandwidth

Bandwidth tests measure the aggregate data transfer rate and message-handling efficiency across all participating processes.

Metrics collected:

- Message Rate (million operations per second) – The total number of Completion Queue Entries (CQEs) processed per second across all test processes. This is an operation/completion rate; one CQE does not necessarily correspond to one packet on the wire.
- Bandwidth (Gigabits per second) – The application-level data transfer rate, calculated from completed benchmark operations tracked through CQEs and the configured message size (`-s` or `--msg_size`). This metric evaluates the system's ability to sustain high-throughput communication.

Measurement notes:

- Concurrency Handling – Results reflect the sum of completed-operation counts derived from CQEs across all concurrent test processes, as specified by the `-C` command-line argument or `--cores` field in the input JSON file.
- Test duration – The duration is averaged across all test processes, ensuring consistency in measuring sustained performance over time.

In multi-node scenarios, bandwidth results are aggregated at multiple levels: per-process, per-device, per-node, and per-test. The JSON output includes all aggregation levels for post-processing.

Interpretation tips:

Observation	Possible Cause
High message rate, low bandwidth	Small message sizes
High bandwidth, moderate message rate	Larger messages or a lower completed-operation rate

7.7.6.2 Latency

Latency tests measure RDMA operation delay using a ping-pong protocol. By default, both sides participate in the measurement loop: the requestor sends data, the responder receives it and sends data back, and the requestor reports half of the measured round-trip time (RTT/2). This is not a separately clock-synchronized endpoint-to-endpoint one-way measurement.

7.7.6.2.1 Latency Modes

Two measurement modes are available, selected via the `--lat_mode` CLI flag or the `"latencyMode"` JSON field:

Mode	CLI Value	Description
Full Round-Trip	`full_round_trip`	Default. Both sides run a ping-pong loop. Reports RTT/2 as an estimate of per-direction latency; it is not a directly measured one-way latency.
Requestor-side completion	`one_way`	Measures local requestor-side time from posting the operation to receiving its send completion. The responder does not send data back; this is not endpoint-to-endpoint one-way network latency.

```
# Full round-trip latency (default)
doca_perftest -d mlx5_0 -n server-name -m lat -v write

# One-way mode (local send-completion timing)
doca_perftest -d mlx5_0 -n server-name -m lat -v send --lat_mode one_way
```

JSON configuration:

```
"metric": "lat",
"latencyMode": "full_round_trip"
```

7.7.6.2.2 Verb Compatibility

All RDMA verbs support full round-trip latency by default. The one-way mode has additional restrictions:

Verb	`full_round_trip`	`one_way`
Write	Yes	Yes
Read	Yes	No
Send	Yes	Yes
Write IMM	Yes	Yes
Atomic	Yes	No

⚠ Important: `READ` and `ATOMIC` are inherently round-trip operations (`one_way` mode is not applicable).

7.7.6.2.3 Metrics Collected

- Minimum latency – Fastest observed transaction
- Maximum latency – Longest observed transaction
- Mean latency – Average across all iterations
- Median latency – Midpoint value (less influenced by outliers)
- Standard deviation – Variability indicator
- 99% tail latency – 99% of messages completed within this time
- 99.9% tail latency – Outlier detection for extreme cases

7.7.6.2.4 Measurement Notes

- Full round-trip mode uses a tight ping-pong loop where both sides alternate sending and receiving.
- Timing is collected on the requestor side.
- In `one_way` mode, only the requestor posts and polls its send completion queue. The responder does not send data back; the result is requestor-side operation-completion time, not endpoint-to-endpoint one-way network latency.
- Aggregated across processes for final reporting.

Interpretation tips:

Pattern	Insight
Low mean/median, high max/tail	Indicates jitter or queue buildup
Low standard deviation	Indicates stable and predictable performance
High 99%/99.9% tail	Indicates possible SLA breaches in real-time workloads

When raw data output is enabled, latency tests export per-iteration sample data in CSV format.

Bandwidth raw data: enabling raw-data output on a bandwidth test (`metric: bw` with `rawDataFilePath` set, or `--save_raw_data <path>` in CLI) samples cumulative bytes at a fixed interval and writes one merged CSV per test at `rawDataFilePath`. CLI and MPI use the same tidy format. CLI includes both sides of bidirectional and `SEND` tests when the server is auto-launched; a manually launched server must also be given `--save_raw_data`.

- `rawDataIntervalMs` / `--raw_data_interval_ms`: sample interval in ms (default 1).
- CLI resolves `$TMPDIR` independently on each host; MPI forwards the launcher's `$TMPDIR` to all ranks. Every selected path must be disk-backed and node-local.
- BW raw-data collection is not supported with the GPUNetIO execution engine.

Visualize the merged CSV with the bundled dashboard: `python3 tools/plot_bw_raw_data.py <csv> --serve` (open the printed `localhost` URL; drill down host → NIC → process), or `--standalone --out dashboard.html` for a single portable file.

7.7.7 Common Arguments and Use-Cases

This section highlights some of the most commonly used parameters and use-cases.

7.7.7.1 Traffic Directions

DOCA Perftest supports three traffic directions that determine how data moves between nodes and how queue pairs are allocated.

7.7.7.1.1 Unidirectional Traffic (Default)

- In unidirectional mode, traffic flows in one direction only.
- The client (requestor) initiates operations, and the server (responder) receives them.
- This is the default mode and provides clear, predictable performance metrics.

7.7.7.1.2 Bidirectional Traffic

In bidirectional mode, traffic flows in both directions simultaneously. Each side acts as both requestor and responder, creating full-duplex communication.

Bidirectional tests use two traffic runners (requestor + responder) sharing resources. It may show different aggregate bandwidth than 2× unidirectional. Bidirectional mode is supported for Write, Send and Atomic verbs.

Run bidirectional traffic from the command line:

```
# Enable bidirectional traffic
doca_perftest -d mlx5_0 -n <server-name> -b
```

7.7.7.1.3 Dual Unidirectional Traffic

Dual unidirectional mode runs two simultaneous unidirectional flows in opposite directions. Unlike **BIDIR**, each direction uses its own connection and dedicated QP, and each endpoint retains a
single requestor or responder role per connection.

Run dual unidirectional traffic in CLI scenario mode:

```
doca_perftest -d mlx5_0 -n 'node[01-02]' --traffic_pattern one_to_one --dual_unidir
```

The **--dual_unidir** flag requires **--traffic_pattern** and is mutually exclusive with **--bidirectional**. In JSON mode, set **"trafficDirection"** to **"DUAL_UNIDIR"**. The other supported JSON values are **"UNIDIR"** and **"BIDIR"**.

7.7.7.2 Traffic Patterns

Traffic patterns provide built-in shortcuts for complex multi-node communication scenarios.

While these configurations were always possible through detailed JSON definitions, traffic patterns dramatically simplify setup for common topologies.

Example JSONs using traffic patterns are available under **/usr/share/doc/doca-perftest/examples**.

Available patterns:

- **ONE_TO_ONE**
- **ONE_TO_MANY**
- **MANY_TO_ONE**
- **ALL_TO_ALL**
- **BISECTION**

- **RING**

> Multicast is not supported. Each connection is point-to-point, synchronized to start simultaneously.

They collapse complex multi-node wiring into a few lines of JSON. Instead of manually listing dozens of connections, you specify a regex-like host list and a pattern (e.g., **ALL_TO_ALL**) and DOCA Perftest generates and synchronizes all connections for you.

7.7.7.2.1 One-to-One (O2O)

Simple point-to-point between two nodes; useful for baseline performance testing.

```
"testNodes": [ {"hostName": "node01", "deviceName": "mlx5_0"},
               {"hostName": "node02", "deviceName": "mlx5_0"} ],
"trafficPattern": "ONE_TO_ONE"
```

7.7.7.2.2 One-to-Many (O2M)

Single sender to multiple receivers; the first node sends to all others.

```
"testNodes": [ {"hostName": "sender", "deviceName": "mlx5_0"},
               {"hostName": "receiver[1-10]", "deviceName": "mlx5_0"} ],
"trafficPattern": "ONE_TO_MANY"
```

This creates 10 connections: sender → receiver1, sender → receiver2, ..., sender → receiver10.

7.7.7.2.3 Many-to-One (M2O)

Multiple senders to one receiver; all nodes send to the first node.

```
"testNodes": [  {"hostName": "aggregator", "deviceName": "mlx5_0"},
                {"hostName": "client[01-20]", "deviceName": "mlx5_0"}  ],
"trafficPattern": "MANY_TO_ONE"
```

This creates 20 connections: client1 → aggregator, client2 → aggregator, ..., client20 → aggregator.

7.7.7.2.4 All-to-All (A2A)

Full-mesh connectivity; every node connects to every other node.

```
"testNodes": [  {"hostName": "compute[01-16]", "deviceName": "mlx5_0"} ],
"trafficPattern": "ALL_TO_ALL",
"trafficDirection": "UNIDIR"
```

This creates 240 connections (16×15) for **UNIDIR** or **DUAL_UNIDIR**, or 120 bidirectional pairs for **BIDIR**.

7.7.7.2.4.1 Variable All-to-All (per-pair message sizes)

Standard **ALL_TO_ALL** uses a single message size for every directional pair. For workloads where each **(source, dest)** connection moves a different volume of data (e.g. UCC **ALLTOALLV**, MoE expert routing), the per-pair size can be supplied via the **ptiniFile** parameter, pointing at a UCC **ALLTOALLV** pattern file (**.ptini**) as produced by UCC's own benchmark tool, **ucc_perftest**. The values in such a file are element counts, and **ptiniDatatype** names the element type they are scaled by (default **float32**).

JSON:

```
"trafficPattern": "ALL_TO_ALL",
"trafficDirection": "UNIDIR",
"ptiniFile": "/path/to/sizes.ptini",
"ptiniDatatype": "float32"
```

CLI scenario mode:

```
doca_perftest --traffic_pattern all_to_all -n compute[01-05] -d mlx5_0 \
              --ptini_file /path/to/sizes.ptini --ptini_datatype float32 -D 10
```

~/... paths are expanded against **$HOME**.

File format

The file is a UCC (**openucx/ucc**) **ALLTOALLV** pattern file (the same format **ucc_perftest** consumes):

```
[format]
version = 1.0

[collective]
type = ALLTOALLV
team_size = 5
flags = COUNT_32BIT
counts = {
    0,        33554432, 262144,   2097152,  524288
    33554432, 0,        1048576,  0,        524288
    262144,   1048576,  0,        16777216, 524288
    2097152,  0,        16777216, 0,        524288
    2097152,  0,        16777216, 1048576,  0
}
```

Inside **counts**, each row is a source rank and each column is a destination rank. Values are element counts, not bytes: exactly as in UCC, the wire size of a pair is its count multiplied by the size of the datatype. The file itself does not carry the datatype; **ucc_perftest** takes it on its command line via **-d**, and its file reader never looks for a **datatype** key. **ptiniDatatype** / **--ptini_datatype** is the DOCA Perftest equivalent of that flag, and any **datatype = ...** line inside a **.ptini** file is ignored.

ptiniDatatype accepts the UCC predefined datatype names (**int8**, **uint8**, **int16**, **uint16**, **float16**, **bfloat16**, **int32**, **uint32**, **float32**, **float32_complex**, **int64**, **uint64**, **float64**, **float64_complex**, **int128**, **uint128**, **float128**, **float128_complex**) and defaults to **float32** (4 bytes). That default is **ucc_perftest**'s own default, so replaying a capture without naming a datatype produces the same message sizes UCC would have produced. With it, the matrix above spans 1 MiB to 128 MiB. Specify **int8** if you want the counts to be plain byte counts.

`type`, `team_size`, and `counts` are interpreted; `datatype`, `flags`, and the `[format]` section are ignored. `type` must be `ALLTOALLV` and `team_size` must equal the rank count when either is present. Line breaks inside `counts` are cosmetic (UCC flattens the block into a single list) and `;` or `#` begin a comment.

For `N` ranks the block must hold exactly `N × N` values. Diagonal entries are ignored. Zero off-diagonal entries skip that directional pair (no traffic, no QP). A file holding more than one `[collective]` pattern is rejected: UCC cycles through every pattern within a run, while a DOCA Perftest test benchmarks exactly one, so such a file must be split.

The expanded `testNodes` list (after hostlist + device-list + `cudaDeviceId` expansion) defines the rank ordering: rank `i` is the `i`-th entry of the expanded list.

Constraints

- Only `ALL_TO_ALL` / `A2A`. Other patterns reject `ptiniFile`.
- `UNIDIR` only, in every mode (including `iterationSync`). `BIDIR + ptiniFile` is rejected because a `BIDIR` connection carries one `msgSize` for both directions and would collapse asymmetric pair sizes. With `UNIDIR`, each directed pair is independent. Iteration-sync connections report their own BW; the combined synchronized result omits aggregate `bwResults` and displays `N/A`.
- `msgSize`, `operations`, and `ptiniFile` are mutually exclusive.
- `ptiniDatatype` requires `ptiniFile`; specifying it on its own is rejected.
- `bwThreshold` is not supported with variable-A2A `iterationSync` because bandwidth is unavailable.
- `verb` is independent and can be combined with `ptiniFile`.

Visualizing the matrix

When you set `ptiniFile`, you see the parsed matrix before any traffic starts, so you can confirm the run will send the sizes you meant it to. Rows are the sending endpoints and columns the receiving ones, named the same way the connection matrix report names them.

`Total sent` and `Total recv` close the table with each endpoint's row and column sums, so you can tell at a glance which NIC carries the most bytes. The diagonal shows `-`, a pair that carries no traffic shows `0`, and every other cell shows its byte count. The line above the table names the datatype the `.ptini` counts were scaled by, so you can check the scaling that produced those bytes.

```
Test: VariableA2A, endpoints: 5
Hosts: gb-ida-lr1-r8-*
.ptini counts scaled by datatype float32 (4 bytes per element)
Message Size Matrix [bytes]:
             |    01     |    02     |    03     |    04     |    05     |   Total
             |  mlx5_0   |  mlx5_0   |  mlx5_0   |  mlx5_0   |  mlx5_0   |   sent
----- ------ | --------- | --------- | --------- | --------- | --------- | ---------
01    mlx5_0 |     -     | 134217728 |  1048576  |  8388608  |  2097152  | 145752064
----- ------ | --------- | --------- | --------- | --------- | --------- | ---------
02    mlx5_0 | 134217728 |     -     |  4194304  |     0     |  2097152  | 140509184
----- ------ | --------- | --------- | --------- | --------- | --------- | ---------
03    mlx5_0 |  1048576  |  4194304  |     -     | 67108864  |  2097152  | 74448896
----- ------ | --------- | --------- | --------- | --------- | --------- | ---------
04    mlx5_0 |  8388608  |     0     | 67108864  |     -     |  2097152  | 77594624
```

```
----- ------ | --------- | --------- | --------- | --------- | --------- | ---------
05    mlx5_0 |  8388608  |     0     | 67108864  |  4194304  |     -     | 79691776
----- ------ | --------- | --------- | --------- | --------- | --------- | ---------
Total recv   | 152043520 | 138412032 | 139460608 | 79691776  |  8388608
```

This example omits `(2→4)`, `(4→2)`, and `(5→2)`, leaving 17 of 20 directional connections; the parser logs `skipped 3/20 zero-size pairs`.

The output is `stdout`-only and is never written to the results JSON. The per-pair `msgSize` values survive into `connections[*].testConfig.resourceConfig.operations[0].msgSize` in the results JSON, so the same totals can be reconstructed programmatically by grouping the connections array.

Runnable configurations are provided in `input/examples/all2all_v_example.json` and `input/examples/all2all_v_iter_sync_example.json`.

The JSON output schema version (`versionInfo.jsonOutputSchema`) is `1.4.0`. Schema `1.3.0` added `ptiniFile` and `ptiniDatatype` to the serialized general config; `ptiniDatatype` is empty for runs that do not use a `.ptini` file.

7.7.7.2.5 Bisection (B)

Divides nodes into two equal halves; the first half connects to the second half. Requires an even number of nodes.

```
"testNodes": [  {"hostName": "rack1-[01-10]", "deviceName": "mlx5_0"},
                {"hostName": "rack2-[01-10]", "deviceName": "mlx5_0"} ],
"trafficPattern": "BISECTION"
```

This creates 10 connections: rack1-01 ↔ rack2-01, rack1-02 ↔ rack2-02, ..., rack1-10 ↔ rack2-10.

7.7.7.2.6 Ring (R)

Each node connects to the next in a circular ring: node *i* sends to node *(i+1) % N*. Requires at least 2 nodes.

```
"testNodes": [  {"hostName": "compute[01-08]", "deviceName": "mlx5_0"} ],
"trafficPattern": "RING"
```

This creates 8 connections: compute01 → compute02, compute02 → compute03, ..., compute08 → compute01. All connections run simultaneously, not sequentially.

7.7.7.2.7 Traffic Partitions

When nodes have multiple NICs, a traffic partition specifies which connections to establish based on the relationship between devices and hosts. Combined with a traffic pattern, it provides a concise way to describe common multi-device topologies in a single test definition.

Set via the `"trafficPartition"` JSON field or the `--traffic_partition` CLI flag.

Partition	Use case	Connections established
`GLOBAL`	Test all possible connections (default).	`nodeA:mlx5_0` ↔ `nodeB:mlx5_0`, `nodeA:mlx5_0` ↔ `nodeB:mlx5_1`, ...
`RAILED`	Test each NIC against its counterpart on peer nodes.	`nodeA:mlx5_0` ↔ `nodeB:mlx5_0`, `nodeA:mlx5_1` ↔ `nodeB:mlx5_1`
`CUDA_RAILED`	Test each NIC/GPU pair against its counterpart.	`nodeA:mlx5_0/GPU0` ↔ `nodeB:mlx5_0/GPU0`
`LOCAL`	Test NIC-to-NIC traffic within the same host (loopback).	`nodeA:mlx5_0` → `nodeA:mlx5_1` `nodeB:mlx5_0` → `nodeB:mlx5_1`
`REMOTE`	Test only cross-host traffic, excluding loopback.	`nodeA:mlx5_0` → `nodeB:mlx5_0` `nodeA:mlx5_0` → `nodeB:mlx5_1`
`CROSS_RAILED`	Test each NIC against a different NIC on peer nodes.	`nodeA:mlx5_0` ↔ `nodeB:mlx5_1`, `nodeA:mlx5_0` ↔ `nodeA:mlx5_1`, `nodeA:mlx5_1` ↔ `nodeB:mlx5_0`
`REMOTE_CROSS_RAILED`	Same as `CROSS_RAILED`, but only across different hosts.	`nodeA:mlx5_0` ↔ `nodeB:mlx5_1`, `nodeA:mlx5_1` ↔ `nodeB:mlx5_0`

Example — railed all-to-all across a cluster with two NICs per node:

```
"testNodes": [  {"hostName": "node[01-16]", "deviceName": "mlx5_[0,1]"} ],
"trafficPattern": "ALL_TO_ALL",
"trafficPartition": "RAILED"
```

Without the partition, achieving the same railed topology would require a separate test for each device with otherwise identical parameters.

Traffic partitions apply to any traffic pattern. The partition determines which connections are kept regardless of how they were generated. Some pattern-partition combinations may result in zero connections (for example, `LOCAL` with a single device per node, or `REMOTE` on a one-to-one test within the same host).

7.7.7.3 Per-iteration-sync Flow (Lock-step Benchmarking)

Designed to mimic AI workloads, this flow ensures data transfer occurs in distinct, synchronized steps. By forcing every process to wait for all peers to complete an iteration before proceeding, it enables granular data validation and allows for QP parameter modification between steps.

Configuration constraints:

- Requires the `ALL_TO_ALL` pattern. Supports both `UNIDIR` and `BIDIR` traffic: in `BIDIR` both endpoints of a connection exchange data each iteration; in `UNIDIR` each directed pair `(i→j)` is its own connection and only the requestor sends data while the responder receives. `UNIDIR` is required for variable-A2A (`ptiniFile`) so each pair can use its own message size. The lock-step sync handshake (zero-length RDMA Write with Immediate) is symmetric and runs in both directions regardless.
- Must be defined by specific iterations (time-based duration is not supported).

- Supported in both JSON mode (`"iterationSync": true`) and CLI mode (`--traffic_pattern a2a --iteration_sync`).

7.7.7.3.1 Logic and Implementation

The flow utilizes an `ALL_TO_ALL` pattern (`UNIDIR` or `BIDIR`). Each iteration consists of four distinct phases:

1. Data phase:
    a. Every sender sends a data message to its peers (in `BIDIR` both endpoints send; in `UNIDIR` only the requestor of each directed pair sends, while the responder receives).
    b. The total `msgSize` is split across available QPs. Each QP writes to a specific offset to utilize the full buffer.
2. Sync phase:
    - Once data transfer completes, each process sends a Sync Message to all peers.
    - The Sync Message is a zero-length RDMA Write with Immediate Data.
3. Barrier phase:
    - A process completes the iteration only after it has received confirmation for its own Sync Send *and* received Sync Messages from all peers.
4. Post-iteration (management) phase:
    - Occurs after synchronization but before the next iteration begins.
    - Performs non-timed management tasks, such as modifying QPs, checking data validation results, or updating pointers.

7.7.7.3.2 Configuration

Configure iteration-sync in JSON (per-test fields below) or via CLI (`--traffic_pattern a2a --iteration_sync`).

Field	Value	Notes
`iterationSync`	`true`	Activates the flow logic.
`trafficPattern`	`"ALL_TO_ALL"`	Required. Must be used even for 1:1 node connections.
`trafficDirection`	`"UNIDIR"` / `"BIDIR"`	Optional (default `UNIDIR`). `BIDIR` exchanges data both ways per connection; `UNIDIR` sends one way per directed pair and is required for variable-A2A (`ptiniFile`).
`iterations`	(Integer)	Required. Defines the run duration. Time-based `"Duration"` mode is not supported.
`verb`	`"write"` / `"write_imm"`	Determines the verb used in the Data Phase.
`metric`	`"bw"`	Calculates both Bandwidth and Latency.

7.7.7.3.3 Data Validation Integration

When `dataValidation` is set to `true`, the flow performs a bit-exact verification of all received data at the end of every iteration.

- Highly effective for catching transient data corruption in complex A2A patterns.
- Validation occurs during the "post-iteration" management phase, outside of the timed performance interval.

7.7.7.3.4 Bandwidth Measurement

In bidirectional iteration-sync mode, bandwidth includes both TX and RX bytes. In unidirectional mode each
directed connection reports its own one-way bandwidth. Variable-A2A connections retain those per-connection
values, while the combined synchronized result reports aggregate BW as `N/A`.

7.7.7.3.5 Limitations

- Due to the heavy synchronization barrier, the measured "streaming" bandwidth will be lower than a standard continuous A2A test.
- A single scenario file cannot mix synchronization types. All tests must be with or without iteration-sync.

When raw data output is enabled, iteration-sync and latency tests export per-iteration data in CSV format.

7.7.7.4 Peer Shifting

`peerShifting` runs an `ALL_TO_ALL` mesh as *N-1* sequential rounds instead of all at once: in round *p* every endpoint *i* sends to endpoint *(i+p) % N*, so each endpoint has exactly one peer per round. Across the rounds every ordered endpoint pair is measured, giving a per-route bandwidth or latency figure for the whole fabric without full-mesh contention.

Supported in both JSON mode (`"peerShifting": true`) and CLI mode (`--traffic_pattern all_to_all --peer_shifting`).

```
"testNodes": [ {"hostName": "compute[01-08]", "deviceName": "mlx5_0"} ],
"trafficPattern": "ALL_TO_ALL",
"trafficDirection": "UNIDIR",
"peerShifting": true
```

This resolves 56 connections grouped into 7 rounds of 8. Each round is timed separately and the rounds run one after another, all reported under a single test. If a pair fails it is reported as failed and the remaining rounds still run. In the JSON output every connection carries its round in `generalConfig.trafficPhaseId`.

7.7.7.5 Hostname, Device Name, and CUDA Device ID Ranged Selection

To streamline configuration for multi-node and multi-device scenarios, DOCA Perftest supports bracket-based range expansion in both JSON and CLI modes. This allows you to define large-scale clusters concisely.

7.7.7.5.1 Supported Syntax

Feature	Syntax Example	Expansion Result
Numeric Range	`perf-host[0-3]`	`perf-host0` , `perf-host1` , `perf-host2` , `perf-host3`
Comma List	`perf-host[0,2,4]`	`perf-host0` , `perf-host2` , `perf-host4`
Zero Padding	`node[01-03]`	`node01` , `node02` , `node03` (Padding is preserved)

7.7.7.5.2 Expansion Logic

When ranges are defined for both hostnames and device names, the tool generates all possible combinations (Cartesian product).

For example:

- Input: `hostName=host[1-2]` , `deviceName=mlx5_[0-1]`
- Result (4 Connections):
    a. `host1` ↔ `mlx5_0`
    b. `host1` ↔ `mlx5_1`
    c. `host2` ↔ `mlx5_0`
    d. `host2` ↔ `mlx5_1`

7.7.7.5.3 CUDA Device ID Ranged Selection

The `cudaDeviceId` field supports the same bracket-based range expansion syntax. When both `deviceName` and `cudaDeviceId` use ranges, values are paired by order (not as a Cartesian product).

Examples:

- `deviceName = "mlx5_[0-1]"` with `cudaDeviceId = "[1,0]"` creates the pairs: `[mlx5_0, GPU1]` and `[mlx5_1, GPU0]` .
- `deviceName = "mlx5_[0-3]"` with `cudaDeviceId = "[0,0,1,1]"` creates the pairs: `[mlx5_0, GPU0]` , `[mlx5_1, GPU0]` , `[mlx5_2, GPU1]` , `[mlx5_3, GPU1]` .

The number of `deviceName` elements must be equal to the number of `cudaDeviceId` elements.

7.7.7.5.4 Duplicate Connections

Duplicate `(hostName, deviceName, cudaDeviceId (if exists))` connections are filtered out after pattern resolution.

7.7.7.6 Multiprocess (Cores)

DOCA Perftest can run synchronized multi-process tests, ensuring traffic starts simultaneously across all cores.

By default, it runs a single process on one automatically selected core.

Process and core selection:

Option	Description
`-N` / `"numOfProcesses"`	Number of processes; cores auto-selected.
`-C` / `"cores"`	Explicitly specify core IDs or ranges.
`--exclude_cores` / `"excludedCores"`	Exclude core IDs or ranges from automatic core selection.

Examples:

```
# Run on 3 synchronized processes (cores auto-selected)
doca_perftest -d mlx5_0 -n <server> -N 3

# Auto-select cores while excluding specific cores
doca_perftest -d mlx5_0 -n <server> -N 4 --exclude_cores 0,1

# Run on specific cores
doca_perftest -d mlx5_0 -n <server> -C 5
doca_perftest -d mlx5_0 -n <server> -C 5,7
doca_perftest -d mlx5_0 -n <server> -C 5-9
```

Host data buffers are placed on the NUMA node of the RDMA device that transfers them, whichever cores the test runs on, so that data transfers do not cross the interconnect. For a device that reports no node of its own, the node of the first selected core is used instead. The resolved node is reported as `numaNode` in the test configuration of the output.

7.7.7.7 Working with GPUs – Device Selection

DOCA Perftest can automatically select the most suitable GPU for each network device based on PCIe topology proximity. The ranking follows NVIDIA's `nvidia-smi topo` hierarchy: NV > PIX > PXB > PHB > NODE > SYS. It also load-balances by choosing the least-loaded GPU among those with the best available connection type.

This ensures that the GPU closest to the NIC is chosen, minimizing latency and maximizing throughput.

Although auto-selection is the default behavior, users can still manually specify a GPU device using the `-G` argument in CLI mode, or the `"cudaDeviceId"` field in JSON mode.

```
# Manually choose a specific GPU
doca_perftest -d mlx5_0 -n server-name -G 0

# Automatically select both GPU and memory type (recommended)
doca_perftest -d mlx5_0 -n server-name -M cuda
```

```
# Deprecated syntax (still supported, equivalent to cuda_auto_detect)
doca_perftest -d mlx5_0 -n server-name --cuda 0
```

7.7.7.8 Working with GPUs – Memory Types

RDMA operations can leverage GPU memory directly, bypassing CPU involvement for maximum throughput and minimal latency.

DOCA Perftest supports several CUDA memory modes optimized for different hardware and driver configurations.

7.7.7.8.1 Auto-Detection Mode (cuda_auto_detect)

Automatically selects the best available CUDA memory type in this order:

1. Data Direct
2. DMA-BUF
3. Peermem

This is the recommended mode for most users.

```
# Auto-detect best GPU memory type (recommended)
doca_perftest -d mlx5_0 -n server-name -M cuda -G 0

# With custom CUDA library path
doca_perftest -d mlx5_0 -n server-name -M cuda -G 0 --cuda_lib_path /usr/local/cuda-12/lib64

# Deprecated but equivalent syntax
doca_perftest -d mlx5_0 -n server-name --cuda 0
```

Fallback behavior: With **-M cuda_auto_detect**, **doca_perftest** automatically tries **cuda_data_direct** → **cuda_dmabuf** → **cuda_peermem** in this order.

7.7.7.8.2 Standard CUDA Memory (cuda_peermem)

Traditional CUDA peer-memory allocation.

Supported on all CUDA-capable systems, though with slightly higher overhead compared to newer methods.

```
# Explicitly force peermem (bypasses auto-detect)
doca_perftest -d mlx5_0 -n server-name -M cuda_peermem -G 0
```

7.7.7.8.3 DMA-BUF Memory (cuda_dmabuf)

Uses the Linux DMA-BUF framework for zero-copy GPU–NIC transfers. Requires CUDA 11.7+ and kernel support.

```
doca_perftest -d mlx5_0 -n server-name -M cuda_dmabuf -G 0
```

7.7.7.8.4 Data Direct Memory (cuda_data_direct)

Most efficient GPU memory access method using direct PCIe mappings. Requires specific hardware and driver support; provides the lowest latency and highest throughput.

```
doca_perftest -d mlx5_0 -n server-name -M cuda_data_direct -G 0
```

7.7.7.9 Memory Types

Beyond GPU memory types, DOCA Perftest supports several memory allocation strategies for RDMA operations.

7.7.7.9.1 Host Memory (host)

Default mode using standard system RAM.

```
# Default host memory usage
doca_perftest -d mlx5_0 -n <server-name>

# Explicitly specify host memory
doca_perftest -d mlx5_0 -n <server-name> -M host
```

7.7.7.9.2 Null Memory Region (nullmr)

Does not allocate real memory; useful for ultra-low-latency synthetic tests.

```
# Null memory region for bandwidth testing
doca_perftest -d mlx5_0 -n <server-name> -M nullmr
```

7.7.7.9.3 Device Memory (device)

Allocates memory directly on the adapter hardware (limited by on-board capacity).

```
doca_perftest -d mlx5_0 -n <server-name> -M device
```

7.7.7.10 RDMA Drivers

Two RDMA driver backends are supported:

The available drivers depend on your installed packages and hardware.

Driver	Prerequisites	Usage
IBV (`libibverbs`)	The standard RDMA Verbs delivered as part of DOCA-OFED (and standard inbox drivers). Recommended for general compatibility across all IB/RoCE adapters.	`-r ibv` (default)

Driver	Prerequisites	Usage
DV (`doca_verbs`)	The specialized DOCA RDMA Verbs backend. This provides a high-performance alternative to standard verbs and is optimized for the DOCA SDK ecosystem.	`-r dv`

DOCA Verbs is not a mandatory dependency. DOCA Perftest runs without it and loads the DOCA Verbs driver on demand when `-r dv` is requested. If the DOCA Verbs package is not installed, the IBV driver is always available.

7.7.7.10.1 Atomic Verbs

RDMA atomic operations (`FETCH_AND_ADD` , `COMPARE_AND_SWAP`) are supported on `RC` queue pairs with the `libibverbs` driver (`-r ibv`).

Interface	Instruction
CLI	`-v atomic_fa` or `-v atomic_cs`
JSON	`"verb": "atomic_fa"` or `"verb": "atomic_cs"`
Operations	`atomic_fa:8` / `atomic_cs:8` in `--operations` or JSON `operations` array

Constraints:

Parameter	Value / rule
Message size	Fixed at `8` bytes (`msgSize` defaults to `8` when omitted)
Connection type	`RC` only
Driver	`libibverbs` (`ibv`) only
Inline	Not supported
Bidirectional BW	Supported (`-b` , or `--dual_unidir` in `--traffic_pattern` mode)
Iteration sync / data validation	Not supported
Multi-verb groups	Supported in bandwidth tests (`--operations`); multi-verb groups are not available for latency tests

Example:

```
doca_perftest -d mlx5_0 -n server-name -m lat -v atomic_fa -D 10
doca_perftest -d mlx5_0 -n server-name -m bw -v atomic_cs -D 10
doca_perftest -d mlx5_0 -n server-name -m bw --operations write:65536,atomic_fa:8,atomic_cs:8 -D 10
```

7.7.7.11 Auto-launching Remote Server

DOCA Perftest can automatically launch the remote server via SSH (CLI-only).

Requires passwordless SSH and identical versions on both sides.

Prerequisites:

- Available only in CLI mode (not relevant for JSON configuration).
- Requires passwordless SSH access to the server machine.
- Both client and server must have the same DOCA Perftest version.

```
# Auto-launch server (default)
doca_perftest -d mlx5_0 -n server-name

# Disable auto-launch
doca_perftest -d mlx5_0 -n server-name --launch_server disable
```

Running server detection:

The system checks if a server is already running on the target host. If detected, it connects to the existing server instead of launching a new one. This prevents multiple server instances and allows sharing servers between tests.

Server override examples:

```
# Server uses different device than client
doca_perftest -d mlx5_0 -n server-name --server_device mlx5_1

# Server uses different memory type
doca_perftest -d mlx5_0 -n server-name -M host --server_mem_type cuda_auto_detect

# Server runs on specific cores
doca_perftest -d mlx5_0 -n server-name -C 0-3 --server_cores 4-7

# Alternate server executable path
doca_perftest -d mlx5_0 -n server-name --server_exe /tmp/other_doca_perftest_version

# Different SSH username, supported by passwordless-ssh
doca_perftest -d mlx5_0 -n server-name --server_username testuser
```

7.7.7.12 QP Histogram

The QP histogram provides visibility into how work is distributed across multiple queue pairs during a test. This is useful for identifying load balancing issues, scheduling inefficiencies, or hardware limitations when using multiple QPs.

Enabling QP histogram:

Command Line Interface (CLI)

CLI execution is strictly restricted to one process (`-N 1` is the default).

```
# Enable QP histogram with multiple queue pairs
doca_perftest -d mlx5_0 -n server-name -q 8 -H

# Shorthand flag
doca_perftest -d mlx5_0 -n server-name -q 8 --qp_histogram
```

JSON Configuration Mode

Supports multi-process execution.

```
"printQpHistogram": true
```

Example CLI output:

```
-------------------- QP WORK DISTRIBUTION --------------------
```

```
Qp num 0:                                          45.23 Gbit/sec  |  Relative deviation: -2.1%
Qp num 1:                                          46.89 Gbit/sec  |  Relative deviation: 1.5%
Qp num 2:                                          45.67 Gbit/sec  |  Relative deviation: -1.2%
Qp num 3:                                          48.21 Gbit/sec  |  Relative deviation: 4.3%
```

JSON output:

The histogram data is nested within the **resultsPerProcess** array:

```
"connections": [
    {
        "resultsPerProcess": [
            {
                "qpHistogram": [
                    {
                        "qpIndex": 0,
                        "bwGbSec": 45.23,
                        "relativeDeviation": -2.1
                    },
                    {
                        "qpIndex": 1,
                        "bwGbSec": 46.89,
                        "relativeDeviation": 1.5
                    },
                    {
                        "qpIndex": 2,
                        "bwGbSec": 45.67,
                        "relativeDeviation": -1.2
                    },
                    {
                        "qpIndex": 3,
                        "bwGbSec": 48.21,
                        "relativeDeviation": 4.3
                    }
                ]
            }
        ]
    }
]
```

7.7.7.13 Start Packet Sequence Number

Start PSN controls the initial Packet Sequence Number for each Queue Pair (QP) at connection initialization. If unspecified, a random value is generated.

This feature is essential for debugging sequence-sensitive behavior, ensuring reproducibility, and interoperability testing.

Interface	Configuration	Requirement
CLI	**--start-psn <val1>,<val2>...**	The number of values must exactly match the number of QPs.
JSON	**"startPsn": { "qp0": 1000, "qp1": 1001 }**	Keys must be contiguous (e.g., **qp0**, **qp1**) and start at **qp0**.

7.7.7.14 Data Validation

Data validation verifies the integrity of RDMA traffic during bandwidth tests. When enabled, the requestor generates a deterministic payload for each message, and the responder compares the received data against the expected pattern.

To enable validation, set **"dataValidation": true** in JSON or use **--validate** in CLI mode.

> No other specific JSON changes are required, provided the test meets the constraints listed below.

Validation introduces CPU and memory overhead, reducing measured bandwidth. `iteration-sync` mode, however, performs validation during the inter-iteration gap, preserving performance accuracy.

7.7.7.14.1 Prerequisites and Constraints

- Test type: Supported only for bandwidth tests (latency testing is not supported).
- Supported modes:
  - Standard `send` verb tests.
  - Tests running in `iteration-sync` mode.
- Buffer configuration: `rxDepth` must be greater than or equal to `txDepth`.
- Warmup: Warmup must be explicitly disabled (`-w 0`).
- Enhanced Reliability (ER): If "ER Auto Mode" is set, it will be automatically disabled when validation is active.

7.7.7.14.2 Output and Reporting

When validation is enabled, the JSON output includes a `validationResults` section.

- Key metric: `invalidDataSampleCount` (the total number of messages that failed validation).
- Logging: Individual failure logs are capped at the first 5,000 invalid samples. Additional failures are counted in the metric but not logged individually.

> ***If validation is disabled, this section is omitted entirely.***

7.7.7.15 Enhanced Connection Establishment

ECE is an optional RDMA setup phase that aligns connection capabilities between the client and server before traffic begins.

When enabled, DOCA Perftest exchanges ECE parameters for each connection, leveraging the hardware-firmware negotiation that occurs during the Queue Pair (QP) transition from `RESET` to `INIT`.

7.7.7.15.1 High-Level Flow

The ECE process ensures both sides agree on supported features before establishing the connection.

1. The client queries its local ECE capabilities and sends them to the server via the control channel.
2. The server applies the client's proposal, transitions its QP to `INIT`, and queries the device for the final accepted ECE configuration.
3. The server sends the finalized ECE configuration back to the client.
4. The client applies the finalized configuration, transitions its QP to `INIT`, and validates the negotiated result.
5. Standard QP data exchange and `RTR` / `RTS` transitions proceed as usual.

7.7.7.15.2 ECE Configuration

Interface	Instruction
CLI	Add the `--use_ece` flag.
JSON	Set `"useEce": true` in the test configuration.

7.7.7.15.3 Limitations and Constraints

Driver Support	Currently supported only with the `libibverbs` driver (`ibv`)
Connection Type	Supported only on `RC` QPs

7.7.7.16 QP Hints

DOCA RDMA Verbs supports attaching opaque Congestion Control (CC) hints to Queue Pairs (QPs) for use by the Programmable Congestion Control (PCC) algorithm.

DOCA Perftest allows users to provide a binary hints file along with specific metadata (file size, vendor ID, and format ID). These parameters are passed directly to the PCC via the DOCA RDMA Verbs driver.

> This feature is available only when using the DOCA RDMA Verbs driver (`-r dv`).

7.7.7.16.1 Configuring QP Hints

You can configure QP hints via CLI or JSON.

- CLI – Pass a comma-separated list containing the file path and metadata using the `--cc_group_hints` flag.
- JSON input – Add the `ccGroupHints` object to your test configuration:

```
"ccGroupHints": {
    "filePath": "/path/to/hints.bin",
    "fileSize": 1024,
    "vendorId": 1,
    "formatId": 1
}
```

7.7.7.17 TPH

PCIe optimization providing hints to CPUs for cache management and reduced memory-access latency.

> Requires ConnectX-6+ hardware and a TPH-enabled kernel.

Parameters:

Option	Meaning
`--ph`	Processing hint: `0` = Bidirectional (default), `1` = Requester, `2` = Completer, `3` = High-priority completer
`--tph_core_id`	Target CPU core for TPH handling
`--tph_mem`	Memory type: `pm` = Persistent, `vm` = Volatile

Examples:

```
# Invalid: Core ID without memory type
doca_perftest -d mlx5_0 -n server-name --tph_core_id 0  # ERROR

# Invalid: Memory type without core ID
doca_perftest -d mlx5_0 -n server-name --tph_mem pm  # ERROR

# Valid: Both or neither
doca_perftest -d mlx5_0 -n server-name --ph 1  # OK (hints only)
doca_perftest -d mlx5_0 -n server-name --ph 1 --tph_core_id 0 --tph_mem pm  # OK (full config)
```

7.7.7.18 Warmup Configuration

By default, DOCA Perftest performs a time-based warmup before measuring results. Two warmup modes are available:

Option	JSON Field	Description
`-w seconds`	`warmupTimeSecs`	Duration-based warmup (default).
`--warmup_iterations count`	`warmupIterations`	Iteration-based warmup.

The two modes are mutually exclusive.

Constraints:

- Iteration-sync tests reject duration-based warmup; use `--warmup_iterations` instead.
- Latency tests reject iteration-based warmup; use `-w` instead.
- Data validation requires warmup to be explicitly disabled (`-w 0`).

7.7.7.19 End-margin

In multi-process or multi-node setups, processes may start at slightly different times. This means some processes finish measurement while others are still running, potentially causing the early finishers to stop sending traffic (which can affect the late starters' results).

The `--end_margin` option keeps traffic running for a specified number of seconds after measurement ends, ensuring all processes see consistent traffic throughout their measurement window.

Option	JSON Field	Description
`--end_margin secs`	`endMarginSecs`	Continue traffic for this many seconds after measurement.

Defaults:

- CLI single-process (no `--traffic_pattern`): `0` (disabled)
- CLI multi-process / `--traffic_pattern` / JSON mode: `2` seconds

End-margin applies to BW tests only and is not applicable for GPUNetIO. The reported bandwidth and iteration count reflect the measurement period only (end-margin traffic is excluded from results).

7.7.7.20 Live Bandwidth Reporting

The `--report_interval` (`-R`) flag enables periodic live bandwidth reporting during CLI bandwidth tests. When set, a scrolling log of throughput samples (Gb/s, Mpps) is printed at the configured interval.

```
# Print live BW every 2 seconds
doca_perftest -d mlx5_0 -n server-name -m bw -R 2
```

Default is `0` (disabled). Live reporting is supported in unidirectional, bidirectional, and dual-unidirectional modes.

7.7.7.21 QP Transport Timeout

The `--qp_timeout` flag (CLI) or `"qpTimeOut"` field (JSON) configures the transport-level timeout for each Queue Pair. This controls how long the QP waits before retransmitting an unacknowledged packet.

```
doca_perftest -d mlx5_0 -n server-name --qp_timeout 18
```

7.7.7.22 Multi-verb Operations

The `--operations` flag (CLI) or `"operations"` field (JSON) enables posting multiple Work Requests per iteration, each with a different RDMA verb and message size. This is useful for benchmarking mixed-verb workloads.

```
# Post a Write of 64KB followed by a Send of 4KB per iteration
doca_perftest -d mlx5_0 -n server-name --operations write:65536,send:4096
```

Each WR in the group gets its own verb, message size, and memory region.

7.7.8 Automation and CI Integration

DOCA Perftest provides several features designed for integration with automation frameworks, CI/CD pipelines, and regression testing systems.

For automated environments, do not parse the human-readable terminal output. The terminal format may change between versions to improve readability. Use the structured JSON output (via `--print_json_results` or the output file) as the stable programmatic API.

7.7.8.1 Performance Thresholds

Performance thresholds allow automated pass/fail validation of benchmark results. When a threshold is set, DOCA Perftest compares the measured results against the specified value and reports whether the test passed or failed.

If any test fails its threshold check, **doca_perftest** exits with return code 2 (instead of the normal 0). This enables CI systems to detect regressions by checking the process exit code.

Option	JSON Field	Unit	Validation Rule
--bw_threshold	**bwThreshold**	Gbit/sec	Test fails if average NIC bandwidth is below this value
--lat_threshold	**latThreshold**	usec	Test fails if the worst NIC median latency exceeds this value

Both thresholds default to **0** (disabled). The threshold and pass/fail status are included in the JSON output.

Examples:

```
# Bandwidth regression test: fail if below 90 Gbit/sec
doca_perftest -d mlx5_0 -n server-name -m bw --bw_threshold 90

# Latency regression test: fail if median exceeds 5 usec
doca_perftest -d mlx5_0 -n server-name -m lat --lat_threshold 5.0
```

7.7.8.2 Non-interactive Mode

The **--non_interactive** flag suppresses real-time progress indicators (spinners, progress bars, live table redraws). This is recommended when **doca_perftest** output is redirected to a file or pipe, or when a monitoring system reads **stdout**.

```
doca_perftest -d mlx5_0 -n server-name --non_interactive
```

7.7.8.3 Structured JSON Output to stdout

The **--print_json_results** flag (CLI) or **"printJsonOutput": true** field (JSON) prints the aggregated results summary to **stdout** in JSON format. This is the same schema used in the JSON output file.

This JSON output is treated as a stable API contract and will not change without a schema version bump. It is the recommended method for programmatic result consumption.

```
# CLI: print JSON results to stdout
doca_perftest -d mlx5_0 -n server-name --print_json_results

# JSON mode: add to scenario config
"printJsonOutput": true,
"outputFile": "results.json"
```

In JSON mode, **printJsonOutput** requires **outputFile** to be set.

7.7.8.4 Connection Matrix Report

JSON scenarios and CLI `--traffic_pattern` runs print a directed connection matrix for each test and message size by default. Rows are source endpoints and columns are destination endpoints, labelled `host:device`, or `host:device:gpu` when the traffic runs from GPU memory. Bandwidth cells use red for lower values and green for higher values; latency uses the opposite scale. When a threshold is configured, cells use pass/fail colors instead of the heatmap.

A test with more than 16 endpoints prints its best and worst 10 connections instead of a matrix, since a wider matrix no longer fits a terminal. The JSON output retains all connection results either way.

Set `"printConnectionMatrix": false` at the scenario level to suppress the report for every test, on a single test to suppress only that one, or pass `--disable_connection_matrix` in a CLI multi-node run.

7.7.9 Running DOCA Perftest on BlueField-3 Devices

DOCA Perftest is capable of generating traffic from either the x86 host or the BlueField Arm cores, determined entirely by the input JSON configuration.

7.7.9.1 MPI Network Configuration

When launching `doca_perftest` from the server (regardless of whether the traffic originates from the x86 host or the BlueField), it is recommended to explicitly specify the MPI TCP network interface.

Add the subnet that connects the management server and the BlueField devices to the `mpiTcpNetworkInterfaces` field in your JSON input (e.g., `"mpiTcpNetworkInterfaces": "10.7.8.0/24"`).

7.7.9.2 Traffic Originating from x86 Host (Server)

In this mode, traffic is generated by the x86 server. The RDMA device on the host (e.g., `mlx5_0`) performs DMA operations directly to/from host DRAM via PCIe.

Data path:

- Path: NIC ↔ PCIe ↔ Host Memory
- Bottlenecks: Performance is influenced by PCIe bandwidth and host CPU behavior, in addition to the network link and NIC capabilities.

JSON configuration:

- `hostName`: Set to the x86 server hostname.
- `deviceName`: Set to the RDMA device on the server (e.g., `mlx5_0`).

7.7.9.3 Traffic Originating from BlueField (Arm Cores)

In this mode, traffic is generated by the BlueField Arm cores, even if the test is launched from the x86 server. The RDMA device on the BlueField (e.g., `p0`, `p1`, `mlx5_2`, `pf0vf0`, depending on the operating mode) performs DMA operations to/from the BlueField's on-board DDR.

Data path:

- Path: NIC ↔ DPU DDR (No PCIe hop)
- Bottlenecks: Performance is typically limited by the network link, NIC, and DPU DDR bandwidth. The PCIe bus is not involved in the data path.

JSON Configuration:

- **hostName**: Set to the BlueField hostname.
- **deviceName**: Set to the RDMA device on the BlueField (e.g., **p0**, **mlx5_2**).

Device naming conventions may vary depending on the BlueField operating mode.

7.7.10 SLURM

DOCA Perftest integrates seamlessly with SLURM job schedulers, leveraging MPI for multi-node orchestration within SLURM allocations.

The following is a basic usage example with **salloc**:

1. Allocate nodes via SLURM (e.g., **salloc -N8**).
2. Update the JSON to include the allocated nodes. Simple bisection example:

```
"testNodes": [  {"hostName": "rack1-[01-04]", "deviceName": "mlx5_0"},
                {"hostName": "rack2-[05-08]", "deviceName": "mlx5_0"} ],
"trafficPattern": "BISECTION"
```

3. Run **doca_perftest** with the updated JSON:

```
doca_perftest -f <updated-json>
```

7.7.11 GPUNetIO Engine (GPU-initiated RDMA)

DOCA Perftest supports offloading the entire RDMA data-path to an NVIDIA GPU using DOCA GPUNetIO.

When the **gpunetio** engine is selected, WQE preparation, doorbell ringing, and CQ polling all execute inside CUDA kernels on the GPU. The host CPU is only involved during setup, teardown, and result collection.

This is useful for benchmarking RDMA as experienced by the GPU itself, which is the relevant data-path for GPU-centric workloads like distributed AI training.

Key points:

- Requires DOCA Verbs driver (**-r dv**) and CUDA memory (**-M cuda**)
- Supports Write verb only (**-v write**), both BW and Latency metrics
- Unidirectional traffic only (no bidirectional)
- Requires **doca-gpunetio** package, CUDA Toolkit, and a Volta+ GPU

Basic usage:

```
# CLI
doca_perftest -d mlx5_0 -n server-name -M cuda -G 0 -r dv --engine gpunetio
```

```
"engine": "gpunetio",
"memoryType": "cuda",
"cudaDeviceId": 0,
"rdmaDriver": "dv",
"verb": "write",
"metric": "bw"
```

For more information about DOCA GPUNetIO, refer to the DOCA GPUNetIO Programming Guide.

7.8 DOCA Pipeline Language Developer Tools

These pages describe the NVIDIA DOCA Pipeline Language (DPL) Developer Tools, which are used to inspect and debug DPL programs on NVIDIA® BlueField® DPUs.

> For instructions on creating, compiling, and deploying DPL programs, refer to the DOCA Pipeline Language Service Guide.

7.8.1 Introduction

NVIDIA provides a suite of developer tools to assist with the validation and debugging of DPL applications:

Tool	Description
DPL Compiler	Converts DPL source code into a binary format that can be loaded by the DPL Runtime Daemon (`dpl_rtd`).
P4 Runtime Shell	Open-source control plane manager implementing the P4Runtime API.
DPL Nspect	Command-line interface (CLI) for inspecting DPL runtime state, including tables, keys, counters, and more.
DPL Debugger	Graphical interface for visualizing and debugging packet flow through the DPL program.
DPL Admin	CLI tool for dynamically modifying configuration items in the DPL Runtime Daemon.
DPL Runtime Controller SDK	SDK for building custom P4Runtime applications or integrating with DPL shared memory APIs.

All tools interact with the DPL Runtime Daemon (`dpl_rtd`) running on the BlueField DPU.

7.8.2 Prerequisites

To use the DPL developer tools, the following components must be set up:

1. DPL Runtime Service
    a. The DPL Runtime Service container must be deployed and running on the DPU. This service manages the DPL Runtime daemon.
    b. Ensure the `dpl_rtd` process is active and the DPU is configured to operate in DPU mode.
2. Starting a debug session requires the DPL program to be compiled for debugging:

a. Compile the DPL program using the DPL Compiler with the debug flag `-g` enabled.
b. Load the compiled binary into `dpl_rtd` using the P4Runtime controller.

> For further details, refer to Compiling DPL Applications.

7.8.3 Setup

The developer tools are included in the DPL Development Container, which is publicly available on NGC. For installation instructions, refer to the DPL Installation Guide.

7.8.4 Support and Feedback

To ask questions or provide feedback regarding DPL Developer Tools, visit the NVIDIA DOCA Enterprise Networking Forum.

You must register with the NVIDIA Developer Program to post on the forums. Registration is not required to view existing discussions. For help with registration, please refer to the FAQ.

7.8.5 DPL Nspect

This page describes the DPL Nspect tool, which is used to inspect DPL programs loaded onto the NVIDIA® BlueField® DPU.

7.8.5.1 1. Introduction

The DOCA Dependency Compatibility Policy is designed to provide customers with high velocity and flexibility during updates, enabling them to update specific layers of their system without requiring a full stack overhaul.

The primary objectives of this policy are to support decoupled updates for teams managing separate firmware and software stacks, allow for targeted upgrades on specific devices, minimize maintenance downtime by updating only necessary components, and guarantee investment protection by ensuring application continuity across platform releases.

This policy applies to all NVIDIA network device products and their associated software and firmware stacks.

7.8.5.2 2. Release Notes

7.8.5.2.1 2.1. New Features

- Externs Command: This new command replaces the per-extern commands (**counters** , **meters** , **tcpstates**) with a single **externs** command. Use the **types** argument to filter by extern type, including selecting only direct or indirect externs.
- Memory Extern: Added support for querying the **memory** extern type with the **externs** command.

- IPsec Extern: (BETA) Added support for querying the `ipsec` extern type with the `externs` command.
- Additional Results Hint: Commands that support batching and max results now show a hint when more results are available beyond what was returned, so you can decide whether to increase the batch size or max results accordingly.
- New Performance Stats:
    - Added per-table entry modification statistics (add / modify / delete counts and rates) to the `perf` command.
    - Added per-table ternary runtime-optimization (RTO) statistics to the `perf` command, reported when the table's cache is enabled.
- Filter Improvements: Added support for P4 objects alias names. You can now reference and filter P4 objects using their optional `p4_alias`.
- Nspect Textual UI App: (BETA) Introduced a new Textual-based interactive UI for `dpl_nspect`. Launch it with the `--tui` flag for an interactive experience. The traditional headless CLI mode remains available and is the default mode.

7.8.5.2.2 2.2. Fixes

- The `meter` extern's `initial_color` was not populated in related commands.
- Extern filtering by name of direct externs.
- `perf` command truncating long numeric values with an ellipsis.
- `perf` command failing when running against an RT Service with Packet I/O disabled.
- Race condition that could cause intermittent query failures when new entries were being added concurrently.

7.8.5.2.3 2.3. Other

- Replaced `pickle` with `msgpack` package for data serialization to address a security concern.
- Log files are now automatically rotated to prevent unbounded disk usage, and the verbosity level remains configurable.

7.8.5.3 3. DPL Nspect Execution

To view basic usage information, run the `dpl nspect` without arguments. For a detailed usage guide, use the `-h` or `--help` flags. You can also view command-specific help by running `dpl nspect [command] -h`.

7.8.5.3.1 3.1. Prerequisites and Configuration

- Server Connection: DPL Nspect requires a connection to a host (local or remote) where the DPL Nspect Server is running as part of the DPL Runtime Service.
- Address Configuration: The host address and port can be configured in two ways:
    a. Directly using the `-a` / `--address` argument.
    b. By setting the `DPL_NSPECT_SERVER_ADDRESS` environment variable (e.g., `DPL_NSPECT_SERVER_ADDRESS=127.0.0.1:9560`).

> ⚠ Hostnames are resolved at runtime (e.g., **localhost** resolves to **127.0.0.1**).

- Data Directory: Configuration and logs are stored in the Data directory.
- Refer to the "DPL Service Configuration" for a TLS setup example

7.8.5.3.2 3.2. Command Usage

```
usage: dpl_nspect [-h] [-v] [-nh] [-ll] -a ADDRESS [-t TIMEOUT] [--tui] [--cacert PATH_TO_CA_CERT] [--cert
PATH_TO_CERT]
                  [--private-key PATH_TO_KEY]
                  {system-info,devices,tables,query,externs,graph,config,hw-steering,debug,perf} ...

options:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  -nh, --no-hints       Disable hints for further information.
                        You may set the DPL_NSPECT_DISABLE_HINTS environment variable instead of using this flag:
                        "export DPL_NSPECT_DISABLE_HINTS=1" (default: False)
  -ll, --low-level      Display low level internal information. (default: False)
  -a ADDRESS, --address ADDRESS
                        The dpl_nspect server socket address of the form IPv4/IPv6* [address][:port]
                        IPv6*: To use with a port the address must be enclosed in literal square brackets.
                        Example: ipv6:[2607:f8b0:400e:c00::ef]:9560 or ipv6:[::]:9560
                        The address/port can be assigned to and read from the environment variable:
DPL_NSPECT_SERVER_ADDRESS. (default: None)
  -t TIMEOUT, --timeout TIMEOUT
                        [Optional] gRPC channel connectivity timeout. (default: 0)
  --tui                 Run the application in TUI mode. (default: False)

commands:
  For further info run dpl_nspect <command> -h

  {system-info,devices,tables,query,externs,graph,config,hw-steering,debug,perf}
    system-info         Display system information.
    devices             List the configured virtual devices.
    tables              List tables details.
    query               List table entries, keys and actions of the loaded program.
    externs             List externs details and values.
    graph               Generate a graph using DOT language.
                        The generated graph can be rendered using DOT renderers, such as online DOT graphics
visualizer websites.
    config              Get/Set the device configuration.
    hw-steering         Display HW Steering rules dump.
    debug               Start a debug recording session.
    perf                Display live performance monitor statistics.

Authentication:
  Mutually inclusive arguments for secure connection using SSL authentication

  --cacert PATH_TO_CA_CERT
                        CA certificate path for secure connections (default: None)
  --cert PATH_TO_CERT   Client certificate path for mutual authentication (default: None)
  --private-key PATH_TO_KEY
                        Client private key path for mutual authentication (default: None)

Data directory: /root/.local/share/dpl_devtools
```

7.8.5.4 4. DPL Nspect Commands and Flags

The following sections describe specific commands and their outputs.

> ⚠ The DPL program **hello_packet.p4** (referenced at the end of this section) was used to generate the examples below.

> ✓ Most commands support the **--json** argument to output data in JSON format, facilitating integration with automated scripts.

7.8.5.4.1 4.1. system-info

```
dpl nspect system-info
```

Displays system information.

Example output:

System Information

Key	Value
Linux OS version	"Ubuntu 24.04.4 LTS"
DPL Runtime Service version	Not found
DOCA version	3.4.0072-1
RDMA Core Version	2604.0.5-1
OFED Version	OFED-internal-26.04-0.4.0:
Debug Schema Version	Device ID: 1000, doctype: /nvp4c/devtools/v20, target: doca-bf3

7.8.5.4.2 4.2. devices

```
dpl nspect devices
```

Lists all configured DPL devices, displaying a table for each device.

- Title: Contains program details only if a program is loaded.
- Interface: The interface name as defined in the DP device configuration file.
- ID: The port ID as defined in the DPL device configuration file.
- IB: The physical InfiniBand port ID used for the DPL port ID.

Example output:

Device 1000

Program: hello_packet, Loaded At: 2026-04-23 08:56:04, Load Duration: 0.159s

Ports

Interface	ID	IB	REG_C0	MAC
ens6f0np0	0	50	0x10000/0xffff0000	00:00:00:00:00:00
ens6f0r0	1	2	0x30000/0xffff0000	00:00:00:00:00:00
ens6f0r1	2	3	0x40000/0xffff0000	00:00:00:00:00:00
ens6f0r2	3	4	0x50000/0xffff0000	00:00:00:00:00:00
ens6f0r3	4	5	0x60000/0xffff0000	00:00:00:00:00:00
ens6f0r4	5	6	0x70000/0xffff0000	00:00:00:00:00:00
ens6f0r5	6	7	0x80000/0xffff0000	00:00:00:00:00:00
ens6f0r6	7	8	0x90000/0xffff0000	00:00:00:00:00:00
ens6f0r7	8	9	0xa0000/0xffff0000	00:00:00:00:00:00

Configuration

Key	Value
P4 Controller Port ID	9876
FW Version	32.47.0218
HW Model	BlueField-3
Multiport eSwitch	Disabled

7.8.5.4.3 4.3. tables

```
dpl nspect tables [--json]
```

Lists all tables defined in the DPL program.

- ID: The P4 runtime ID of the object as defined in the **p4info.txt** file.
- Direct Externs: Lists the names of all direct extern instances defined for the table (e.g., direct counter, direct meter).

Output example:

hello_packet.forward_table - 50144908

Match Fields			Actions			Direct Externs	Entries Count	Source Location	Supports Timeout
ID	Name	Type	ID	Name	Params				
1	headers.ethernet.src_addr	exact	19510525	hello_packet.drop		hello_packet.hello_direct_counter hello_packet.hello_direct_meter	6	hello_packet.p4:45:11	False
			19237259	hello_packet.forward	ID: 1, Name: port, Size: 32				
			21257015	NoAction					

The **--json** flag provides the output in JSON format. For example:


```
[
  {
    "title": {
      "Table ID": 40048916,
      "Table Name": "tbl_drop"
    },
    "rows": [
      {
        "Keys": {
          "title": {},
          "rows": []
        },
        "Actions": {
          "title": {},
          "rows": [
            {
              "ID": "19510525",
              "Name": "hello_packet.drop",
              "Params": ""
            }
          ]
        },
```

```
        "Direct Externs": [],
        "Entries Count": "1",
        "Source Location": "None",
        "Supports Timeout": "False"
      }
    ]
  },
  {
    "title": {
      "Table ID": 50144908,
      "Table Name": "hello_packet.forward_table"
    },
    "rows": [
      {
        "Keys": {
          "title": {},
          "rows": [
            {
              "ID": "1",
              "Name": "headers.ethernet.src_addr",
              "Type": "exact"
            }
          ]
        },
        "Actions": {
          "title": {},
          "rows": [
            {
              "ID": "19510525",
              "Name": "hello_packet.drop",
              "Params": ""
            },
            {
              "ID": "19237259",
              "Name": "hello_packet.forward",
              "Params": {
                "title": {},
                "rows": [
                  {
                    "ID": "1",
                    "Name": "port",
                    "Size": "32"
                  }
                ]
              }
            },
            {
              "ID": "21257015",
              "Name": "NoAction",
              "Params": ""
            }
          ]
        },
        "Direct Externs": [
          "hello_packet.hello_direct_counter",
          "hello_packet.hello_direct_meter"
        ],
        "Entries Count": "6",
        "Source Location": "hello_packet.p4:46",
        "Supports Timeout": "False"
      }
    ]
  }
]
```


7.8.5.4.4 4.4. query

```
dpl nspect query
```

Lists all entries currently contained in each table.

- Entry ID: The runtime ID assigned to the entry at creation time (not a P4 context ID).
- Keys: If no key is displayed, it indicates a default entry.
- Extern entries: Displays data for extern entries associated with the table entry (result of an explicit extern call within the Action).

Example output:

hello_packet.forward_table - 50144908

Entry ID [Priority]	Match	Action	Externs
5[0]	headers.ethernet.src_addr == 0xbbbbbbbbbb (exact)	NoAction()	
4[0]	headers.ethernet.src_addr == 0xaaaaaaaaaa (exact)	NoAction()	
3[0]	headers.ethernet.src_addr == 0xdddddddddd (exact)	hello_packet.drop()	hello_packet.hello_counter[1] = (packets=0, bytes=0)
2[0]	headers.ethernet.src_addr == 0x2222222222 (exact)	hello_packet.forward(port=0x2)	hello_packet.hello_direct_counter[2] = (packets=0, bytes=0)
1[0]	headers.ethernet.src_addr == 0x1111111111 (exact)	hello_packet.forward(port=0x1)	hello_packet.hello_direct_counter[1] = (packets=0, bytes=0)
0[0]		hello_packet.forward(port=0x3)	hello_packet.hello_direct_counter[0] = (packets=0, bytes=0)

7.8.5.4.5 4.5. externs

```
dpl nspect externs
```

Lists all externs in the DPL program (one table per extern instance???).

- Indirect Externs: First column shows Index.
- Direct Externs: First column shows Table Entry ID (corresponds to the ID seen in the **query** command).

NvDirectCounter(Pkts/Bts) hello_packet.hello_direct_counter - 320479669

Table Entry ID	Packets	Bytes
5	0	0
4	0	0
3	0	0
2	0	0
1	0	0
0	0	0

NvCounter(4, Pkts/Bts) hello_packet.hello_counter - 307179965

Index	Packets	Bytes
0	0	0
1	0	0

2	0	0
3	0	0

NvDirectMeterPeakTrTCM(128), Pkts hello_packet.hello_direct_meter - 359616562

Table Entry ID	CIR	CBS	PIR	PBS	Initial Color
5	1	2	3	4	UNDEFINED
4	1	2	3	4	UNDEFINED
3	1	2	3	4	UNDEFINED
2	1	2	3	4	UNDEFINED
1	1	2	3	4	UNDEFINED
0	1	2	3	4	UNDEFINED

NvTcpState(4) hello_packet.hello_tcp - 2319426381

Index	Last Direction	Last ACK Number	Last SEQ Number
0	ORIGINAL	0	0
1	ORIGINAL	0	0
2	ORIGINAL	0	0
3	ORIGINAL	0	0

The **`--types`** argument allows querying specific extern categories (counter, meter, tcp_state, memory) and/or scope (direct for table-bound, indirect for standalone).

```
dpl nspect externs --types counter,meter,direct
```

NvDirectCounter(Pkts/Bts) hello_packet.hello_direct_counter - 320479669

Table Entry ID	Packets	Bytes
5	0	0
4	0	0
3	0	0
2	0	0
1	0	0
0	0	0

NvDirectMeterPeakTrTCM(128), Pkts hello_packet.hello_direct_meter - 359616562

Table Entry ID	CIR	CBS	PIR	PBS	Initial Color
5	1	2	3	4	UNDEFINED
4	1	2	3	4	UNDEFINED
3	1	2	3	4	UNDEFINED
2	1	2	3	4	UNDEFINED
1	1	2	3	4	UNDEFINED
0	1	2	3	4	UNDEFINED

7.8.5.4.6 4.6. graph

```
dpl nspect --low-level graph
```

Generates a DOT graph representing all potential packet paths within the pipeline.

Example output:

```
digraph hello_packet_pipeline {
    graph [compound=true nodesep=0.1]
    node [height=0 width=0]
    // Legend
    subgraph cluster_legend {
        graph [fontsize="6pt" label=Legend]
        control [label=Control color=blue fontsize="6pt" id=control shape=rectangle]
        table [label=Table color=orange fontsize="6pt" id=table shape=rectangle]
        ll_table [label=<<B>Low Level Table ID</B><BR/>[Origin(s)]> fontsize="6pt" id=ll_table shape=diamond]
        legend_invis [height=0 id=legend_invis shape=point style=invis width=0]
        ll_root_table [label=<<B>Root LL Table</B>> color=green fontsize="6pt" id=ll_root_table shape=diamond]
        ll_unreachable_table [label=<<B>Unreachable LL Table</B>> color=red fontsize="6pt" id=ll_unreachable_table
shape=diamond]
        default_rule_invis_tail [fontsize="6pt" height=0 id=default_rule_invis_tail shape=point style=invis
width=0]
        default_rule_invis_head [fontsize="6pt" height=0 id=default_rule_invis_head shape=point style=invis
width=0]
```

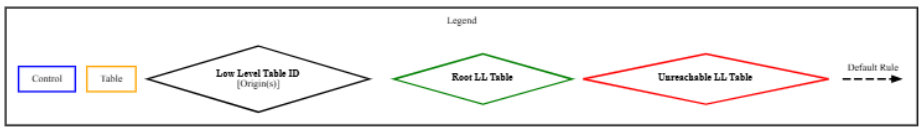
```
        default_rule_invis_tail -> default_rule_invis_head [label="Default Rule" arrowsize="0.5px" fontsize="6pt"
id="default_rule_invis_tail-default_rule_invis_head" style=dashed]
        subgraph rank_0 {
            rank=same
            control [id=control]
            default_rule_invis_head [id=default_rule_invis_head]
            default_rule_invis_tail [id=default_rule_invis_tail]
            ll_root_table [id=ll_root_table]
            ll_table [id=ll_table]
            ll_unreachable_table [id=ll_unreachable_table]
            table [id=table]
        }
    }
    // Pipeline Stage: main
    subgraph cluster_hello_packet {
        graph [color=blue label=hello_packet shape=rectangle]
        legend_invis [height=0 id=legend_invis shape=point style=invis width=0]
        "51[4]" [label=<<B>51[4]</B><BR/>[hello_packet]> color=green id="51[4]" shape=diamond]
        "42[2]" [label=<<B>42[2]</B><BR/>[hello_packet, hello_packet.forward_table]> id="42[2]" shape=diamond]
        DROP [fontcolor=red id=DROP shape=box]
        "47[3]" [label=<<B>47[3]</B><BR/>[hello_packet]> id="47[3]" shape=diamond]
        legend_invis -> "51[4]" [id="legend_invis-51[4]" style=invis]
        "51[4]" -> "42[2]" [id="51[4]-42[2]"]
        "42[2]" -> DROP [id="42[2]-DROP"]
        "51[4]" -> "47[3]" [id="51[4]-47[3]" style=dashed]
        "47[3]" -> DROP [id="47[3]-DROP"]
        subgraph rank_0 {
            rank=same
            "51[4]" [id="51[4]"]
        }
        subgraph rank_1 {
            rank=same
            "42[2]" [id="42[2]"]
            "47[3]" [id="47[3]"]
        }
        subgraph rank_2 {
            rank=same
            DROP [id=DROP]
        }
    }
}
```

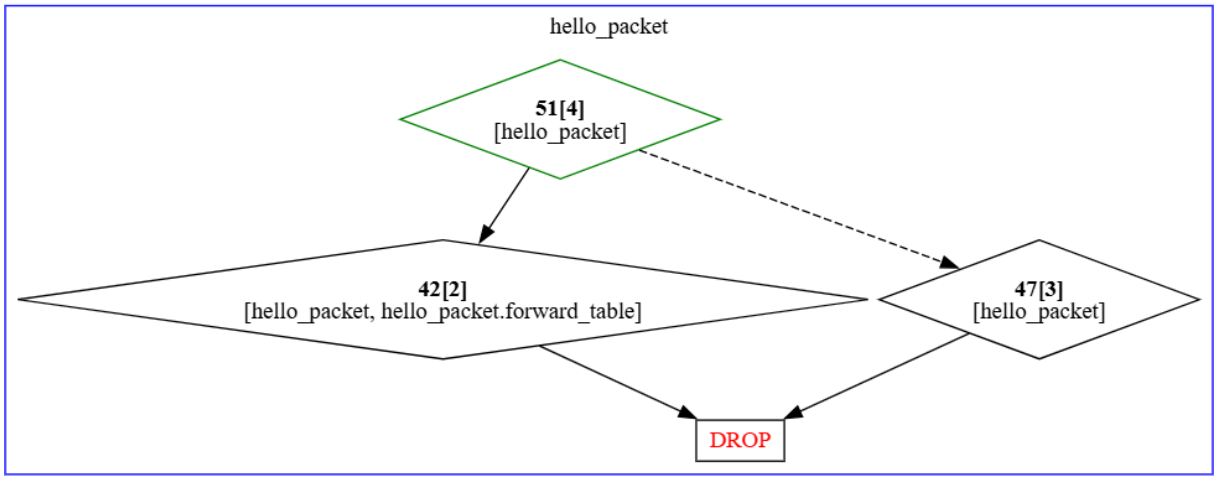
```
dpl nspect --low-level graph --url
```

The **`--url`** flag generates an HTTP URL link to render the graph using GraphvizOnline.

Legend:

- Low Level Table (LL Table): Refers to the internal HWS Table object created from loading the DPL program.
- Root LL Table: The starting point of the generated pipeline.





7.8.5.4.7 4.7. debug

Starts a debug session. Packets sent during this session are traced and recorded to an output archive, which can be inspected using the DPL Debugger.

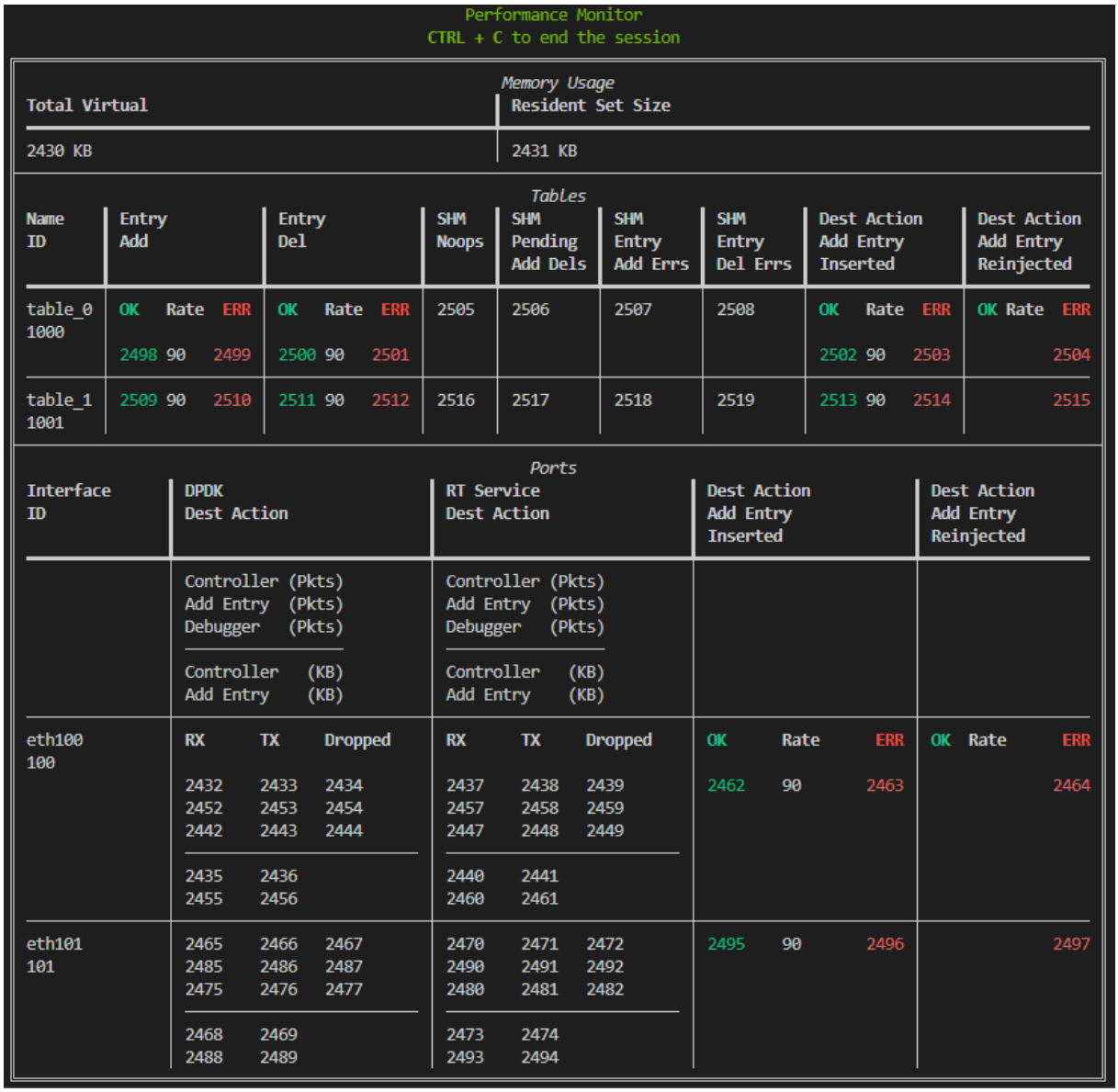
```
dpl nspect debug -o hello_packet.tar.gz
```

7.8.5.4.8 4.8. perf

```
dpl nspect perf
```

Displays real-time performance statistics.

Example output:



The following subsections break down the statistics provided in the example output.

7.8.5.4.8.1 4.8.1. Memory Usage Statistics

Displays the real-time memory consumption of the DPL Runtime Service.

Metric	Description
Total Virtual Memory	The total amount of virtual memory used by the service.
Resident Set Size	The amount of physical memory (RAM) currently being used (in KB).

7.8.5.4.8.2 4.8.2. Table Entry Operations Statistics

Tracks operations for each DPL table (one row per table).

Category	Column	Description	Notes
Entry Add/Del	OK	Total number of successful additions or deletions	Tracks *all* table entry operations.
	ERR	Total number of failed additions or deletions	
	Rate	The difference in successful operations between the current and previous sample	
SHM	Noops	Count of entries where a delete was requested before the addition was fully processed	**Verbose mode only.** Tracks *entries*. Only for SHM-enabled tables
	Pending Add Dels	Count of delete operations received for entries that are still pending hardware confirmation	**Verbose mode only.** Tracks *operations*
	Entry Add/Del Errs	Total number of failed SHM entry additions or deletions	**Verbose mode only.** Tracks *operations*
Dest Action Add Entry	Inserted	Successful table entry additions triggered specifically by the *Add Entry* action	**Verbose mode only.** Tracks *table entries*
	Reinjected	Total number of *failed* reinjection attempts (ERR column).	**Verbose mode only.** Tracks *packets*. The OK/Rate columns are always empty

7.8.5.4.8.3 4.8.3. Port Statistics Statistics

Tracks packet flow through destination actions (one row per Port Interface ID).

Layer	Row/ Action	Description	Notes
DPDK Dest Action	N/A	Packet flow statistics at the hardware (DPDK) layer	
RT Service Dest Action	N/A	Packet flow statistics at the application (RT Service) layer	
Action Types	Controller	Flow statistics for packets destined for the Controller	
	Add Entry	Flow statistics for packets destined for the Add Entry action	Same metrics as the *Tables* section, but tracked per port
	Debugger	Flow statistics for packets destined for the Debugger	Only displays Packet Counts (no KB)
Flow Direction	RX / TX	Packets received or transmitted in this layer/ action	Displays both Packet Counts and KB
	Dropped	Packets dropped in this layer/action	Only displays Packet Counts (no KB)

7.8.5.4.9 4.9. hello_packet.p4

```
/*
```

```
 * SPDX-FileCopyrightText: Copyright (c) 2025 NVIDIA CORPORATION & AFFILIATES. All rights reserved.
 * SPDX-License-Identifier: LicenseRef-NvidiaProprietary
 *
 * NVIDIA CORPORATION, its affiliates and licensors retain all intellectual
 * property and proprietary rights in and to this material, related
 * documentation and any modifications thereto. Any use, reproduction,
 * disclosure or distribution of this material and related documentation
 * without an express license agreement from NVIDIA CORPORATION or
 * its affiliates is strictly prohibited.
 */

#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>

/*
 * This basic application demonstrates a simple match/action pipeline using NVIDIA
 * DOCA P4 Target Architecture.
 * - Match on destination MAC address
 * - Forward or drop the packet
 */

control hello_packet(
    inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvCounter(4, NvCounterType.PACKETS_AND_BYTES) hello_counter;
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) hello_direct_counter;
    NvDirectMeterPeakTrTCM(NvMeterUnits.PACKETS, 1, 2, 3, 4) hello_direct_meter;
    NvTcpState(4) hello_tcp;

    action drop() {
        hello_counter.count(1);
        nv_send_debug_pkt();
        nv_drop();
    }

    action forward(bit<32> port) {
        hello_direct_counter.count();
        nv_send_to_port(port);
    }

    table forward_table {
        key = {
            headers.ethernet.src_addr : exact;
        }
        actions = {
            drop;
            forward;
            NoAction;
        }
        default_action = forward(3);
        direct_meter = hello_direct_meter;
        direct_counter = hello_direct_counter;
        const entries = {
            (48w0x001111111111) : forward(1);
            (48w0x002222222222) : forward(2);
            (48w0x00dddddddddd) : drop();
            (48w0x00aaaaaaaaaa) : NoAction();
            (48w0x00bbbbbbbbbb) : NoAction();
        }
    }
    apply {
        hello_counter.count(0);
        if (headers.ipv4.isValid()) {
            forward_table.apply();
        }
        drop();
    }
}

NvDocaPipeline(
    nv_fixed_parser(),
    hello_packet()
) main;
```

7.8.6 DPL Debugger

This section describes the DPL Debugger Tools, used to debug pipeline packets that DPL programs are processing in NVIDIA® BlueField®.

> ⚠ The debugger functions only with DPL programs compiled in these debug mode.

7.8.6.1 1. Introduction

The DPL Debugger is a graphical user interface (GUI) application used to debug network packets processed by DPL programs on NVIDIA® BlueField® devices. It allows developers to trace packet flows throughout a DPL program's pipeline, inspecting detailed states at each stage.

> ⚠ For details, see the documentation on the extern function nv_send_debug_pkt.

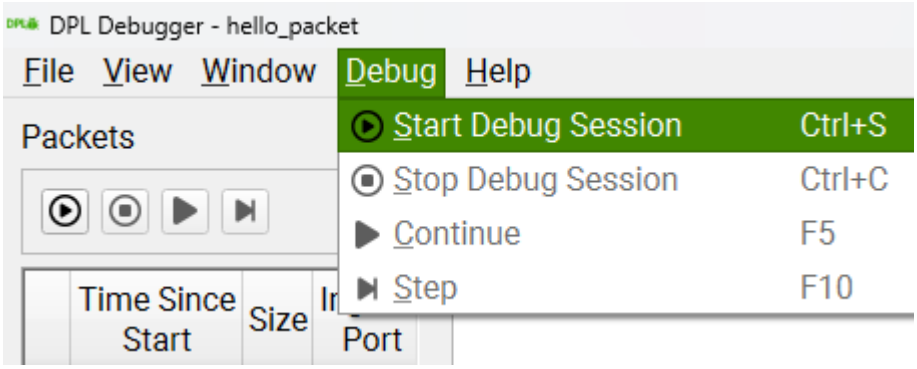
7.8.6.2 2. Launching the Debugger

You can launch the DPL Debugger to either run a live debug session or analyze a pre-recorded session file. Running `dpl debugger` from the command line with no arguments opens the standard GUI.

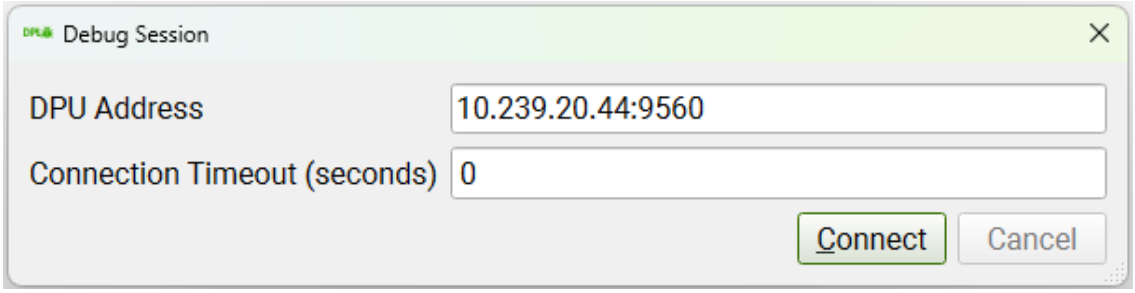
7.8.6.2.1 2.1. Option A: Live Debug Session

Running a live session allows you to process, inspect, and reinject packets at runtime.

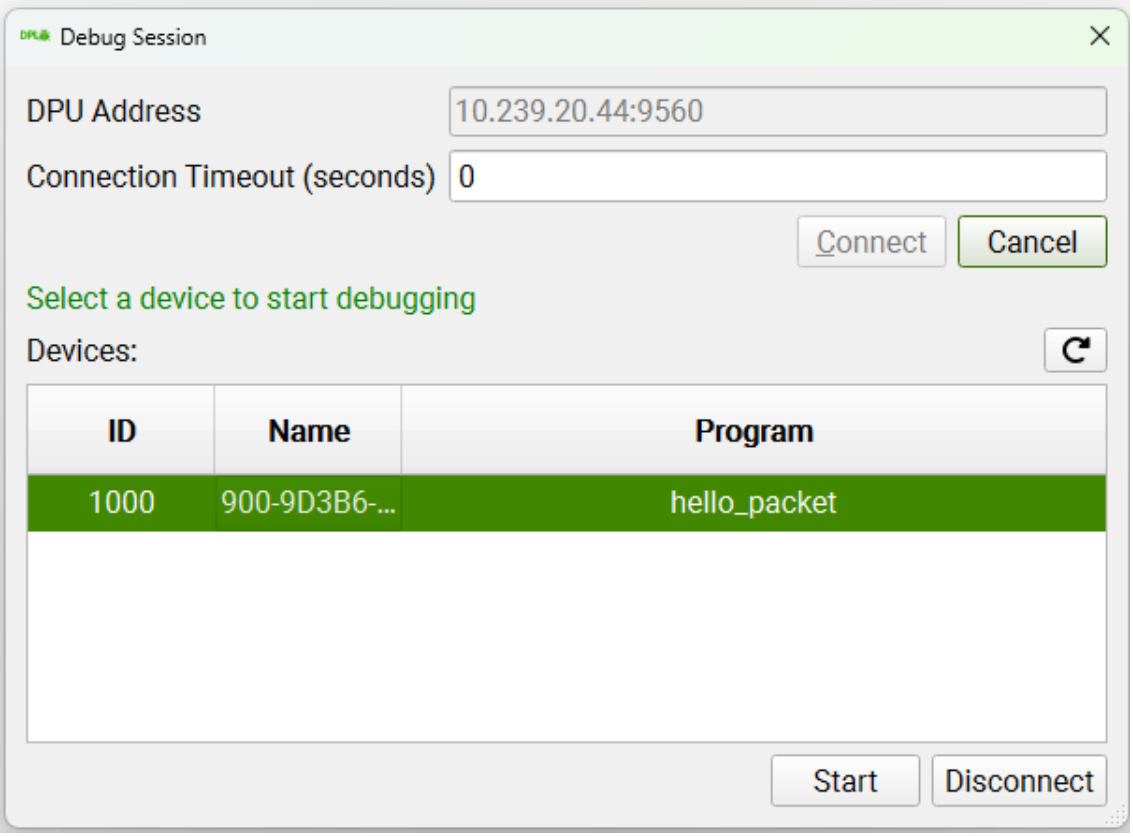
1. Click the Start Debug Session button, press Ctrl+S, or navigate to Debug → Start Debug Session.



2. Enter the DPU address, port (default is 9560), and connection timeout (use 0 for no timeout), then click Connect.



3. Select the desired device and click **Start**.



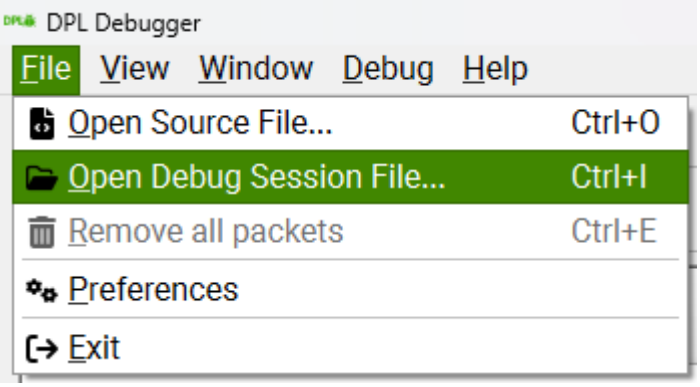
7.8.6.2.2 2.2. Option B: Pre-recorded Debug Session

Pre-recorded sessions are `.tar.gz` archives generated using the **`nspect debug`** command.

- Via CLI: Specify the archive path when launching the tool:

```
dpl debugger hello_packet.tar.gz
```

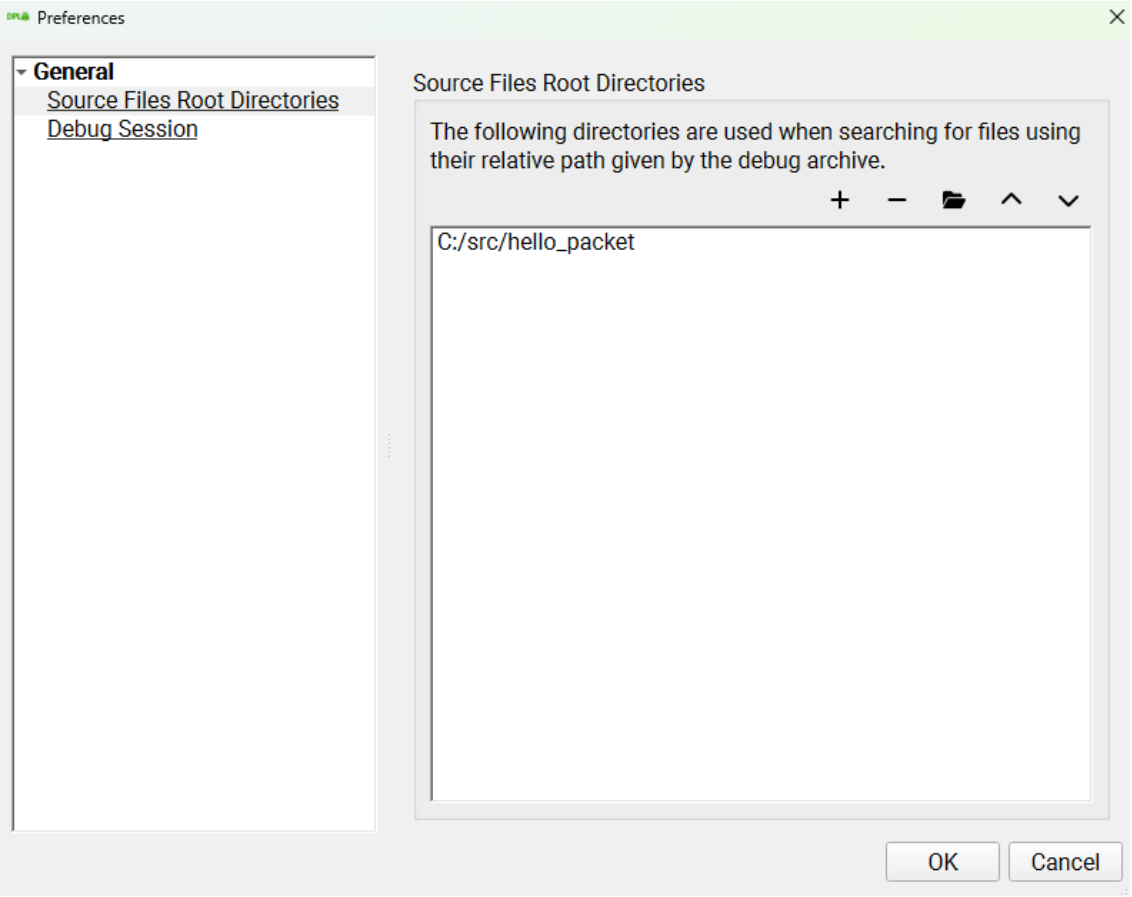
- Via GUI: Click **File** → **Open Debug Session File** and select the archive.



7.8.6.3 3. Setting Source Directories

To ensure the debugger can display the relevant code when a debug point is hit, configure the location of your DPL program sources.

1. Navigate to File → Preferences.



2. Set the search paths for your DPL source files.

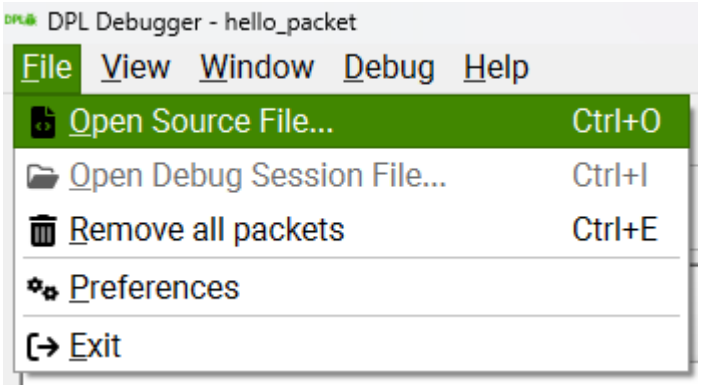
7.8.6.4 4. Debug Workflow

7.8.6.4.1 4.1. Enabling Debug Points

When a live session starts, all debug points in the program are disabled by default. Packets flow through the pipeline unaffected until a debug point is enabled.

To enable a debug point:

1. Open the source file within the debugger (File → Open Source File).



2. Click the left margin next to the desired line, or place your caret on the line and press F9.

> If a line does not have an associated debug point, the next available debug point will be enabled instead.

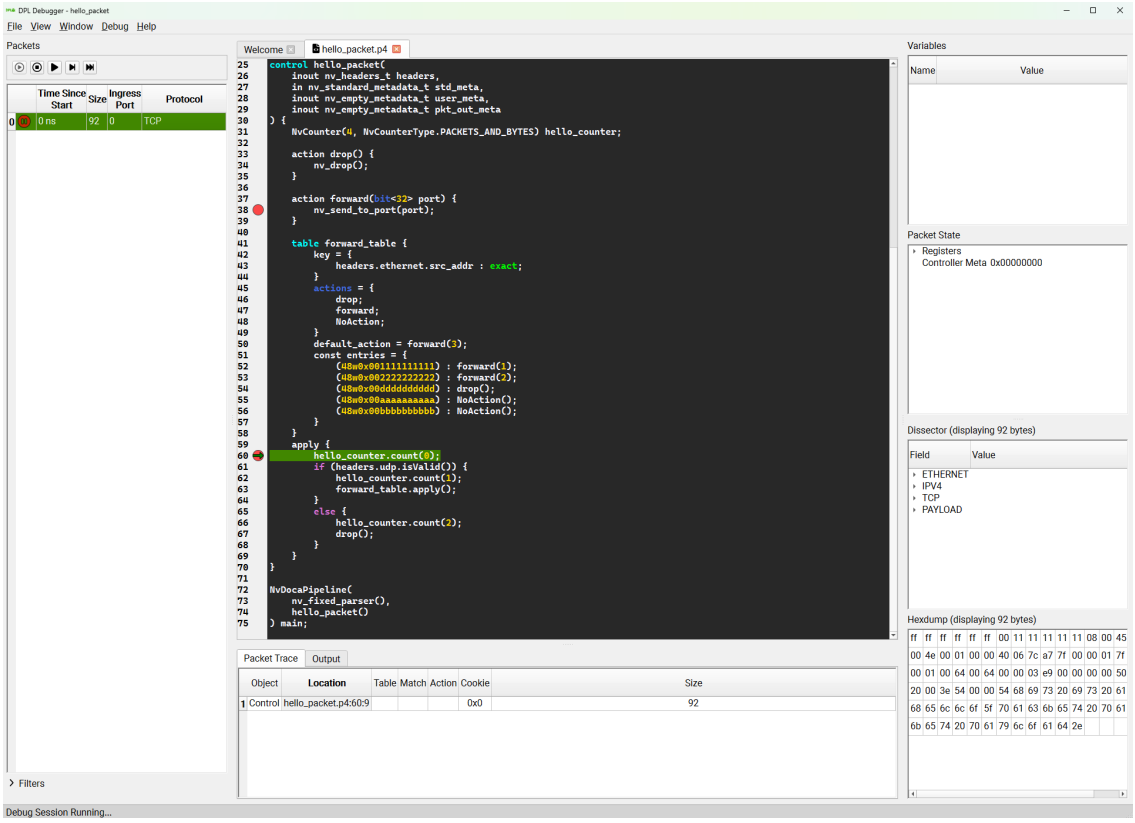
> The available debug points to enable depend on which debug mode you compiled the program with.

7.8.6.4.2 4.2. Inspecting and Editing Packets

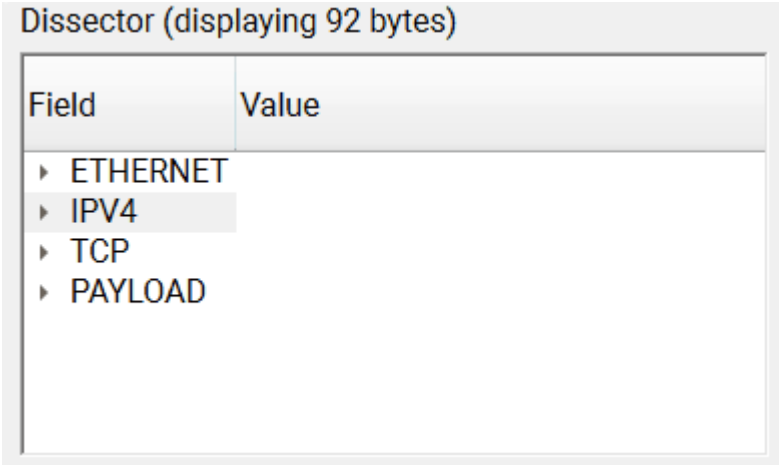
When a packet hits an enabled debug point, it is sent to the debugger.

- A red icon indicates the packet is currently paused by the debugger.

- A **green arrow** indicates the specific debug point where the packet is halted.



Once paused, you can edit the packet's content using either the **Dissector** or the **Hexdump** views. For example, you can edit the packet's protocol from TCP to UDP directly within the Dissector.



7.8.6.4.3 4.3. Advancing the Pipeline

After inspecting or editing a paused packet, inject it back into the pipeline using one of the following commands under the Debug menu:

- Step: Advances the packet to the very next line, even if no breakpoint is explicitly enabled there. This creates a new row in the Packet Trace widget.

- Continue: Advances the packet until it hits the next enabled breakpoint. If no further breakpoints exist, the packet finishes its pipeline and the pause icon disappears.
- Continue with trace: Functions like **Continue**, but records and displays the trace of all debug points the packet passed through along the way.

The packet in the **Packets** list correlates to the first row in the **Packet Trace** list. Changes made to the packet (e.g., size or protocol) will only be reflected in the subsequent row of the Packet Trace after reinjection.

7.8.6.4.4 4.4. Example Debugging Scenario

The following example demonstrates tracing a simple match/action pipeline.

7.8.6.4.4.1 4.4.1. The DPL Program (`hello_packet.p4`)

This program matches on a destination MAC address and either forwards or drops the packet. It was compiled with the `debug-mode=auto` flag.

```
#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>

control hello_packet(
    inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvCounter(4, NvCounterType.PACKETS_AND_BYTES) hello_counter;

    action drop() {
        nv_drop();
    }

    action forward(bit<32> port) {
        nv_send_to_port(port);
    }

    table forward_table {
        key = {
            headers.ethernet.src_addr : exact;
        }
        actions = {
            drop;
            forward;
            NoAction;
        }
        default_action = forward(3);
        const entries = {
            (48w0x001111111111) : forward(1);
            (48w0x002222222222) : forward(2);
            (48w0x00dddddddddd) : drop();
            (48w0x00aaaaaaaaaa) : NoAction();
            (48w0x00bbbbbbbbbb) : NoAction();
        }
    }
    apply {
        hello_counter.count(0);
        if (headers.udp.isValid()) {
            hello_counter.count(1);
            forward_table.apply();
        }
        else {
            hello_counter.count(2);
            drop();
        }
    }
}

NvDocaPipeline(
    nv_fixed_parser(),
    hello_packet()
) main;
```

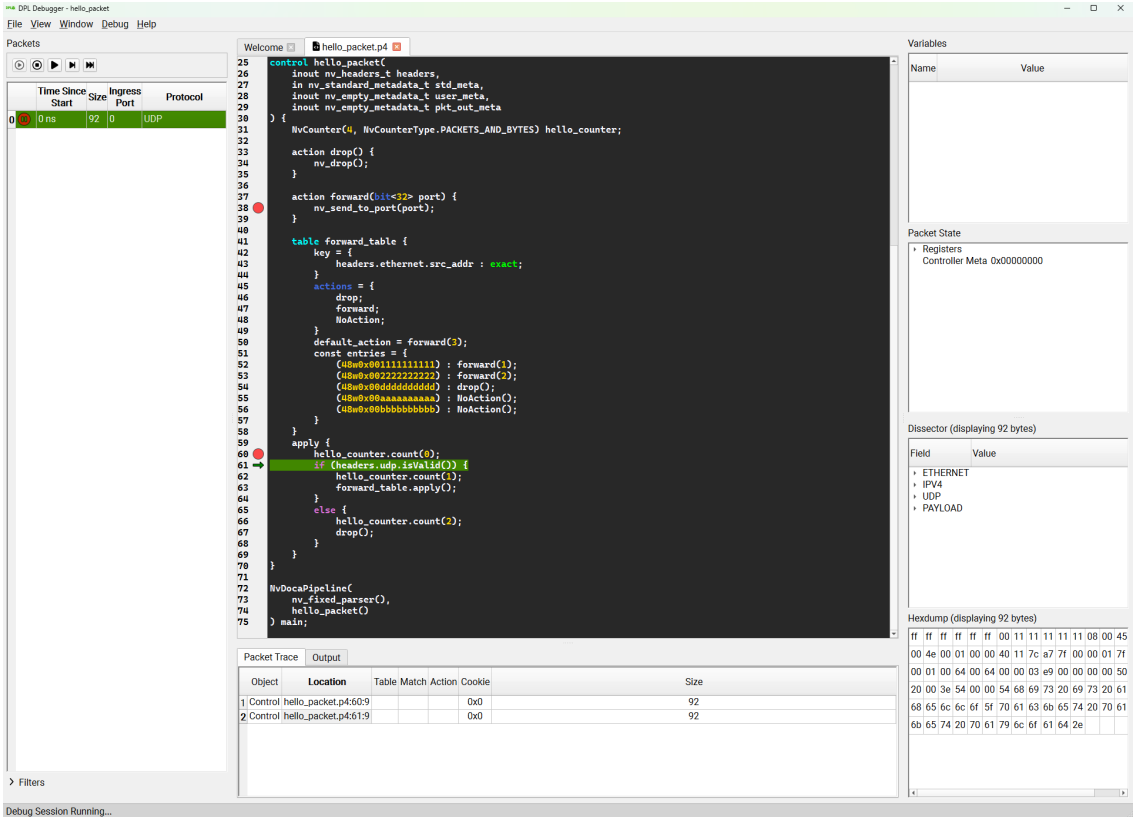
7.8.6.4.4.2 4.4.2. Sending Traffic (`send_hello_packet.py`)

Assuming port `p1` is connected via loopback to `p0`, this Python script sends a TCP packet to the pipeline.

```
from scapy.all import IP, TCP, Ether, Packet, Raw, sendp

packet = (Ether(src='00:11:11:11:11:11', dst='ff:ff:ff:ff:ff:ff') /
          IP() /
          TCP(sport=100, dport=100, seq=1001, flags='P') /
          Raw(load=b'This is a hello_packet packet payload.'))

sendp(packet, "p1", count=1)
```

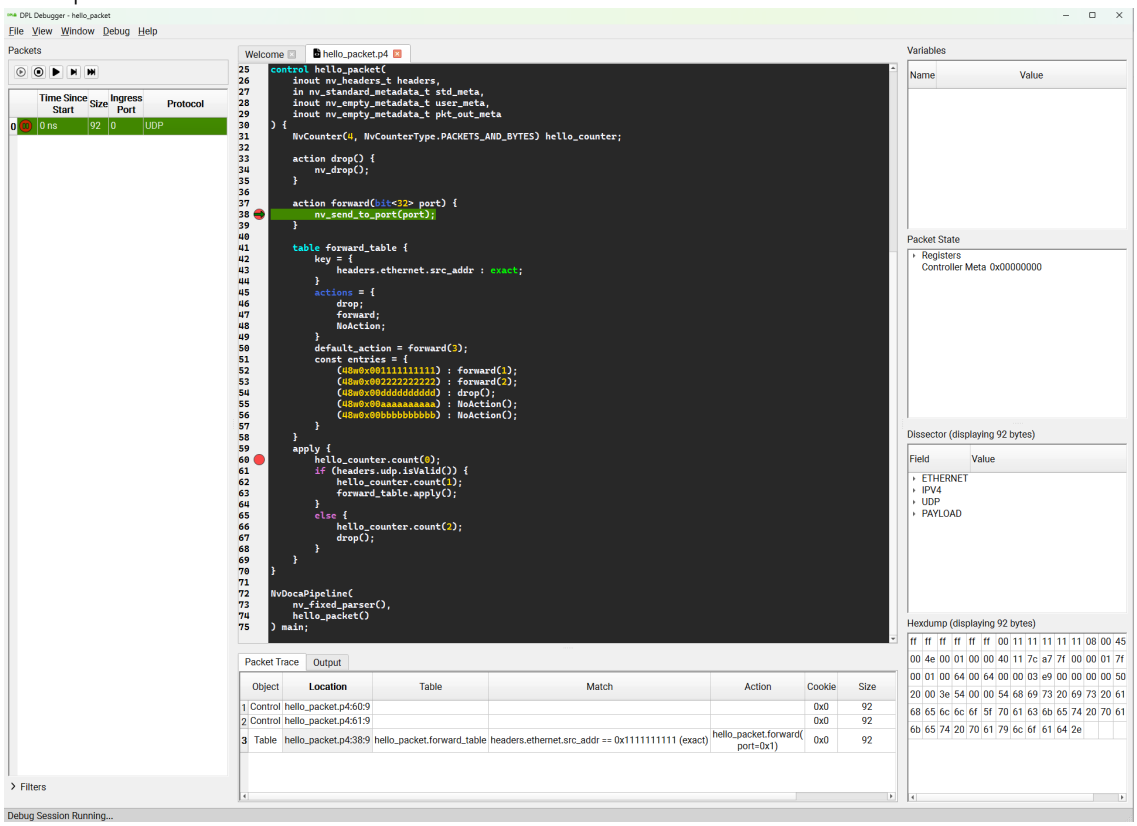
7.8.6.4.4.3 4.4.3. Execution & Trace

- The packet arrives at the debugger and pauses at the enabled breakpoint.
- Clicking **Step** creates a new row in the trace and moves the packet to the next line.



- If you edit the packet in the Dissector (e.g., changing TCP to UDP) and click **Continue**, the packet diverges based on your edit (satisfying the `if` statement) and pauses at the next

breakpoint.

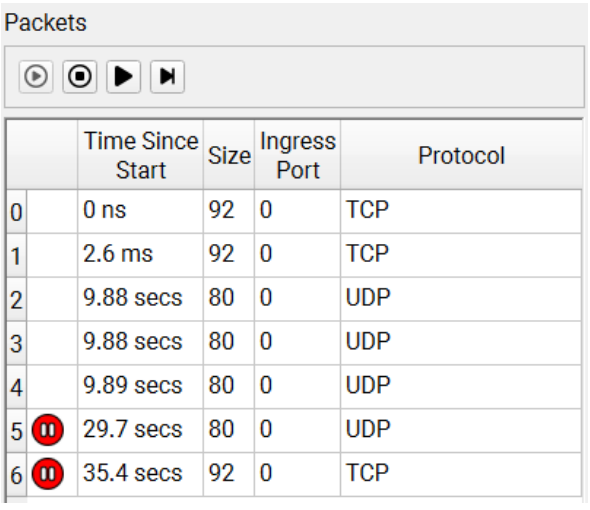


7.8.6.5 5. UI Reference: Widgets

The DPL Debugger provides multiple widgets to analyze packet states and system health:

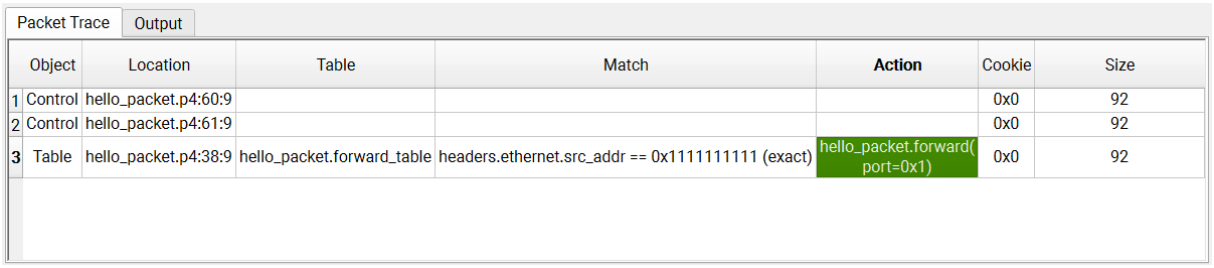
7.8.6.5.1 5.1. Packet List

Displays packets sent to the debugger. Selecting a packet reveals its trace. (Clear this list via **File → Remove all packets**).



7.8.6.5.2 5.2. Packet Trace

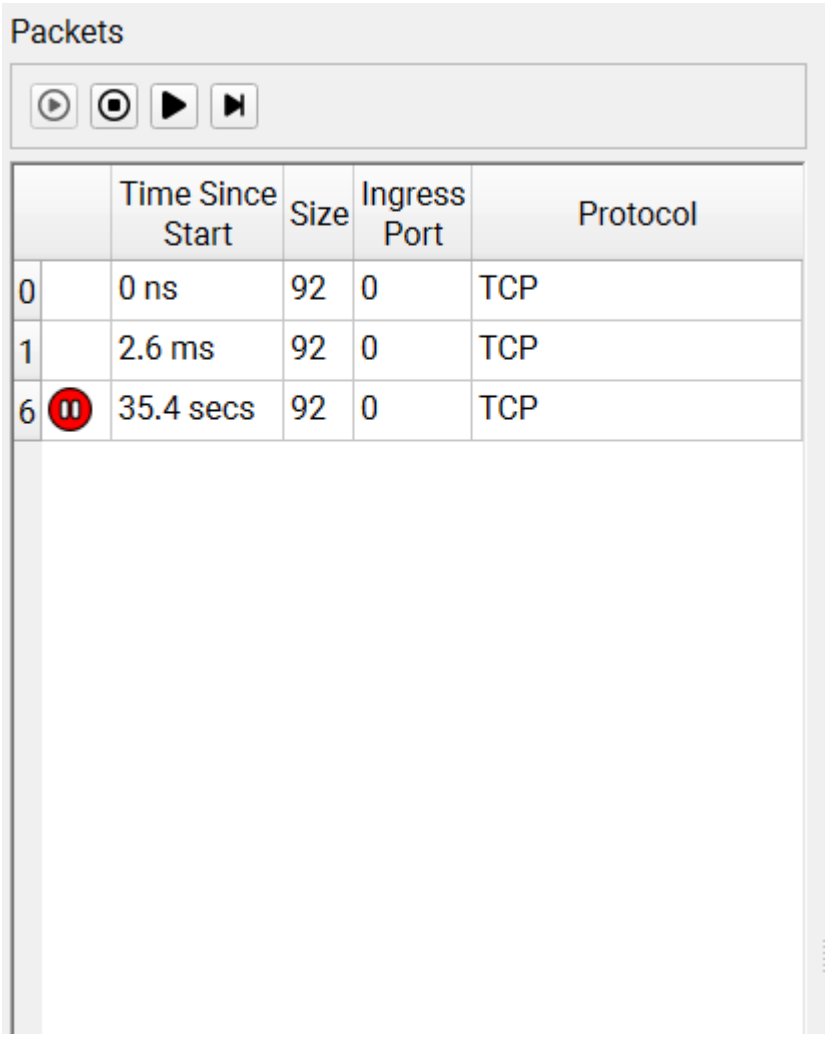
Each row represents a single debug point hit. Selecting a row reveals the packet data, state, and pipeline at that exact moment.

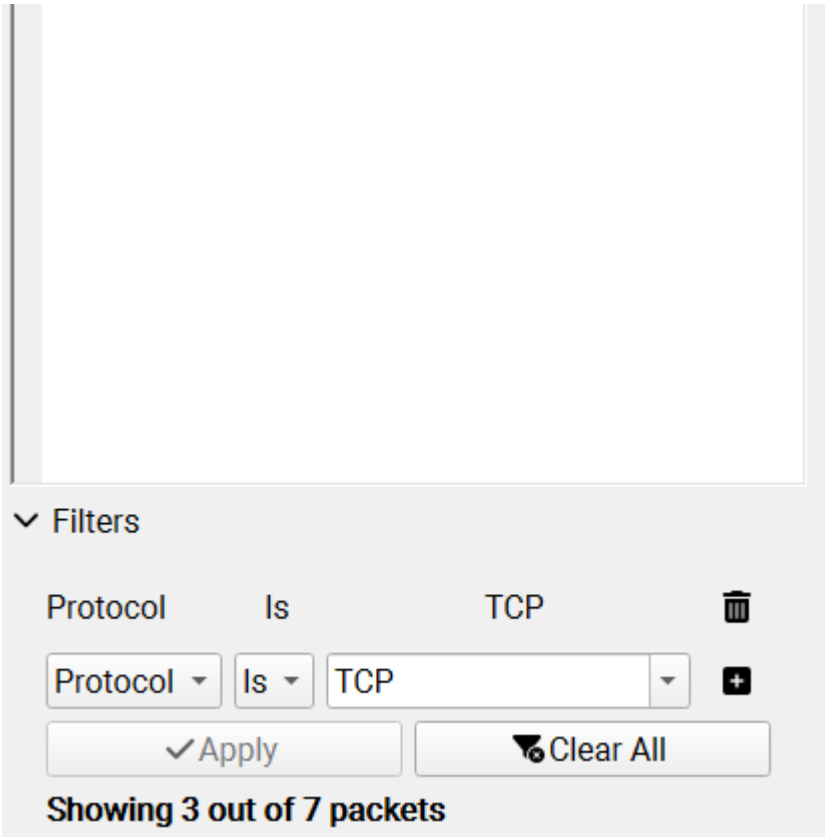


	Object	Location	Table	Match	Action	Cookie	Size
1	Control	hello_packet.p4:60:9				0x0	92
2	Control	hello_packet.p4:61:9				0x0	92
3	Table	hello_packet.p4:38:9	hello_packet.forward_table	headers.ethernet.src_addr == 0x1111111111 (exact)	hello_packet.forward(port=0x1)	0x0	92

7.8.6.5.3 5.3. Filter

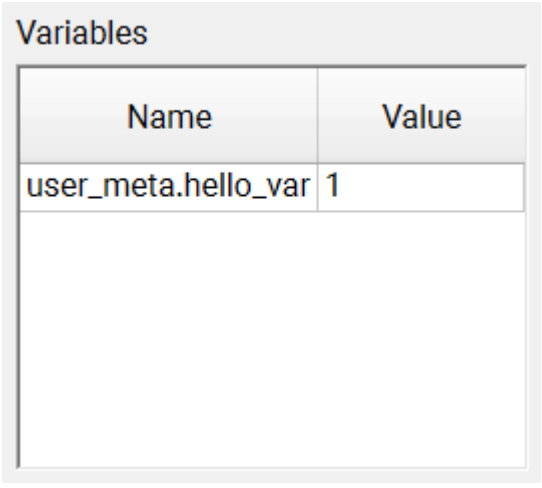
Allows packet filtering by Table Name, Action, Match Field, Match Type, Match Value, Ingress Port, Packet Size, or Protocol.





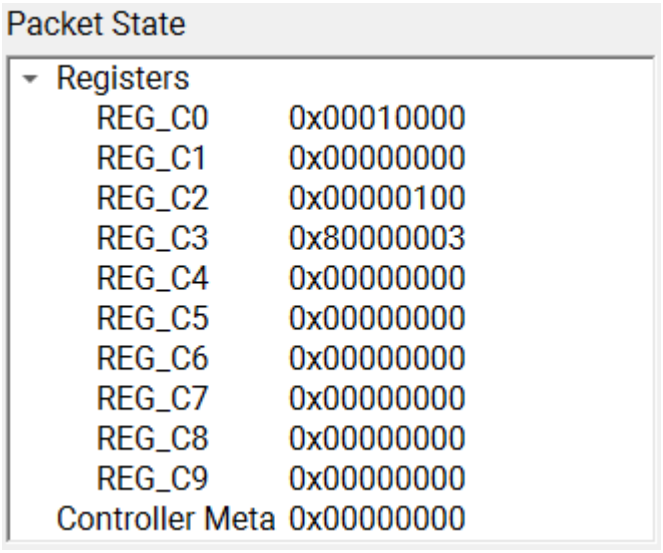
7.8.6.5.4 5.4. Variables

Displays the program's defined variables and their values at the selected debug point.



7.8.6.5.5 5.5. Packet State

Displays low-level variables, such as registers and controller metadata.



7.8.6.5.6 5.6. Packet Dissector

Displays the parsed packet. Paused packets can be edited by double-clicking values or pressing F2. (Values are shown/edited in decimal format and translated upon saving).

Dissector (displaying 80 bytes)

Field	Value
▾ ETHERNET	
dst_addr	ff:ff:ff:ff:ff:ff
src_addr	00:11:11:11:11:11
ether_type	Internet Protocol version 4 (IPv4)
▾ IPV4	
version	4
ihl	5
diffserv	CS0
ecn	Not-ECT
total_len	66
identification	1
flags	0
frag_offset	0
ttl	64
protocol	UDP
hdr_checksum	31912
src_addr	127.0.0.1
dst_addr	127.0.0.1
▾ UDP	
src_port	100
dst_port	100
length	46
checksum	45592
▾ PAYLOAD	
data	This is a hello_packet packet payload.

7.8.6.5.7 5.7. Hex Dump

Displays a raw hexadecimal dump of the packet. Double-click or press F2 to edit.

Hexdump (displaying 80 bytes)

ff	ff	ff	ff	ff	ff	00	11	11	11	11	11	08	00	45	00
00	42	00	01	00	00	40	11	7c	a8	7f	00	00	01	7f	00
00	01	00	64	00	64	00	2e	b2	18	54	68	69	73	20	69
73	20	61	20	68	65	6c	6c	6f	5f	70	61	63	6b	65	74
20	70	61	63	6b	65	74	20	70	61	79	6c	6f	61	64	2e

7.8.6.5.8 5.8. Source Code

Displays DPL source files. Clicking objects with source correlations (tables, actions, debug points) will automatically open the relevant file.

Welcome | hello_packet.p4

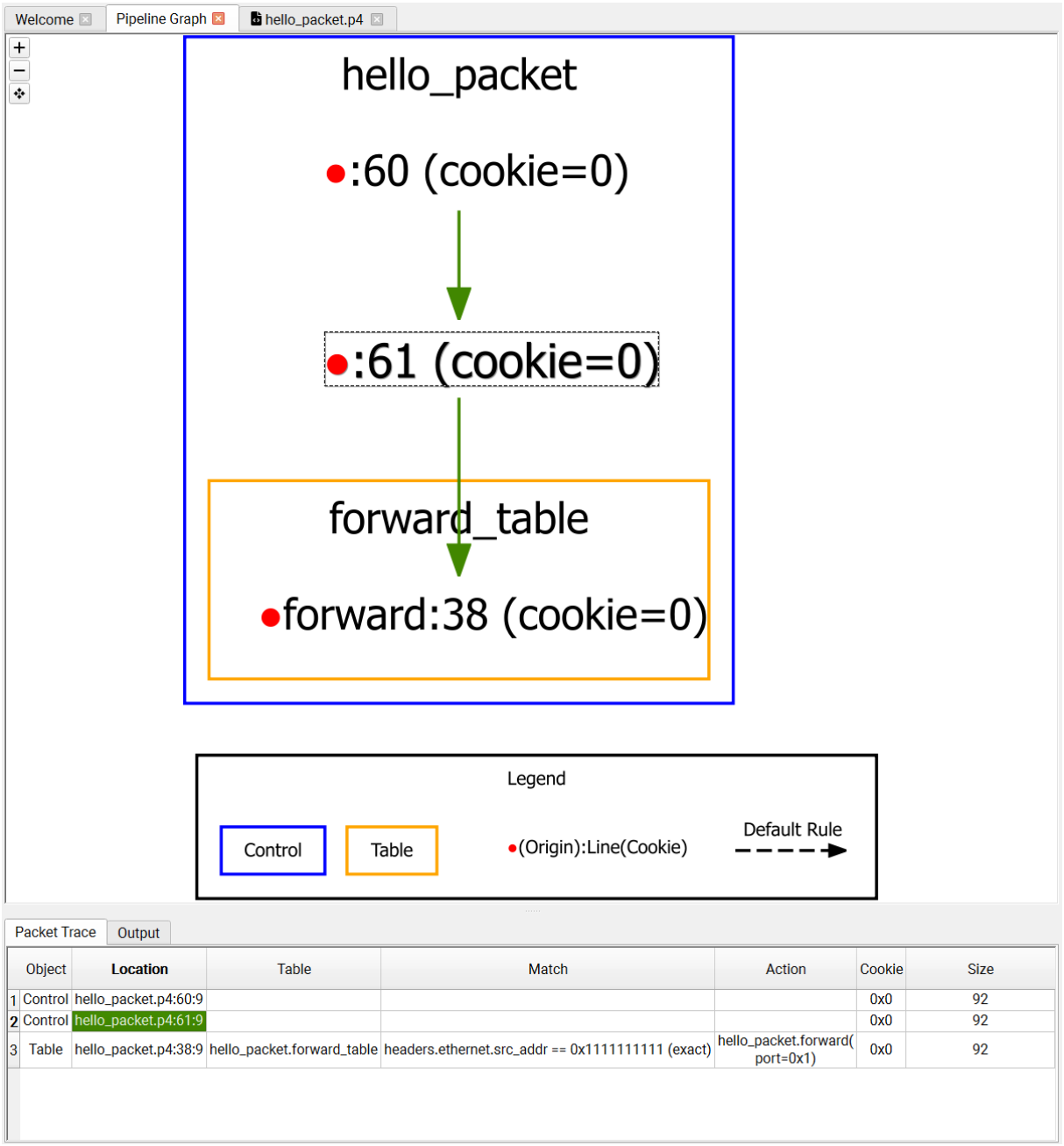
```
12
13  #include <doca_model.p4>
14  #include <doca_headers.p4>
15  #include <doca_externs.p4>
16  #include <doca_parser.p4>
17
18  /*
19   * This basic application demonstrates a simple match/action pipeline using NVIDIA
20   * DOCA P4 Target Architecture.
21   * - Match on destination MAC address
22   * - Forward or drop the packet
23   */
24
25  control hello_packet(
26      inout nv_headers_t headers,
27      in nv_standard_metadata_t std_meta,
28      inout nv_empty_metadata_t user_meta,
29      inout nv_empty_metadata_t pkt_out_meta
30  ) {
31      NvCounter(4, NvCounterType.PACKETS_AND_BYTES) hello_counter;
32
33      action drop() {
34          nv_drop();
35      }
36
37      action forward(bit<32> port) {
38          nv_send_to_port(port);
39      }
40
41      table forward_table {
42          key = {
43              headers.ethernet.src_addr : exact;
44          }
45          actions = {
46              drop;
47              forward;
48              NoAction;
49          }
50          default_action = forward(3);
51          const entries = {
52              (48w0x001111111111) : forward(1);
53              (48w0x002222222222) : forward(2);
54              (48w0x00dddddddddd) : drop();
55              (48w0x00aaaaaaaaaa) : NoAction();
56              (48w0x00bbbbbbbbbb) : NoAction();
57          }
58      }
59      apply {
60          hello_counter.count(0);
61          if (headers.udp.isValid()) {
62              hello_counter.count(1);
63              forward_table.apply();
```

Packet Trace | Output

	Object	Location	Table	Match	Action	Cookie	Size
1	Control	hello_packet.p4:60:9				0x0	92
2	Control	hello_packet.p4:61:9				0x0	92
3	Table	hello_packet.p4:38:9	hello_packet.forward_table	headers.ethernet.src_addr == 0x1111111111 (exact)	hello_packet.forward(port=0x1)	0x0	92

7.8.6.5.9 5.9. Pipeline

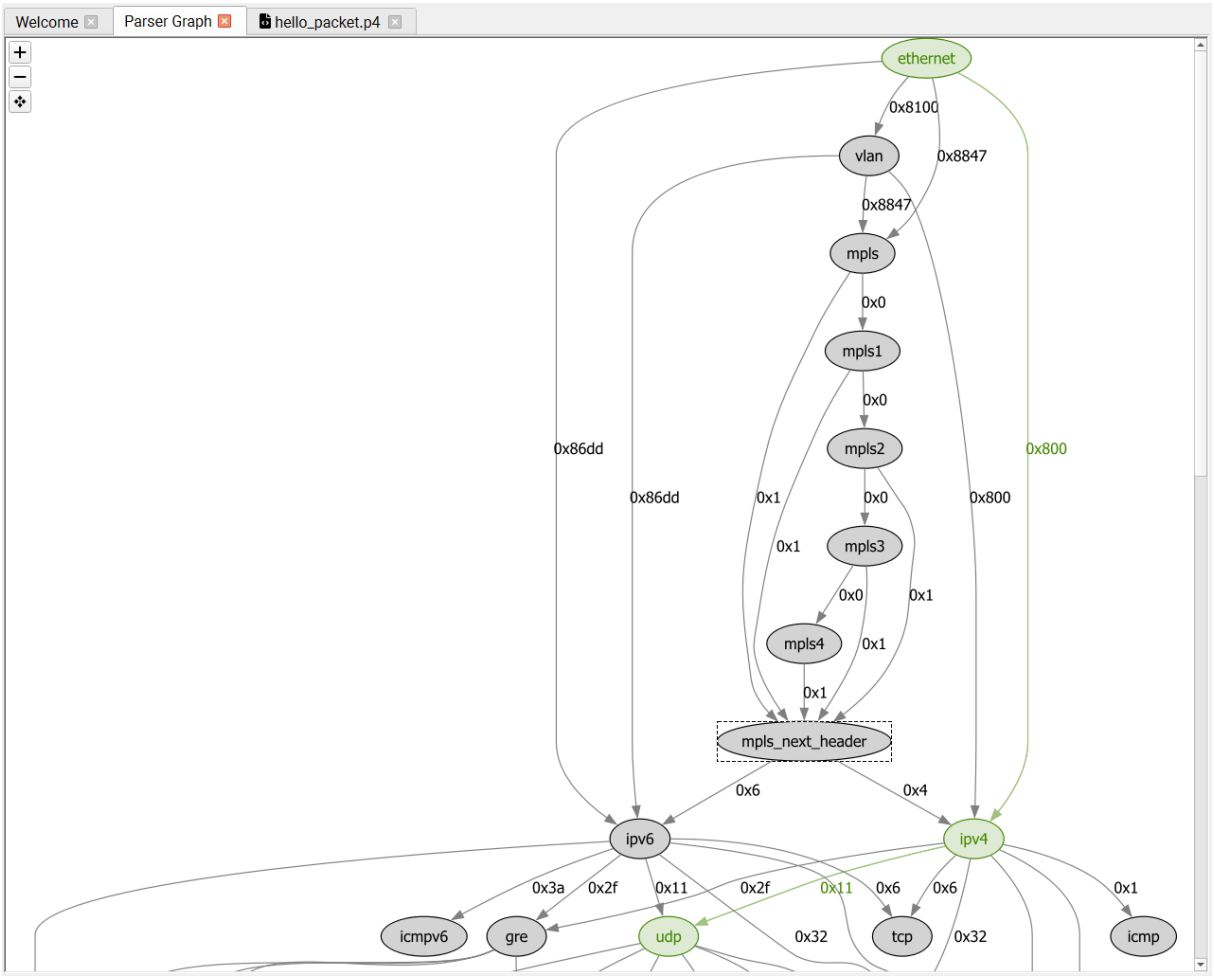
Displays the selected packet's pipeline (View → Pipeline Graph). Clicking a trace row highlights the corresponding debug point in the graph.



	Object	Location	Table	Match	Action	Cookie	Size
1	Control	hello_packet.p4:60:9				0x0	92
2	Control	hello_packet.p4:61:9				0x0	92
3	Table	hello_packet.p4:38:9	hello_packet.forward_table	headers.ethernet.src_addr == 0x1111111111 (exact)	hello_packet.forward(port=0x1)	0x0	92

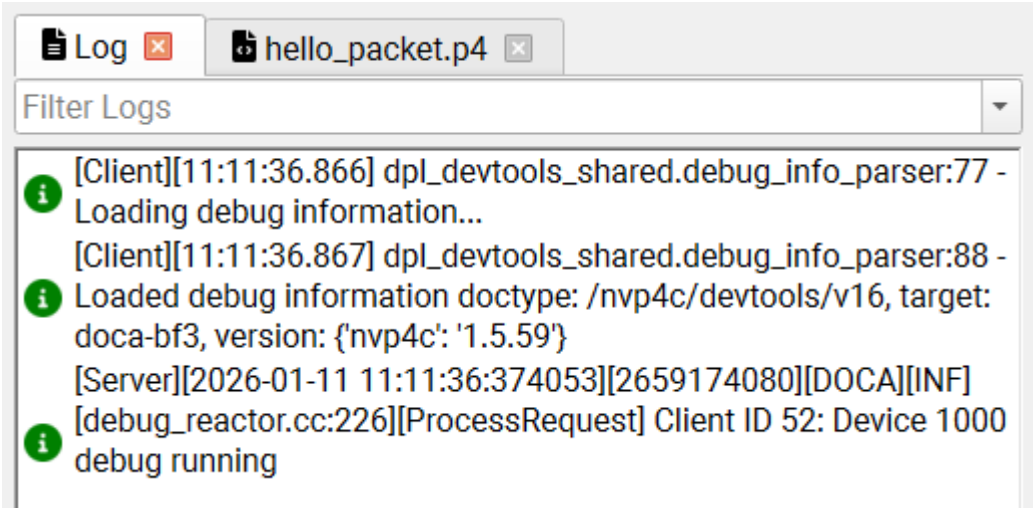
7.8.6.5.10 5.10. Parser

Displays the complete parser graph (both user-defined flex nodes and hardware-fixed nodes) and outlines how the selected packet was parsed (View → Parser Graph).

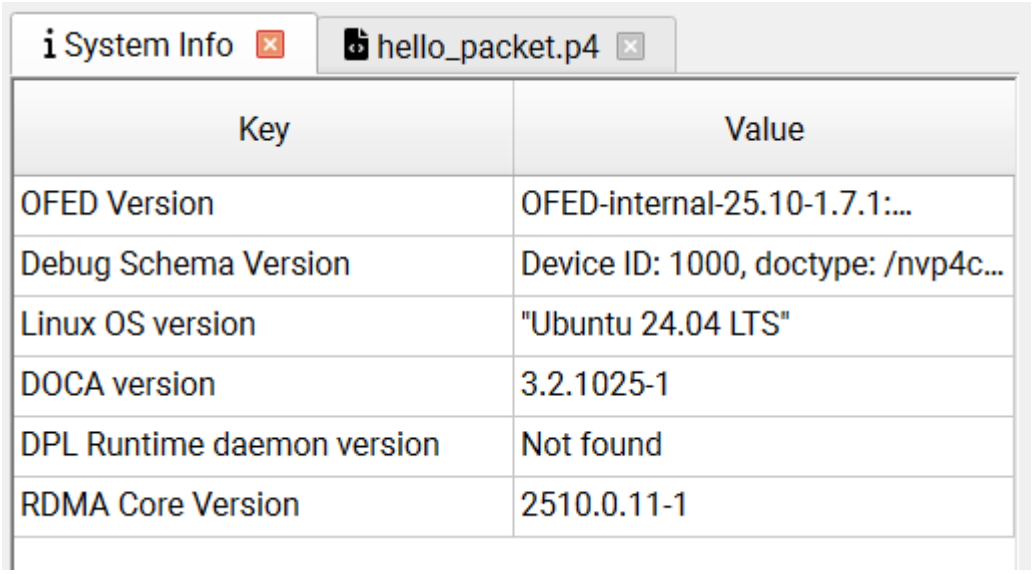


7.8.6.5.11 5.11. Log

Displays internal DPL Runtime daemon logs, filterable by severity (View → Log).



System Info: (View → System Info) Displays target system information.



Key	Value
OFED Version	OFED-internal-25.10-1.7.1:...
Debug Schema Version	Device ID: 1000, doctype: /nvp4c...
Linux OS version	"Ubuntu 24.04 LTS"
DOCA version	3.2.1025-1
DPL Runtime daemon version	Not found
RDMA Core Version	2510.0.11-1

7.8.6.6 6. Release Notes

7.8.6.6.1 6.1. New Features

Feature	Description
Correlated hexdump and dissector	Added highlighting, selecting a state or field in the dissector now highlights the corresponding bytes in the hexdump, and vice versa.
Source code keyboard controls	Enabled keyboard interaction in the code viewer. A caret is now visible and starts at the first line, it can be moved with the mouse or keyboard. Press `F9` to toggle a breakpoint on the caret line, matching the existing click-on-margin behavior. The breakpoint margin now has a tooltip.
Graph view buttons	Added zoom-in/zoom-out/fit-to-view buttons to the graph views.
Auto reinject checkbox	Added a checkbox to automatically reinject filtered-out packets as they arrive, so the debug session is not interrupted by filtered packets.
Paused packets checkbox	Added a checkbox to filter out unpaused packets from the packets table, so you can focus on paused packets only. When unchecked, both paused and unpaused packets are shown.
Externs column	Added an "Externs" column to the packet trace table, showing the externs associated with the table entry that the packet matched and their details.

7.8.6.6.2 6.2. Bug Fixes

- Code viewer syntax highlighting losing its bold weight after the window was resized.
- When a graph is not available, a placeholder message is now shown instead of a blank view.
- Packet filters using the "Is", "Is Not", "Starts With", or "Ends With" operators raise an exception when applied to fields that resolve to multiple values (such as match keys with several fields).
- Packets table's filter dropdowns not refreshing their available values after the packets table was cleared.
- Packet variables and packet details panes not keeping even proportions when their pane was resized.
- Packet state transitions during a debug session are more consistent and predictable.

7.8.7 DPL Admin

7.8.7.1 Introduction

DPL Admin is a CLI tool used for managing and inspecting the DPL Runtime daemon. It provides the following functionalities:

- Viewing the current configuration of the DPL Runtime daemon
- Reading the DPL Runtime Service logs
- Setting the log level for the DPL Runtime Service

- Retrieving the hardware steering dump, capabilities, and NVConfig of the DPL Runtime Service

> The DPL Admin tool is accessed and managed through the `dpl_dev` container. For detailed execution steps, refer to the Usage Guide.

7.8.7.2 Options and Commands

Usage:

```
dpl_admin.sh help
```

```
usage: dpl admin [--help] [-a <host:port>] ... <dpl_admin command> [args...]
```

Options:

Flag / Option	Description
`-h` , `--help`	Show help message and exit
`-V` , `--version`	Display the program's version number and exit
`-v` , `--verbose`	Increase output verbosity (default: `False`)
`-a HOST:PORT` , `--address HOST:PORT`	Set the `dpl_admin` server socket address in the format `[ipv4/6]:[address]:[port]` IPv6 addresses with ports must be enclosed in square brackets. Examples: `ipv6:[2607:f8b0:400e:c00::ef]:443` `ipv6:[::]:1234` `10.1.1.1:9600` If not specified, defaults to `localhost:9600`
`-t [SEC]` , `--timeout [SEC]`	Set gRPC channel connectivity timeout in seconds (default: `60`)

Authentication options (mutually inclusive for secure connection):

Flag / Option	Description
`--cacert PATH_TO_CA_CERT`	Path to CA certificate for secure connections
`--cert PATH_TO_CERT`	Path to client certificate for mutual authentication
`--private_key PATH_TO_KEY`	Path to client private key for mutual authentication

Refer here for TLS setup example

Available commands:

> Use `dpl admin <command> -h` for more details on each.

Command	Description
`config`	Retrieve device configurations
`set-counter-cache-timeout`	Set counter cache timeout for the device
`set-log-level`	Configure the logging level
`get-log-full`	Fetch the full device log
`get-log-recent`	Retrieve recent log entries
`hws-dump`	Dump hardware steering data
`hca-capabilities`	Show HCA capabilities
`nv-config`	Access non-volatile configuration settings
`set-idle-timeout-polling-interval`	Set the idle timeout polling interval

The `-a` flag is only required if the address and port are not set in the environment variable `DPL_ADMIN_DEFAULT_ARGS`, as documented under section "Configuring Launch Scripts". The examples below assume `DPL_ADMIN_DEFAULT_ARGS` has already been configured.

7.8.7.2.1 config

Displays the DPL Runtime Service configuration information, including servers, logging, devices and HAL configurations.

- Usage:

```
dpl_admin -a <host:port> config
```

- Options:
  - `-h`, `-help` – Displays command's help message and exit
- Example:

```
dpl_admin config
```

- Output:

```
  CONFIGURATION

    servers
                                                                server_address
tcp_port

                                                          p4_rt          [::]
9559
                                                       dpl_nspect        [::]
9560
                                                         admin           [::]
9600
```

```
       log
                                                                                attr                      value


                                                                               name:    /var/log/doca/dpl_rt_service/
dpl_rtd.log                                                                    level:                       INFO


     devices

                    id    counter cache timeout [msec]   idle timeout polling interval [sec]
controller attr                        interfaces

                  1000                0                                  2

                                                                                                                attr
  value       name   port_id   mtu           mac                                                                 port_id:
   9876       eth2       0      1514   00:00:00:00:00:00
                         eth4       1      1514   00:00:00:00:00:00
    eth5       2      1514   00:00:00:00:00:00
    eth6       3      1514   00:00:00:00:00:00
    eth7       4      1514   00:00:00:00:00:00


       hal
                                                                                    attr          value

                                                                              number of queues:      1
                                                                                 queue size:        1024
                                                                                 burst size:         32
```

7.8.7.2.2 set-counter-cache-timeout

Sets counter cache timeout for device.

- Usage:

```
dpl admin -a <host:port> set-counter-cache-timeout --device_id/-d <DEVICE_ID> --timeout_value/-t
<TIMEOUT_VALUE>
```

- Options:
  - **-h**, **- help** – Displays command's help message and exit.
  - **--device_id**, **-d** – Required; specifies device ID.
  - **--timeout_value**, **-t** – Required; specifies new counter cache time-out value.
- Example:

```
dpl admin set-counter-cache-timeout --device_id 1000 --timeout_value 3
```

- Output:

```
Device ID: 1000, new timeout: 3 msec
```

7.8.7.2.3 set-log-level

Sets DPL Runtime Service's log level.

- Usage:

```
dpl admin -a <host:port> set-log-level --level <LOG_LEVEL>
```

- Options:
  - **-h**, **- help** – Displays command's help message and exit.
  - **--level**, **-l** – Required; new log level out of **{DISABLE, CRITICAL, ERROR, WARNING, INFO, DEBUG, TRACE}**
- Example:

```
dpl admin set-log-level --level DEBUG
```

- Output:

```
dpl_rtd log level set to: 'DEBUG'
```

> Setting log level is not persistent. DPL Admin restarts with log level according to config file.

7.8.7.2.4 get-log-full

Reads DPL Runtime Service's log into a file.

- Usage:

```
dpl admin -a <host:port> get-log-full
```

- Options:
  - **-h**, **- help** – Displays command's help message and exit.
  - **--outpath**, **-o** – Optional; specifies output log file path. Defaults to current directory.
- Example:

```
dpl_admin get-log-full --outpath '/tmp/dpl_agent.log'
```

- Output:

```
Find Log file at: '/tmp/dpl_agent.log'
```

7.8.7.2.5 get-log-recent

Displays recent DPL Runtime Service's log lines.

- Usage:

```
dpl admin -a <host:port> get-log-recent
```

- Options:
  - **-h** , **- help** – Displays command's help message and exit.
  - **--number_of_lines** , **-n** – Optional; specifies number of recent log lines to read. Defaults to 10.
- Example:

```
dpl admin get-log-recent --number_of_lines 6
```

- Output:

```
[12:45:09:480451][73079][DOCA][INF][OnReadDone] New client connected to device 1000
[12:45:09:481295][73078][DOCA][INF] A program was previously loaded, clearing it before applying new one...
[12:45:09:481314][73078][DOCA][INF] [DPL Device 1000] Removing DPL program ...
[12:45:09:498476][73078][DOCA][INF] [DPL Device 1000] DPL program was removed.
[12:45:09:499432][73078][DOCA][INF] [DPL Device 1000] Loading DPL program ...
[12:45:09:522419][73078][DOCA][INF] [DPL Device 1000] DPL program was loaded successfully.
```

7.8.7.2.6 hws-dump

Dumps DPL Runtime Service's hardware steering rules into a file.

- Usage:

```
dpl admin -a <host:port> hws-dump --device_id/-d <DEVICE_ID>
```

- Options:
  - **-h** , **- help** – Displays command's help message and exit.
  - **--device_id** , **-d** – Required; specifies device ID.
  - **--parser_args** , **-a** – Optional; specifies arguments to pass to HWS dump tool.
  - **--outpath** , **-o** – Optional; specifies output file path. Defaults to current directory.
- Example:

```
dpl admin hws-dump --device_id 1000 --parser_args '-vvv' --outpath '/tmp/dpl_hws_dump.txt'
```

- Output:

```
Find HWS dump file at: '/tmp/dpl_hws_dump.txt'
```

Intended as a debugging aid for developers.

Passing multiple options for the **--parser_args** flag requires wrapping them with back-slash escaped quotation marks or apostrophe. For example:

```
dpl admin hws-dump --device_id 1000 --parser_args \'-vvv --extra_hw_res all\' --outpath '/tmp/
dpl_hws_dump.txt'
```

7.8.7.2.7 hca-capabilities

Dumps NIC HCA capabilities into a file.

- Usage:

```
dpl admin -a <host:port> hca-capabilities --device_id/-d <DEVICE_ID>
```

- Options:
  - **-h** , **- help** – Displays command's help message and exit.
  - **--device_id** , **-d** – Required; specifies device ID.
  - **--outpath, -o** – Optional; specifies output file path. Defaults to current directory.
- Example:

```
dpl admin hca-capabilities --device_id 1000 --outpath '/tmp/dpl_hca_capabilities.json'
```

- Output:

```
Find HCA capabilities file at: '/tmp/dpl_hca_capabilities.json'
```

7.8.7.2.8 nv-config

Dumps DPL Runtime Service's NV config into a file.

- Usage:

```
dpl admin -a <host:port> nv-config --device_id/-d <DEVICE_ID>
```

- Options:
  - **-h** , **- help** – Displays command's help message and exit.
  - **--device_id** , **-d** – Required; specifies device ID.
  - **--outpath, -o** – Optional; specifies output file path. Defaults to current directory.
- Example:

```
dpl admin nv-config --device_id 1000 --outpath '/tmp/dpl_nv_config.txt'
```

- Output:

```
Find NV config file at: '/tmp/dpl_nv_config.txt'
```

7.8.7.2.9 set-idle-timeout-polling-interval

Sets idle timeout polling interval [in seconds]

- Usage:

```
dpl admin -a <host:port> set-idle-timeout-polling-interval --device_id/-d <DEVICE_ID> --interval/-i
<NEW_INTERVAL>
```

- Options:

- **-h**, **- help** – Displays command's help message and exit.
- **--device_id**, **-d** – Required; specifies device ID.
- **--interval**, **-i** – Required; New polling interval to be set [seconds].

- Example:

```
dpl admin set-idle-timeout-polling-interval --device_id 1000 --interval 3
```

- Output:

```
Device ID: 1000, new polling interval: 3 seconds
```

7.8.8 P4 Runtime Controller

DPL applications are deployed to the NVIDIA® BlueField® networking platform (DPU or SuperNIC) via the P4Runtime API. Since DPL is derived from the P4-16 language, it is compatible with the P4Runtime specification, enabling standard runtime interaction with the compiled DPL pipeline

7.8.8.1 Introduction

The P4 Runtime shell (**p4runtime_sh**) is a Python-based CLI tool used to interact with this API. It is primarily used for loading DPL programs, testing match-action tables, and debugging pipeline behavior.

- Loading simple DPL programs
- Testing match-action tables
- Debugging pipeline behavior

The shell can be invoked using the launch script provided in the DPL Development container.

> For detailed instructions, refer to "Loading DPL Applications" documentation.

7.8.8.2 Basic Information and Inspection

These commands allow you to view the high-level structure of your loaded P4 program.

- Retrieves the content of **p4info.txt** for the currently loaded DPL program:

```
p4info
```

- Lists all available tables:

```
tables
```

- Displays information about a specific table:

```
tables["<P4_TABLE_NAME>"]
```

- Lists all defined actions:

```
actions
```

- Shows parameters and sizes for a specific action:

```
actions["<P4_ACTION_NAME>"]
```

7.8.8.3 Working with P4 Table Entries

7.8.8.3.1 Specifying Match Keys

When creating or modifying entries, you must define the match keys. The syntax depends on the match type defined in your DPL program.

Match Method	Syntax for Specifying Match Key Value	Note
Exact	`te.match["<KEY_NAME>"] = "<VALUE>"`	Exact value match.
Ternary	`te.match["<KEY_NAME>"] = "<VALUE>&&&<MASK>"` `te.priority = <PRIORITY_VALUE>`	The mask is provided in the match line, separated by `&&&`. If the mask is not specified, a full match mask will be used.
LPM	`te.match["<KEY_NAME>"] = "<VALUE>/<PREFIX_LEN>"`	The prefix length is provided in the match line, separated by `/`. If the prefix length is not specified, the full field bitwidth is used.
Range	`te.match["<KEY_NAME>"] = "<MIN_VALUE>..<MAX_VALUE>"`	The values are are separated by `..`, both values must be present.

7.8.8.3.2 Adding Entries

To add a standard entry, you must define the table, the action, the match keys, and the action parameters.

- Regular entry:

```
te = table_entry["<P4_TABLE_NAME>"](action="<P4_ACTION_NAME>")
te.match["<MATCH_KEY_NAME>"] = "<MATCH_VALUE>"
te.action["<PARAMETER_NAME>"] = "<PARAMETER_VALUE>"
te.insert()
```

- Entry with idle timeout:

```
te = table_entry["<P4_TABLE_NAME>"](action="<P4_ACTION_NAME>")
te.match["<MATCH_KEY_NAME>"] = "<MATCH_VALUE>"
te.action["<PARAMETER_NAME>"] = "<PARAMETER_VALUE>"
te.idle_timeout_ns = 3000000000 # Example: 3 seconds
te.insert()
```

7.8.8.3.3 Reading Entries

- Read a specific entry:

```
te = table_entry["<P4_TABLE_NAME>"]
te.match["<MATCH_KEY_NAME>"] = "<MATCH_VALUE>"
te.read(lambda te: print(te))
```

- Read all entries in a table:

```
te = table_entry["<P4_TABLE_NAME>"]
te.is_default = False
te.read(lambda te: print(te))
```

- Read time since last hit:

> ⚠ Only for tables with **idle-timeout** enabled.

```
te = table_entry["<P4_TABLE_NAME>"]
te.time_since_last_hit.elapsed_ns = 0 # Enable reading timestamp
te.match["<MATCH_KEY_NAME>"] = "<MATCH_VALUE>"
te.read(lambda te: print(te))
```

7.8.8.3.4 Deleting Entries

- Delete a specific entry:

```
te = table_entry["<P4_TABLE_NAME>"]
te.match["<MATCH_KEY_NAME>"] = "<MATCH_VALUE>"
te.delete()
```

- Delete all entries in a table:

```
te = table_entry["<P4_TABLE_NAME>"]
te.read(lambda te: te.delete())
```

7.8.8.3.5 Modifying Entries

You can modify specific action parameters of an existing table entry without removing it, provided those parameters are explicitly marked as modifiable in the P4 program.

> ⚠ Modification is supported only for action parameters annotated with **@nv_modifiable**. Refer to DOCA Target Architecture for more details.

> ⚠ Match keys, Action IDs, and non-annotated parameters cannot be changed in place. To modify these fields, you must delete the existing entry and insert a new one.

The following example demonstrates how to update a modifiable parameter:

```
# Select the table and action
te = table_entry["<P4_TABLE_NAME>"](action="<P4_ACTION_NAME>")

# Define the match keys to identify the specific entry
te.match["<MATCH_KEY_NAME>"] = "<MATCH_VALUE>"

# Update the specific parameter (Must be annotated with @nv_modifiable)
te.action["<PARAMETER_NAME>"] = "<NEW_VALUE>"

# Commit the modification
te.modify()
```

7.8.8.3.6 Managing Default Entries

⚠ Default entries are implicit. They cannot be inserted or deleted, only modified or read.

- Reading a default entry:

```
te = table_entry["<P4_TABLE_NAME>"]
te.is_default = True
te.read(lambda te: print(te))
```

- Reading with counter:

```
te = table_entry["<P4_TABLE_NAME>"]
te.is_default = True
te.counter_data.byte_count = 0  # Request counter data
te.read(lambda te: print(te))
```

- Modifying a default entry: Modification follows the same rules as standard entries. You can only update action parameters annotated with `@nv_modifiable`.

```
te = table_entry["<P4_TABLE_NAME>"](action="<P4_ACTION_NAME>")
te.is_default = True
te.action["<@nv_modifiable PARAMETER_NAME>"] = "<PARAMETER_VALUE>"
te.modify()
```

7.8.8.4 Working with Counters

7.8.8.4.1 Direct Counters

Direct counters are attached to specific table entries.

- List counters: `direct_counters`
- Read specific entry counter:

```
ce = DirectCounterEntry("<P4_DIRECT_COUNTER_NAME>")
ce.table_entry.match["<KEY_NAME>"] = "<VALUE>"
ce.read(lambda ce: print(ce))
```

- Read default entry counter: Set `ce.table_entry.is_default = True` before reading.
- Clear specific counter: Set `ce.packet_count = 0` and call `ce.modify()`.
- Clear all counters:

```
for dc in direct_counters:
    ce = DirectCounterEntry(dc.name)
    ce.packet_count = 0
    ce.modify()
```

7.8.8.4.2 Indirect Counters

Indirect counters are accessed by index.

- List counters: `counters`
- Read specific index:

```
ce = counter_entry["<P4_INDIRECT_COUNTER_NAME>"]
ce.index = <INDEX>
ce.read(lambda ce: print(ce))
```

- Clear specific index: Set `ce.byte_count = 0` and `ce.packet_count = 0`, then call `ce.modify()`.

7.8.8.5 Working with Meters

7.8.8.5.1 Direct Meters

- List meters: `direct_meters`
- Read specific meter:

```
me = DirectMeterEntry("<P4_DIRECT_METER_NAME>")
me.table_entry.match["<KEY>"] = "<VALUE>"
me.read(lambda me: print(me))
```

- Configure meter: Set config values (`cir`, `cburst`, `pir`, `pburst`) and call `me.modify()`.

7.8.8.5.2 Indirect Meters

- List meters: `meters`
- Read specific index:

```
me = meter_entry["<P4_INDIRECT_METER_NAME>"]
me.index = <INDEX>
me.read(lambda me: print(me))
```

- Clear meter: Set `me.cir = 0` (and other rates) and call `me.modify()`.

7.8.8.6 Packet I/O and Notifications

7.8.8.6.1 Packet In (Sniffing)

To capture packets sent from the DPL Runtime to the Controller:

```
packet_in.sniff(lambda m: print(m), timeout=10)
```

For detailed parsing, you can import `impacket` and decode the payload

7.8.8.6.2 Packet Out (Sending)

To send packets from the Controller to the DPL Runtime. You typically use `scapy` to build the packet first.

```
# 1. Build packet string (using Scapy or raw bytes)
my_pkt = b'<PACKET_BYTE_STRING>'

# 2. Send packet with metadata
# Repeat metadata args for every metadata defined in P4
packet_out(payload=my_pkt, <METADATA_NAME>="<VALUE>").send()
```

7.8.8.6.3 Idle Timeout Notifications

To receive notifications for entry timeouts (requires `nv_support_timeout` in DPL source):

```
idle_timeout_notification.sniff(lambda m: print(m), timeout=10)
```

7.8.9 DPL Runtime Controller SDK

DPL applications are deployed to the NVIDIA® BlueField® networking platform (DPU or SuperNIC) using the P4Runtime API. However, tables annotated with `nv_high_update_rate=true` are managed exclusively through the `dpl_rt_controller` SDK.

7.8.9.1 Introduction

7.8.9.1.1 Control Protocols

The DPL runtime service implements two control plane protocols for managing data plane elements:

- P4Runtime – Based on the open, gRPC-based standard (P4Runtime Spec)
- NVIDIA Proprietary Protocol – Optimized for High Update Rate tables, this protocol operates over shared memory.

> ⚠ P4Runtime controllers can run remotely on systems with TCP/IP access to the DPU where the `dpl_rt_service` container is active. The proprietary `dpl_rt_controller` must run locally on the DPU, as it attaches to shared memory created by the `dpl_rt_service` container.

The proprietary `dpl_rt_controller` SDK provides two controller interfaces:

- `dpl_p4rt_controller` (gRPC-based) – Used to manage all data plane elements:
  - Configure pipeline (load P4 program)
  - Manage table entries (add/delete/modify, read counters, read entries)
  - Perform packet I/O (send/receive packets from the P4Runtime server)
  - Receive idle timeout notifications (for tables supporting timeouts)
- `dpl_rt_controller` (Shared Memory-based) – Used to control *High Update Rate* (HUR) table entries:
  - Add/delete/modify operations
  - Counter updates
  - Receive idle timeout notifications (for tables supporting timeouts)
  - Limited to tables marked with `nv_high_update_rate=true`

The `dpl_rt_controller` can be optionally integrated with a P4Runtime controller to offer unified table management across both regular and high-update-rate tables.

7.8.9.1.2 Table Types

The DOCA Pipeline Language (DPL) supports two categories of tables:

7.8.9.1.2.1 Regular P4 Table

- Defined per standard P4Runtime specifications.
- Managed via the gRPC-based P4Runtime controller.
- Supports all match types supported by the DPL compiler.

7.8.9.1.2.2 High Update Rate Table

- Declared like a regular table, with the **nv_high_update_rate = true** attribute. See High Update Rate Tables in the DOCA Target Architecture.
- Managed via the proprietary **dpl_rt_controller** SDK.
- Designed for optimized performance:
  - Eliminates gRPC overhead.
  - Avoids protobuf encoding.
  - Uses rule handles instead of hash-table lookups.
- Restrictions:
  - Only exact match is supported; this is enforced by the DPL Compiler.
  - Inserting duplicate entries is not rejected by **dpl_rtd**. It is the responsibility of the user's controller application to avoid inserting duplicates.
    - Each duplicate entry will receive a distinct **dpl_rt_controller_entry** handle. However, previously existing handles will be invalidated.
    - Deleting an invalidated handle will trigger a completion error. This error is silently ignored by **dpl_rtd** but is tracked in the **dpl_rt_controller_table_statistics.deletion_errors** counter for debugging purposes.
  - Attempting to modify a non-modifiable or unsupported action parameter will be silently ignored. However, this may corrupt or invalidate the output of **dpl_nspect** queries.
  - HUR tables cannot have constant entries defined in the DPL program.
  - HUR tables are only manageable via the proprietary **dpl_rt_controller** SDK:
    - No API for entry lookup. The user controller application must maintain its own map between rules and their **dpl_rt_controller_entry** handles.
    - HUR tables do not support the use of gRPC-based P4Runtime controllers for adding, deleting, modifying or reading entries.
    - Non-HUR tables can continue to use P4Runtime for management.

7.8.9.1.3 SDK Libraries Overview

The DPL SDK includes the following components:

7.8.9.1.3.1 Header Files

Path	Purpose
`/usr/include/dpl_p4rt_controller.hpp`	gRPC-based P4Runtime controller
`/usr/include/dpl_rt_controller.h`	Proprietary controller for High Update Rate tables

7.8.9.1.3.2 Shared Libraries

Library	Description
`libdpl_p4rt_proto.so`	Auto-generated P4Runtime `.proto` bindings
`libdpl_p4rt_controller.so`	gRPC implementation for P4Runtime
`libdpl_rt_controller.so`	API implementation for shared-memory-based High Update Rate tables

7.8.9.1.3.3 API Summary

dpl_p4rt_controller

Controller Class

The main interface for managing the P4 Runtime connection and state.

Method	Description
`Controller`	Constructs a new Controller object.
`~Controller`	Destructor. Cleans up the Controller object.
`ConnectUnsecured`	Establishes an unsecured connection to `dpl_rtd`.
`ConnectSecured`	Establishes a secured TLS connection to `dpl_rtd`.
`GetChannelStub`	Returns the Channel Stub object used for gRPC communication.
`GetClientReaderWriter`	Returns the `ClientReaderWriter` object for bidirectional streaming.
`LoadProgram`	Loads a P4 program onto the `dpl_rtd`.
`GetP4Info`	Retrieves the `p4::config::v1::P4Info` metadata of the currently loaded program.
`GetGrpcOps`	Retrieves gRPC callback operations for manipulating regular tables via generic `dpl_rt_controller` APIs.
`GetElectionId`	Returns the currently defined Election ID.
`GetDeviceId`	Returns the defined Device ID.

Helper Functions

Utility functions for data type conversion and formatting.

Method	Description
`GetBeByteString`	Template function that converts a value into a big-endian (network byte-order) string.
`ByteStringToHexString`	Converts a byte string into its hexadecimal string representation.

dpl_rt_controller

Context Management

Functions for initializing, connecting, and cleaning up the controller context.

Method	Description	Notes
`dpl_rt_controller_attach`	Attach to `dpl_rtd` Shared Memory (SHM).	Must be called first, once per process. If `dpl_rtd` crashes, the controller process must be restarted.
`dpl_rt_controller_detach`	Detach from `dpl_rtd` shared memory.	
`dpl_rt_controller_connect`	Creates a device context.	
`dpl_rt_controller_disconnect`	Cleanup and disconnect a device.	Disconnecting deletes all remaining High Update Rate table entries.

Entry Allocation

Functions to allocate handles for table entries before they are added.

Method	Description	Notes
`dpl_rt_controller_table_entry_alloc_shm`	Allocate a High Update Rate table entry handle from shared memory.	Supported only for tables defined with `nv_high_update_rate=true`.
`dpl_rt_controller_table_entry_alloc_grpc`	Allocate a Regular P4Runtime table entry handle using defined gRPC callbacks.	Not supported for High Update Rate tables. `dpl_rt_controller_grpc_ops_set()` must be called before using this.
`dpl_rt_controller_table_entry_free`	Free the backing memory of an entry handle.	Used only in error flows (e.g., cleaning up after a failure to add an entry).

Entry Management

Core functions for adding, modifying, and deleting table entries.

Method	Description	Notes
`dpl_rt_controller_table_entry_add`	Add a table entry.	• The `dpl_rtd` does not reject duplicate insertions to High Update Rate tables. The user app must prevent duplicates and maintain a map between rules and handles (no lookup API exists). • This API call returns `DOCA_ERROR_AGAIN` error code when the Shared Memory (SHM) requests ring/queue is full, the client application must retry calling the same operation until it gets a different error code.

Method	Description	Notes
`dpl_rt_controller_table_entry_modify`	Modify a table entry.	• Only parameters annotated with `@nv_modifiable` can be changed. • Modifying non-annotated parameters is silently ignored and may corrupt `dpl_nspect` output. • Use compiler-generated wrapper APIs instead. • This API call returns `DOCA_ERROR_AGAIN` error code when the Shared Memory (SHM) requests ring/queue is full, the client application must retry calling the same operation until it gets a different error code.
`dpl_rt_controller_table_entry_delete`	Delete a table entry.	• Deleting an entry invalidates its handle. If duplicates exist, deleting one invalidates the handles for all. Subsequent operations on invalidated handles trigger errors that are silently ignored but counted in `deletion_errors` stats. • This API call returns `DOCA_ERROR_AGAIN` error code when the Shared Memory (SHM) requests ring/queue is full, the client application must retry calling the same operation until it gets a different error code.
`dpl_rt_controller_table_entry_get_shm`	Get the opaque pointer to the C table entry structure.	Supported only for entries allocated using `alloc_shm`.
`dpl_rt_controller_table_entry_get_grpc`	Get the opaque pointer to the `p4::v1::TableEntry`.	Supported only for entries allocated using `alloc_shm`.
`dpl_rt_controller_table_entry_get_table_id`	Get the table ID of the entry.	Supported only for entries allocated using `alloc_grpc`.

gRPC Integration

Callbacks for managing standard P4Runtime tables.

Method	Description
`dpl_rt_controller_grpc_ops_set`	Sets P4Runtime controller gRPC callbacks to allow management of regular table entries.
`dpl_rt_controller_grpc_ops_clear`	Clears the currently set gRPC callbacks.

TCP State Management

Functions for managing TCP state indices.

Method	Description	Notes
`dpl_rt_controller_tcp_state_get_entries`	Get available indexes in a TCP state object for adding new entries.	Do not reuse indexes for multiple entries with the same direction. When adding new entries, call this again to get fresh indexes.

Method	Description	Notes
`dpl_rt_controller_tcp_state_query`	Query the TCP state for a given index.	

Utility & Statistics

Method	Description	Notes
`dpl_rt_controller_table_statistics`	Get table statistics.	• Supported only for tables with `nv_high_update_rate=true`. • The `insertions` counter starts at 1 due to the default entry inserted at load time.
`dpl_rt_controller_event_type_to_str`	Get string representation of an event type.	

Idle Timeout

Method	Description
`dpl_rt_controller_table_entry_modify_idle_timeout`	Modify the idle timeout of a table entry.
`dpl_rt_controller_idle_timeout_query`	Get all entries with expired idle timeout on the device.

PSP (Security & Encryption)

Functions for managing PSP Data Encryption Keys (DEK).

Method	Description	Notes
`dpl_rt_controller_psp_dek_modify`	Modify the DEK of a PSP object at a specific index.	
`dpl_rt_controller_psp_dek_modify_all`	Modify all DEKs of a PSP object.	
`dpl_rt_controller_psp_dek_query`	Query the DEK of a PSP object at a specific index.	User application must free the `key.bytes` array after use.
`dpl_rt_controller_psp_dek_query_all`	Query all DEKs of a PSP object.	User application must free the `key.bytes` array after use.
`dpl_rt_controller_generate_psp_keys`	Generate PSP keys.	User application must free the `key.bytes` array after use.

IPsec (Security & Encryption)

Method	Description	Notes
`dpl_rt_controller_ipsec_configure`	Configure IPsec object at a specific index.	• ipsec_id — the NvIPsec extern ID as it appears in the compiled p4info.txt. • index — slot number (matches the sa_index used in P4 actions). • new_ipsec_config — full SA configuration (key + params). • This call is a full replace, every field is written to hardware. Fields left at their zero value will be interpreted as zero by the hardware, except for `implict_iv` and `salt`.
`dpl_rt_controller_ipsec_configuration_query`	Query IPsec object at a specific index.	• Will invoke a read from HW. • User must call `dpl_rt_controller_ipsec_config_destroy` after usage.
`dpl_rt_controller_ipsec_config_destroy`	Destroy IPsec configuration object.	

7.8.9.1.3.4 Sample Applications

Controller sample applications can be found under:

```
/opt/dpl_rt_controller/samples/
```

7.8.9.2 Installation and System Setup

7.8.9.2.1 SDK Packages and Requirements

The DPL Runtime Controller SDK consists of the following packages:

- `dpl-rt-controller`
- `dpl-rt-controller-dev`
- `dpl-p4rt-controller`
- `dpl-p4rt-controller-dev`
- `dpl-rt-controller-samples`

> ⚠ Currently, the DPL Runtime Controller SDK supports only Ubuntu 24.04 and RHEL 9 operating systems.

7.8.9.2.2 Installation Methods

The installation process depends on your hardware deployment mode.

7.8.9.2.2.1 BlueField DPU Mode

Method	Pre-installed

Details	The SDK packages are included in the BlueField Bootstream (BFB) image. No manual installation is required after installing the BFB.

7.8.9.2.2.2 Host Mode (e.g., ConnectX-9)

Method	Manual installation

Details	Packages must be installed from the DOCA-host repository.
	For Ubuntu:
	`sudo apt install dpl-rt-controller dpl-rt-controller-dev dpl-p4rt-controller dpl-p4rt-controller-dev dpl-rt-controller-samples`
	For RHEL:
	`sudo yum install dpl-rt-controller dpl-rt-controller-devel dpl-p4rt-controller dpl-p4rt-controller-devel dpl-rt-controller-samples`

7.8.9.2.3 Hugepages Configuration

When loading DPL programs containing High Update Rate (HUR) tables, the system uses hugepages to pre-allocate entry buffers. The required buffer size depends on table properties, specifically the number of keys and total table size.

7.8.9.2.3.1 Default Behavior and Troubleshooting

The **dpl_system_setup.sh** script (see *DPL Container Deployment* under section "Pulling the Container Resources and Scripts from NGC") allocates 2048 hugepages by default.

If this allocation is insufficient for your program, **dpl_rtd** will fail to load and log an error suggesting: **(Try increasing hugepages number)**.

7.8.9.2.3.2 Adjusting the Allocation

To resolve this, run the setup script with the **--hugepages-num** flag set to a value appropriate for your workload.

For example, to load a HUR table with 16 million entries on a DPU, increase the allocation to 4096:

```
sudo <PATH_TO>/dpl_system_setup.sh --hugepages-num 4096
```

7.8.9.2.3.3 Large Allocations (>4GB)

If your configuration requires more than 4GB of hugepages (e.g., **hugepages-num > 2048** on standard 2MB hugepages), you must also update the container resource limits.

1. Open **dpl_rt_service.yaml**.
2. Locate the section **spec** → **resources** → **limits**.
3. Increase the limit for **hugepages-2Mi** to match your new allocation.

7.8.9.3 DPL P4Runtime Controller Library

```
namespace DPL_P4RT_Controller {

/* ------------------------------------------------------------------------
 * Controller class for connecting to dpl_rtd over gRPC.
 * ------------------------------------------------------------------------ */
class Controller {
public:
    /*
     * ------------------------------------------------------------------------
     * gRPC APIs
     * ------------------------------------------------------------------------
     */

    /**
     * @brief Construct a new Controller object.
     *
     * Assigns default election IDs of (high = 0, low = 1).
     *
     * @param [in] device_id
     * The Device ID to connect to in the dpl_rtd.
     * @param [in] ipaddr
     * The address for connecting to dpl_rtd
     */
    DOCA_EXPERIMENTAL
    Controller(uint32_t device_id, const char *ipaddr);

    /**
     * @brief Construct a new Controller object with custom election IDs.
     *
     * @param [in] device_id
     * The Device ID to connect to in the dpl_rtd.
     * @param [in] ipaddr
     * The address for connecting to dpl_rtd
     * @param [in] election_id_high
     * High part of the 128bit election ID used to determine the primary controller.
     * @param [in] election_id_low
     * Low part of the 128bit election ID used to determine the primary controller.
     */
    DOCA_EXPERIMENTAL
    Controller(uint32_t device_id, const char *ipaddr, uint64_t election_id_high, uint64_t
election_id_low);

    /**
     * @brief Destroy the Controller object
     *
     */
    DOCA_EXPERIMENTAL
    ~Controller();

    /**
     * @brief Create an unsecured connection to dpl_rtd.
     *
     * This requires that TLS authentication was not enabled on dpl_rtd P4RT_RPC_SERVER.
     *
     * Uses grpc::InsecureChannelCredentials().
     *
     * @return doca_error_t :
     * DOCA_SUCCESS - in case of success.
     * Error code - in case of failure.
     */
    DOCA_EXPERIMENTAL
    doca_error_t ConnectUnsecured(void);

    /**
     * @brief Create a secured TLS connection to dpl_rtd.
     *
     * This requires that TLS authentication was properly enabled and configured on dpl_rtd
P4RT_RPC_SERVER.
     *
     * Uses grpc::SslCredentials().
     *
     * @param [in] ca_crt_path
     * Path to ca.crt (cacert).
     * @param [in] client_key_path
     * Path to client.key (cert).
     * @param [in] client_crt_path
     * Path to client.crt (private-key).
     * @return doca_error_t :
     * DOCA_SUCCESS - in case of success.
     * Error code - in case of failure.
     */
    DOCA_EXPERIMENTAL
    doca_error_t ConnectSecured(const char *ca_crt_path, const char *client_key_path, const char
 *client_crt_path);

    /**
     * @brief Get the Channel Stub object for communicating with the dpl_rtd.
     *
     * @return const p4::v1::P4Runtime::Stub* :
     * Valid pointer - in case of success.
     * NULL - in case of failure.
     */
    DOCA_EXPERIMENTAL
    const p4::v1::P4Runtime::Stub *GetChannelStub();

    /**
     * @brief Get the ClientReaderWriter object for communicating with the dpl_rtd.
```

```
     *
     * @return p4::v1::P4Runtime::Stub* :
     * Valid pointer - in case of success.
     * NULL - in case of failure.
     */
    DOCA_EXPERIMENTAL
    grpc::ClientReaderWriter<p4::v1::StreamMessageRequest, p4::v1::StreamMessageResponse>
*GetClientReaderWriter();

    /*
     * ------------------------------------------------------------------------
     * Utility methods
     * ------------------------------------------------------------------------
     */

    /**
     * @brief Load a program on the dpl_rtd.
     *
     * @param [in] p4info_path
     * Path to the compiled p4info.txt file of the program.
     * @param [in] blob_path
     * Path to the compiled dplconfig blob file of the program.
     * @return doca_error_t :
     * DOCA_SUCCESS - in case of success.
     * Error code - in case of failure.
     */
    DOCA_EXPERIMENTAL
    doca_error_t LoadProgram(const char *p4info_path, const char *blob_path);

    /**
     * @brief Get the p4::config::v1::P4Info of the loaded program.
     *
     * @return const p4::config::v1::P4Info* :
     * Valid pointer - in case of success.
     * NULL - in case of failure.
     */
    DOCA_EXPERIMENTAL
    const p4::config::v1::P4Info *GetP4Info();

    /*
     * ------------------------------------------------------------------------
     * Infrastructure methods
     * ------------------------------------------------------------------------
     */

    /**
     * @brief Get gRPC operations callbacks for working on regular tables from the generic
dpl_rt_controller APIs.
     *
     * @return struct dpl_rt_controller_grpc_ops* :
     * Valid pointer - in case of success.
     * NULL - in case of failure.
     */
    DOCA_EXPERIMENTAL
    const struct dpl_rt_controller_grpc_ops *GetGrpcOps(void);

    /**
     * @brief Get the defined Election IDs.
     */
    DOCA_EXPERIMENTAL
    void GetElectionId(uint64_t *high, uint64_t *low);

    /**
     * @brief Get the defined device_id.
     *
     * @param [out] device_id
     */
    DOCA_EXPERIMENTAL
    uint32_t GetDeviceId(void);

private:
    uint32_t device_id_;
    const char *rtd_ipaddr_;
    uint64_t election_id_high_;
    uint64_t election_id_low_;
    std::shared_ptr<grpc::Channel> channel_;
    std::unique_ptr<p4::v1::P4Runtime::Stub> stub_;
    p4::config::v1::P4Info p4info_;
    std::unique_ptr<grpc::ClientReaderWriter<p4::v1::StreamMessageRequest,
p4::v1::StreamMessageResponse>> stream_;
    grpc::ClientContext stream_context_;
    struct dpl_rt_controller_grpc_ops grpc_ops_;

    doca_error_t Connect(std::shared_ptr<grpc::ChannelCredentials> creds);
};

/* ------------------------------------------------------------------------
 * Helpers for working with Bytestrings.
 * ------------------------------------------------------------------------ */

/**
 * @brief Template for getting a string object representing converted value to big-endian (i.e.
network) byte-order.
 *
 * This must be used when setting match key and action parameter values on gRPC table entries.
 */
template <typename T>
std::string GetBeByteString(T i, size_t bits)
{
    // Ensure T is an integral type
    static_assert(std::is_integral<T>::value, "Template parameter must be an integral type");

    if constexpr (sizeof(T) == sizeof(uint16_t)) {
        i = htons(i);
    } else if constexpr (sizeof(T) == sizeof(uint32_t)) {
```

```
        i = htonl(i);
    } else if constexpr (sizeof(T) == sizeof(uint64_t)) {
        if (*reinterpret_cast<const char *>("\0\x01") == 0) { // check for little endian
            const uint32_t high_part = htonl(static_cast<uint32_t>(i >> 32));
            const uint32_t low_part = htonl(static_cast<uint32_t>(i & 0xFFFFFFFFLL));
            i = (static_cast<uint64_t>(low_part) << 32) | high_part;
            i = i >> (64 - bits);
        }
    } // else: 8-bits or byte-array assumed already to be in big-endian (i.e. network) byte-order

    size_t bytes = (bits + 7) / 8;
    return std::string(reinterpret_cast<char *>(&i), bytes);
};

/**
 * @brief Template for getting a string object representing converted value to big-endian (i.e.
network) byte-order.
 *
 * This must be used when setting match key and action parameter values on gRPC table entries.
 */
template <typename T>
std::string GetBeByteString(T i)
{
    return GetBeByteString(i, sizeof(i) * 8);
}

/**
 * @brief Template for getting a string object from a big-endian array.
 *
 * @note Array is assumed to be in big-endian (i.e. network) byte-order, no byte-order conversion is
performed.
 *
 * This must be used when setting match key and action parameter values on gRPC table entries.
 */
template <typename T, size_t N>
std::string GetByteString(T (&addr)[N])
{
    return std::string(reinterpret_cast<const char *>(addr), N * sizeof(T));
}

/**
 * @brief Get Hex string representation of a byte string value.
 *
 * @param [in] value
 * Value to translate.
 * @return std::string : Hex string representation
 */
std::string ByteStringToHexString(const std::string &value);

}; // namespace DPL_P4RT_Controller
```

7.8.9.4 DPL Runtime Controller Library

```
/**
 * @brief Byte array type.
 */
struct byte_array {
    uint8_t *bytes;
    size_t bytes_num;
};

/* Opaque structures. */
struct dpl_rt_controller_device;
struct dpl_rt_controller_entry;

/** Possible event types. */
enum dpl_rt_controller_event_type {
    /** A new program with High Update Rate tables was loaded. */
    DPL_EVENT_PROGRAM_LOADED,
    /** Program was unloaded; need to cleanup and reconnect once new program is loaded. */
    DPL_EVENT_PROGRAM_UNLOADED,
    /** Entry addition failed. */
    DPL_EVENT_ERROR_ENTRY_ADD,
    /** The dpl_rtd crashed; the controller must be restarted. */
    DPL_EVENT_ERROR_DETACHED,
};

/**
 * @brief Maximum length of a status messages.
 */
#define DPL_RT_MAX_STATUS_MESSAGE_LEN 128

struct dpl_rt_controller_event_data {
    /** Optional details message. */
    char message[DPL_RT_MAX_STATUS_MESSAGE_LEN];
    /** Event data for applicable types. */
    union {
        struct {
            /** Entry that failed to be added. */
            struct dpl_rt_controller_entry *entry;
        } entry_add;
    };
};

/** Event type and data. */
struct dpl_rt_controller_event {
    /** The device that raised the event. */
    uint32_t device_id;
    /** Event type. */
```

```
47      enum dpl_rt_controller_event_type type;
48      /** Event data. */
49      struct dpl_rt_controller_event_data data;
50  };
51
52  /** Event handling callback prototype. */
53  typedef void (*dpl_rt_controller_event_cb_t)(struct dpl_rt_controller_event *event);
54
55  /**
56   * @brief Attributes for attaching to dpl_rtd shared memory.
57   */
58  struct dpl_rt_controller_context_attr {
59      /** Events handling callback, to be called when events arise.  */
60      dpl_rt_controller_event_cb_t event_cb;
61  };
62
63  /**
64   * @brief Attributes for connecting to a device.
65   */
66  struct dpl_rt_controller_device_attr {
67      /** The Device ID to connect to in the dpl_rtd. */
68      uint32_t device_id;
69      /** Program hash from the DPL auto-generated header.
70       * Required to manage High Update Rate, set to <PROGRAM_NAME>_HASH defined by the DPL compiler
    generated header.
71       * Required when without_shm_support=false.
72       */
73      const char *program_hash;
74      /** When set to true, do not connect over SHM, create a device that supports only regular P4Runtime
    tables,
75       * without the ability to manage High Update Rate tables.
76       * Used to allow using the generic entry management APIs when Update Rate tables not required. */
77      bool without_shm_support;
78  };
79
80  /**
81   * @brief Counter data.
82   */
83  struct dpl_rt_controller_counter_data {
84      /** Counter packets count value. */
85      uint64_t packet_count;
86      /** Counter bytes count value. */
87      uint64_t byte_count;
88  };
89
90  /**
91   * @brief The direction of the packet with respect to the TCP connection establishment.
92   * The sender of the first SYN packet is the originator of the flow.
93   * The receiver is the direction that provides the SYN-ACK reply.
94   */
95  enum dpl_rt_controller_tcp_state_direction {
96      DPL_TCP_STATE_DIRECTION_ORIGINAL = 0,
97      DPL_TCP_STATE_DIRECTION_REPLY = 1,
98  };
99
100 /**
101  * @brief A TCP state (Connection Tracking) data.
102  */
103 struct dpl_rt_controller_tcp_state_data {
104     /** The direction of the last packet that hit the TCP state object. */
105     enum dpl_rt_controller_tcp_state_direction last_direction;
106     /** TCP acknowledgement (ACK) number of the last packet that hit the TCP state object. */
107     uint32_t last_ack_num;
108     /** TCP sequence (SEQ) number of the last packet that hit the TCP state object. */
109     uint32_t last_seq_num;
110 };
111
112 /**
113  * @brief Supported DEK key types.
114  */
115 enum dpl_rt_controller_dek_key_type {
116     DPL_DEK_KEY_TYPE_PSP,
117 };
118
119 /**
120  * @brief Supported DEK key sizes.
121  */
122 enum dpl_rt_controller_dek_key_size {
123     DPL_DEK_KEY_SIZE_128,
124     DPL_DEK_KEY_SIZE_256,
125 };
126
127 /**
128  * @brief A DEK config.
129  */
130 struct dpl_rt_controller_dek_config {
131     enum dpl_rt_controller_dek_key_type key_type;
132     enum dpl_rt_controller_dek_key_size key_size;
133     struct byte_array key;
134 };
135
136 /**
137  * @brief Opaque type for passing p4::v1::TableEntry pointer.
138  */
139 typedef void *p4_v1_table_entry_ptr_t;
140
141 /**
142  * @brief Attributes for adding a table entry.
143  *
144  * Only applicable when table was defined with idle timeout support
145  * (i.e.: `nv_support_timeout = true; nv_delayed_counter_stats = true;` in p4 program)
146  */
147 struct dpl_rt_controller_add_entry_attr {
148     /** Idle timeout value in nanoseconds. */
149     uint64_t idle_timeout_ns;
```

```
};

/**
 * @brief gRPC function callbacks required to allow managing gRPC tables using the dpl_rt_controller APIs.
 */
struct dpl_rt_controller_grpc_ops {
    /** Opaque user data pointer, e.g for passing DPL_P4RT_Controller::Controller pointer. */
    void *user_data;
    /** Callback for allocating p4::v1::TableEntry. */
    doca_error_t (*grpc_entry_alloc)(void *user_data, uint32_t table_id, p4_v1_table_entry_ptr_t *entry);
    /** Callback for freeing p4::v1::TableEntry handler */
    void (*grpc_entry_free)(void *user_data, p4_v1_table_entry_ptr_t entry);
    /** Callback for adding p4::v1::TableEntry to loaded program */
    doca_error_t (*grpc_entry_add)(void *user_data,
                        p4_v1_table_entry_ptr_t entry,
                        struct dpl_rt_controller_add_entry_attr *attr);
    /** Callback for deleting p4::v1::TableEntry from loaded program */
    doca_error_t (*grpc_entry_delete)(void *user_data, p4_v1_table_entry_ptr_t entry);
    /** Callback for reading DirectCounter of p4::v1::TableEntry */
    doca_error_t (*grpc_entry_counter)(void *user_data,
                        p4_v1_table_entry_ptr_t entry,
                        uint64_t *byte_count,
                        uint64_t *packet_count);
    /** Callback for reading DirectCounter of table's default entry */
    doca_error_t (*grpc_default_entry_counter)(void *user_data,
                            uint32_t table_id,
                            uint64_t *byte_count,
                            uint64_t *packet_count);
    /** Callback for modifying the idle timeout of a table entry */
    doca_error_t (*grpc_entry_modify_idle_timeout)(void *user_data,
                                p4_v1_table_entry_ptr_t entry,
                                uint64_t new_idle_timeout_ns);
    /** Callback for modifying the DEK of a PSP entry */
    doca_error_t (*grpc_entry_psp_dek_modify)(void *user_data,
                            uint32_t psp_id,
                            bool modify_all,
                            uint32_t index,
                            dpl_rt_controller_dek_config *new_dek_config);
    /** Callback for querying the DEK of a PSP entry */
    doca_error_t (*grpc_entry_psp_dek_query)(void *user_data,
                            uint32_t psp_id,
                            uint32_t index,
                            size_t dek_configs_num,
                            dpl_rt_controller_dek_config *dek_configs);
};

/**
 * @brief Table statistics collected by the dpl_rtd and the controller.
 */
struct dpl_rt_controller_table_statistics_data {
    // -- Maintained by the dpl_rtd side:
    uint64_t insertions;       // The total number of successful entries added to the table.
    uint64_t insertion_errors; // The total number of failed entries additions.
    uint64_t deletions;    // The total number of entries successfully removed from the table.
    uint64_t deletion_errors;  // The total number of failed entries deletions.
    uint64_t noops; // Del entry requested before it was added (both ADD and DEL flags set while entry in INIT).
    uint64_t pending_add_dels; // How many entry delete received while entry was still PENDING_ADD.

    // -- Maintained by controller library side:
    uint64_t entry_add_errors; // When the entry_add function returns a failure.
    uint64_t entry_del_errors; // When the entry_del function returns a failure.
};

/* ------------------------------------------------------------------------
 * Context management methods.
 * ------------------------------------------------------------------------ */

/**
 * @brief Attach to dpl_rtd SHared Memory (SHM).
 *
 * @note This must be called first thing, once per process.
 * In case that th dpl_rtd crashes, the controller process must be restarted.
 *
 * @param [in] attr
 * Attributes for attaching.
 * @return doca_error_t :
 * DOCA_SUCCESS - in case of success.
 * DOCA_ERROR_UNSUPPORTED_VERSION - SHM API mismatch; the dpl_rtd and the controller library were compiled using
 * different shared memory API versions.
 * DOCA_ERROR_ALREADY_EXIST - in case that attach is called more than once.
 * DOCA_ERROR_INITIALIZATION - in case of initialization errors.
 * Error code - in case of failure.
 */
DOCA_EXPERIMENTAL
doca_error_t dpl_rt_controller_attach(struct dpl_rt_controller_context_attr *attr);

/**
 * @brief Detach from dpl_rtd shared memory.
 *
 * @return doca_error_t :
 * DOCA_SUCCESS - in case of success.
 * Error code - in case of failure.
 */
DOCA_EXPERIMENTAL
doca_error_t dpl_rt_controller_detach(void);

/**
 * @brief Creates a device context
 *
 * There two modes when creating a device:
 *   1. attr.without_shm_support = false:
```

```
 *     - Connects to a device over shared memory, to support managing High Update Rate tables.
 *     - Requires calling dpl_rt_controller_attach() first.
 *     - Optional support for regular P4Runtime tables can be added by calling dpl_rt_controller_grpc_ops_set().
 *   2. attr.without_shm_support = true:
 *     - Used when user only wants use the generic APIs to manage regular P4Runtime tables.
 *     - Does not connect to a device over shared memory; No support for managing High Update Rate tables.
 *     - Requires calling dpl_rt_controller_grpc_ops_set().
 *     - No need to call dpl_rt_controller_attach() at all.
 *
 * @note Only one client can be connected to a device over shared memory.
 *
 * @param [in] attr
 * Attributes for connecting.
 * @param [out] device
 * Pointer to created device handle.
 * @return doca_error_t :
 * DOCA_SUCCESS - in case of success.
 * DOCA_ERROR_AGAIN - the device is not loaded with a program yet.
 * DOCA_ERROR_NOT_SUPPORTED - the device is loaded with a program that does not contain High Update Rate tables.
 * DOCA_ERROR_UNSUPPORTED_VERSION - program_hash mismatch; the device is loaded with a different program version.
 * DOCA_ERROR_IN_USE - another client is already connected to this device.
 * Error code - in case of other failures.
 */
DOCA_EXPERIMENTAL
doca_error_t dpl_rt_controller_connect(struct dpl_rt_controller_device_attr *attr,
                       struct dpl_rt_controller_device **device);

/**
 * @brief Cleanup and disconnect a device
 *
 * @note After disconnecting, the dpl_rtd will delete remaining High Update Rate table entries!
 *
 * @param [in] ctx
 * Device to disconnect.
 * @return doca_error_t :
 * DOCA_SUCCESS - in case of success.
 * Error code - in case of failure.
 */
DOCA_EXPERIMENTAL
doca_error_t dpl_rt_controller_disconnect(struct dpl_rt_controller_device *device);

/* ------------------------------------------------------------------------
 * Entry allocation methods.
 * ------------------------------------------------------------------------ */

/**
 * @brief Allocate a High Update Rate table entry handle from shared memory.
 *
 * @note Supported only for tables defined with attribute nv_high_update_rate=true in the program.
 *
 * @param [in] device
 * Device to allocate from.
 * @param [in] table_id
 * High Update Rate table ID to allocate entry from.
 * @param [out] entry
 * Pointer for providing the dpl_rt_controller_entry handler.
 * @param [out] opaque_entry
 * Opaque pointer to the C table entry structure backing this entry.
 * @return doca_error_t :
 * DOCA_SUCCESS - in case of success.
 * DOCA_ERROR_NO_MEMORY - in case of memory allocation failure.
 * Error code - in case of failure.
 */
DOCA_EXPERIMENTAL
doca_error_t dpl_rt_controller_table_entry_alloc_shm(struct dpl_rt_controller_device *device,
                             uint32_t table_id,
                             struct dpl_rt_controller_entry **entry,
                             void **opaque_entry);

/**
 * @brief Allocate a regular P4Runtime table entry handle using defined gRPC callbacks.
 *
 * @note Not supported for tables defined with attribute nv_high_update_rate=true in the program.
 * @note dpl_rt_controller_grpc_ops_set() must be called prior using this API.
 *
 * @param [in] device
 * Device to allocate from.
 * @param [in] table_id
 * Regular P4Runtime table ID to allocate entry from.
 * @param [out] entry
 * @param [out] opaque_entry
 * opaque pointer to p4::v1::TableEntry backing this entry.
 * Pointer for providing the dpl_rt_controller_entry handler.
 * @return doca_error_t :
 * DOCA_SUCCESS - in case of success.
 * DOCA_ERROR_NO_MEMORY - in case of memory allocation failure.
 * Error code - in case of failure.
 */
DOCA_EXPERIMENTAL
doca_error_t dpl_rt_controller_table_entry_alloc_grpc(struct dpl_rt_controller_device *device,
                              uint32_t table_id,
                              struct dpl_rt_controller_entry **entry,
                              p4_v1_table_entry_ptr_t *opaque_entry);

/**
 * @brief Free backing memory of entry handler.
 *
 * @note This is used only in error flows, e.g. after failure to add an entry.
 *
 * @param [in] entry
 * Entry to free.
```

```
352  */
353 DOCA_EXPERIMENTAL
354 void dpl_rt_controller_table_entry_free(struct dpl_rt_controller_entry *entry);
355
356 /* ------------------------------------------------------------------------
357  * Entry management methods.
358  * ------------------------------------------------------------------------ */
359
360 /**
361  * @brief Add a table entry.
362  *
363  * The API is asynchronous non-blocking when adding High Update Rate table entries.
364  * Returning success means that the entry was sent to the dpl_rtd for insertion.
365  * In case of a successful insertion, there is no further feedback from the dpl_rtd.
366  * In case of a failed insertion, an event of type DPL_EVENT_ERROR_ENTRY_ADD is raised.
367  * It is possible to verify the entry insertion status by checking the table statistics using the
368  * dpl_rt_controller_table_statistics() API.
369  *
370  * The API is synchronous blocking when adding a regular table entries, as it is done using P4Runtime
RPC message.
371  * The returned status reflects whether the dpl_rtd inserted the entry successfully or not.
372  *
373  * @note Inserting duplicate entries to High Update Rate tables is not rejected by the dpl_rtd, it is
the user
374  *       controller app responsibility to avoid inserting duplicate entries.
375  * @note There is no API for entry lookup, it is the user controller app responsibility to maintain a
map between the
376  *       rule and its dpl_rt_controller_entry handle.
377  *
378  * @param [in] entry
379  * Entry to add.
380  * @param [in] attr
381  * Attributes for adding the entry, see dpl_rt_controller_add_entry_attr for more details.
382  * @note Can be NULL if not needed.
383  * @return doca_error_t :
384  * DOCA_SUCCESS - in case of success.
385  * DOCA_ERROR_AGAIN - in case posting the request failed due to full requests ring.
386  * DOCA_ERROR_IN_PROGRESS - in case that another addition operation is still in progress.
387  * DOCA_ERROR_EMPTY - in case entry does not exist (entry is already deleted/freed).
388  * Error code - in case of failure.
389  */
390 DOCA_EXPERIMENTAL
391 doca_error_t dpl_rt_controller_table_entry_add(struct dpl_rt_controller_entry *entry,
392                                struct dpl_rt_controller_add_entry_attr *attr);
393
394 /**
395  * @brief Delete a table entry.
396  *
397  * The API is asynchronous non-blocking when deleting High Update Rate table entries.
398  * Returning success means that the entry was sent to the dpl_rtd for deletion.
399  * In case of a successful deletion, there is no further feedback from the dpl_rtd.
400  * In case of a failed deletion, the relevant statistic counters are incremented (no even is raised).
401  * It is possible to verify the entry deletion status by checking the table statistics using the
402  * dpl_rt_controller_table_statistics() API.
403  *
404  * The API is synchronous blocking when deleting a regular table entries, as it is done using P4Runtime
RPC message.
405  * The returned status reflects whether the dpl_rtd deleted the entry successfully or not.
406  *
407  * @note If a duplicate entries were inserted, once any such entry is deleted, the existing handles
will be invalidated.
408  *       As such, a completion error will be triggered by entry deletion of the invalidated handle. The
error is
409  *       silently ignored by the dpl_rtd and counted by
dpl_rt_controller_table_statistics_data.deletion_errors counter
410  *       for debug purposes.
411  *
412  * @param [in] entry
413  * Entry to delete.
414  * @return doca_error_t :
415  * DOCA_SUCCESS - in case of success.
416  * DOCA_ERROR_AGAIN - in case posting the request failed due to full requests ring.
417  * DOCA_ERROR_IN_PROGRESS - in case that another deletion operation is still in progress.
418  * DOCA_ERROR_EMPTY - in case entry does not exist (entry is already deleted/freed).
419  * Error code - in case of failure.
420  */
421 DOCA_EXPERIMENTAL
422 doca_error_t dpl_rt_controller_table_entry_delete(struct dpl_rt_controller_entry *entry);
423
424 /**
425  * @brief Read entry direct counter data.
426  *
427  * The client application is responsible for synchronizing the entry_delete and entry_counter calls.
428  *
429  * @note Supported only if direct_counter was defined on the parent table in the program.
430  * @note This API is always blocking.
431  *
432  * @param [in] entry
433  * Entry to read its counter data.
434  * @param [out] counter
435  * Returned entry counter data.
436  * @return doca_error_t :
437  * DOCA_SUCCESS - in case of success.
438  * DOCA_ERROR_AGAIN - in case posting the request failed due to full requests ring.
439  * DOCA_ERROR_IN_PROGRESS - in case that another query is still in progress.
440  * DOCA_ERROR_NOT_FOUND - in case direct_counter was not defined on the table in the program.
441  * DOCA_ERROR_EMPTY - in case entry does not exist (entry is already deleted/freed).
442  * Error code - in case of failure.
443  */
444 DOCA_EXPERIMENTAL
445 doca_error_t dpl_rt_controller_table_entry_counter(struct dpl_rt_controller_entry *entry,
446                                struct dpl_rt_controller_counter_data *counter);
447
448 /**
449  * @brief Read table's default entry direct counter data.
```

```
 *
 * @note Supported only if direct_counter was defined on the parent table in the program.
 * @note This API is always blocking.
 *
 * @param [in] device
 * Device to read its default entry counter data.
 * @param [in] table_id
 * Table ID to read its default entry counter data.
 * Can be either a High Update Rate table ID or a regular P4Runtime table ID.
 * @param [out] counter
 * Returned entry counter data.
 * @return doca_error_t :
 * DOCA_SUCCESS - in case of success.
 * DOCA_ERROR_AGAIN - in case posting the request failed due to full requests ring.
 * DOCA_ERROR_IN_PROGRESS - in case that another query is still in progress.
 * DOCA_ERROR_NOT_FOUND - in case direct_counter was not defined on the table in the program.
 * Error code - in case of failure.
 */
DOCA_EXPERIMENTAL
doca_error_t dpl_rt_controller_table_default_entry_counter(struct dpl_rt_controller_device *device,
                                uint32_t table_id,
                                struct dpl_rt_controller_counter_data *counter);

/**
 * @brief Get the opaque pointer to the C table entry structure backing this entry.
 *
 * The returned opaque pointer points to memory buffer for providing the table entry details (keys,
params, etc),
 * which must be casted to the corresponding table entry C structure defined at the C header generated
by the DPL
 * Compiler.
 *
 * @note Supported only for entries allocated using dpl_rt_controller_table_entry_alloc_shm().
 *
 * @param [in] entry
 * dpl_rt_controller_entry
 * @return void** :
 * Valid pointer for providing the opaque entry structure pointer - in case of success.
 * NULL - in case of failure.
 */
DOCA_EXPERIMENTAL
void *dpl_rt_controller_table_entry_get_shm(struct dpl_rt_controller_entry *entry);

/**
 * @brief Get the opaque pointer to p4::v1::TableEntry backing this entry.
 *
 * The returned opaque pointer points to a p4::v1::TableEntry for providing the table entry details
(keys, params, etc).
 *
 * @note Supported only for entries allocated using dpl_rt_controller_table_entry_alloc_grpc().
 *
 * @param [in] entry
 * dpl_rt_controller_entry
 * @return void** :
 * Valid pointer for providing the p4::v1::TableEntry pointer - in case of success.
 * NULL - in case of failure.
 */
DOCA_EXPERIMENTAL
p4_v1_table_entry_ptr_t dpl_rt_controller_table_entry_get_grpc(struct dpl_rt_controller_entry *entry);

/* ------------------------------------------------------------------------
 * gRPC integration methods.
 * ------------------------------------------------------------------------ */

/**
 * @brief Set P4Runtime controller gRPC callbacks for allowing managing regular table entries.
 *
 * @param [in] device
 * Device to set the gRPC callback on.
 * @param [in] ops
 * The set of user's gRPC callbacks.
 * @return doca_error_t :
 * DOCA_SUCCESS - in case of success.
 * Error code - in case of failure.
 */
DOCA_EXPERIMENTAL
doca_error_t dpl_rt_controller_grpc_ops_set(struct dpl_rt_controller_device *device,
                        const struct dpl_rt_controller_grpc_ops *ops);

/**
 * @brief Clear the currently set gRPC callbacks.
 *
 * @param [in] device
 * Device to clear its gRPC callbacks.
 */
DOCA_EXPERIMENTAL
void dpl_rt_controller_grpc_ops_clear(struct dpl_rt_controller_device *device);

/*
 * ------------------------------------------------------------------------
 * TCP state methods.
 * ------------------------------------------------------------------------
 */

/**
 * @brief Get avaiable indexes in a TCP state object for adding new entries.
 *
 * Returns indexes that can be be used to add new entries using the TCP state object.
 *
 * @note The user application is responsible to make sure not to use the same index for multiple
entries using the same
 *       direction.
 * @note The user application is responsible to make sure not to reuse indexes after deleting entries
using them.
```

```
549  *       For adding new entries, need to call dpl_rt_controller_tcp_state_get_entries() again to get a
     new avaiable
550  *       index.
551  * @note The dpl_rtd service will automatically track and reset no longer used indexes when the entry
     is deleted.
552  *
553  * @param [in] tcp_state_id
554  * TCP state object ID to get the number of entries.
555  * @param [in, out] indexes
556  * Array of indexes to return the available unused indexes.
557  * @param [in] indexes_num
558  * Requested number of indexes to return.
559  * @param [out] indexes_num_ret
560  * Actual number of indexes returned.
561  * @return doca_error_t :
562  * DOCA_SUCCESS - in case of success.
563  * Error code - in case of failure.
564  */
565 DOCA_EXPERIMENTAL
566 doca_error_t dpl_rt_controller_tcp_state_get_entries(struct dpl_rt_controller_device *device,
567                                uint32_t tcp_state_id,
568                                uint32_t *indexes,
569                                uint32_t indexes_num,
570                                uint32_t *indexes_num_ret);
571
572 /**
573  * @brief Query the TCP state for a given index.
574  *
575  * @param [in] device
576  * Device containing the requested TCP state.
577  * @param [in] tcp_state_id
578  * TCP state object ID to query.
579  * @param [in] index
580  * Index to query.
581  * @param [out] state
582  * Pointer to struct dpl_rt_controller_tcp_state_data to return the state data.
583  * @return doca_error_t :
584  * DOCA_SUCCESS - in case of success.
585  * Error code - in case of failure.
586  */
587 DOCA_EXPERIMENTAL
588 doca_error_t dpl_rt_controller_tcp_state_query(struct dpl_rt_controller_device *device,
589                               uint32_t tcp_state_id,
590                               uint32_t index,
591                               struct dpl_rt_controller_tcp_state_data *state);
592
593 /*
594  * ---------------------------------------------------------------------------
595  * Utility methods
596  * ---------------------------------------------------------------------------
597  */
598
599 /**
600  * @brief Get table statistics.
601  *
602  * @note Supported only for tables defined with attribute nv_high_update_rate=true in the program.
603  * @note After a new DPL program is loaded, the 'insertions' counter will have value 1 since a default
     table entry
604  *       was inserted to the High Update Rate table during loading the program.
605  *
606  * @param [in] device
607  * Device containing the requested table.
608  * @param [in] table_id
609  * Table ID to get it's statistics.
610  * @param [out] stats
611  * Pointer to struct dpl_rt_controller_table_statistics_data to return the statistics.
612  * @return doca_error_t :
613  * DOCA_SUCCESS - in case of success.
614  * Error code - in case of failure.
615  */
616 DOCA_EXPERIMENTAL
617 doca_error_t dpl_rt_controller_table_statistics(struct dpl_rt_controller_device *device,
618                                 uint32_t table_id,
619                                 struct dpl_rt_controller_table_statistics_data *stats);
620
621 /**
622  * @brief Get string representation of a dpl_rt_controller_event_type.
623  *
624  * @param [in] type
625  * Event type value to convert.
626  * @return const char* :
627  * String representation.
628  */
629 DOCA_EXPERIMENTAL
630 const char *dpl_rt_controller_event_type_to_str(enum dpl_rt_controller_event_type type);
631
632 /**
633  * @brief Modify the idle timeout of a table entry.
634  *
635  * @param [in] entry
636  * Entry to modify the idle timeout of.
637  * @param [in] new_idle_timeout_ns
638  * New idle timeout value in nanoseconds.
639  * @return doca_error_t :
640  * DOCA_SUCCESS - in case of success.
641  * DOCA_ERROR_NOT_SUPPORTED - if the table does not support idle timeout.
642  * DOCA_ERROR_EMPTY - in case entry does not exist (entry is already deleted/freed).
643  * Error code - in case of failure.
644  */
645 DOCA_EXPERIMENTAL
646 doca_error_t dpl_rt_controller_table_entry_modify_idle_timeout(struct dpl_rt_controller_entry *entry,
647                                       uint64_t new_idle_timeout_ns);
648
649 /**
650  * @brief Get all entries with expired idle timeout on the device.
```

```
651      *
652      * This API will fill the stale_entry_arr array with pointers to stale entries on High Update Rate
         tables.
653      *
654      * @param [in] device
655      * Device to query idle timeout for.
656      * @param [in, out] stale_entry_arr
657      * Array of stale entries to be filled - should be allocated and passed empty by user.
658      * @param [in] entry_arr_size
659      * Size of the stale_entry_arr array.
660      * @param [out] stale_entries_num
661      * Total number of stale entries found (even if exceeds entry_arr_size).
662      * @return doca_error_t :
663      * DOCA_SUCCESS - in case of success.
664      * DOCA_ERROR_NOT_FOUND - in case that the device does not have any tables supporting idle timeout.
665      * Error code - in case of failure.
666      */
667     DOCA_EXPERIMENTAL
668     doca_error_t dpl_rt_controller_idle_timeout_query(struct dpl_rt_controller_device *device,
669                             struct dpl_rt_controller_entry **stale_entry_arr,
670                             size_t entry_arr_size,
671                             size_t *stale_entries_num);
672
673     /**
674      * @brief Modify the DEK (Data Encryption Key) of a PSP object at given index.
675      *
676      * @param [in] device
677      * Device containing the requested PSP.
678      * @param [in] psp_id
679      * PSP object ID to modify the DEK of.
680      * @param [in] index
681      * Index to modify the DEK of.
682      * @param [in] new_dek_config
683      * New DEK config.
684      * @return doca_error_t :
685      * DOCA_SUCCESS - in case of success.
686      * Error code - in case of failure.
687      */
688     DOCA_EXPERIMENTAL
689     doca_error_t dpl_rt_controller_psp_dek_modify(struct dpl_rt_controller_device *device,
690                             uint32_t psp_id,
691                             uint32_t index,
692                             struct dpl_rt_controller_dek_config *new_dek_config);
693
694     /**
695      * @brief Modify all DEKs (Data Encryption Keys) of a PSP object.
696      *
697      * @param [in] device
698      * Device containing the requested PSP.
699      * @param [in] psp_id
700      * PSP object ID to modify the DEKs of.
701      * @param [in] new_dek_config
702      * New DEK config.
703      * @return doca_error_t :
704      * DOCA_SUCCESS - in case of success.
705      * Error code - in case of failure.
706      */
707     DOCA_EXPERIMENTAL
708     doca_error_t dpl_rt_controller_psp_dek_modify_all(struct dpl_rt_controller_device *device,
709                             uint32_t psp_id,
710                             struct dpl_rt_controller_dek_config *new_dek_config);
711
712     /**
713      * @brief Query the DEK (Data Encryption Key) of a PSP object at given index.
714      *
715      * @param [in] device
716      * Device containing the requested PSP.
717      * @param [in] psp_id
718      * PSP object ID to query the DEK of.
719      * @param [in] index
720      * Index to query the DEK of.
721      * @param [out] dek_config
722      * Pointer to DEK config to store the result.
723      * @note The user application is responsible to free the key.bytes array after using it.
724      * @return doca_error_t :
725      * DOCA_SUCCESS - in case of success.
726      * Error code - in case of failure. No need to free the key.bytes array.
727      */
728     DOCA_EXPERIMENTAL
729     doca_error_t dpl_rt_controller_psp_dek_query(struct dpl_rt_controller_device *device,
730                             uint32_t psp_id,
731                             uint32_t index,
732                             struct dpl_rt_controller_dek_config *dek_config);
733
734     /**
735      * @brief Query all DEKs (Data Encryption Keys) of a PSP object.
736      *
737      * @param [in] device
738      * Device containing the requested PSP.
739      * @param [in] psp_id
740      * PSP object ID to query the DEKs of.
741      * @param [out] dek_configs
742      * Pointer to DEK configs array to store the result.
743      * @param [in] dek_configs_num
744      * Size of the dek_configs array.
745      * @note The user application is responsible to free the key.bytes array after using it.
746      * @return doca_error_t :
747      * DOCA_SUCCESS - in case of success.
748      * Error code - in case of failure. No need to free the key.bytes array.
749      */
750     DOCA_EXPERIMENTAL
751     doca_error_t dpl_rt_controller_psp_dek_query_all(struct dpl_rt_controller_device *device,
752                             uint32_t psp_id,
753                             struct dpl_rt_controller_dek_config *dek_configs,
754                             size_t dek_configs_num);
```

```
755
756  /**
757   * @brief Generate PSP keys.
758   *
759   * Generates an array of PSP keys for the given PSP version.
760   *
761   * @param [in] device
762   * Device to generate the PSP keys on.
763   * @param [in] psp_version
764   * PSP version to generate the keys for.
765   * @param [out] psp_keys
766   * Pointer to PSP keys array to store the result.
767   * @note The user application is responsible to free the key.bytes array after using it.
768   * @param [in] psp_keys_num
769   * Number of PSP keys to generate.
770   * @return doca_error_t :
771   * DOCA_SUCCESS - in case of success.
772   * Error code - in case of failure.
773   */
774  DOCA_EXPERIMENTAL
775  doca_error_t dpl_rt_controller_generate_psp_keys(struct dpl_rt_controller_device *device,
776                          enum dpl_rt_controller_psp_version psp_version,
777                          struct dpl_rt_controller_psp_key *psp_keys,
778                          size_t psp_keys_num);
```

7.8.9.5 Idle Timeout Entries

Efficient management of network device tables is essential for maintaining optimal performance and resource utilization. In dynamic environments, table entries may become stale or unused over time. To address this, *idle timeout* provides a proactive mechanism for table hygiene. This feature continuously monitors the activity of each table entry. If an entry is not accessed (or "hit") within a specified duration (the idle timeout period) it is marked as **stale**.

7.8.9.5.1 Limitations

Idle timeout support applies only to:

- Tables that explicitly support idle timeout
- Non-default entries

7.8.9.5.2 Usage

> See sections "DPL P4Runtime Controller Library" and "DPL Runtime Controller Library" for full details.

- Creating an entry with idle timeout: Use the same flow as for entries without idle timeout, but set the **idle_timeout_ns** field in the **dpl_rt_controller_add_entry_attr** struct, and pass it to the entry addition function.
- Modifying idle timeout: Use the **dpl_rt_controller_table_entry_modify_idle_timeout** API.
- Receiving stale entry notifications:
  - For gRPC-based (regular) tables – stale entry notifications are sent via the gRPC interface.
  - For High Update Rate (HUR) tables – use **dpl_rt_controller_idle_timeout_query** to retrieve stale entries.

7.8.9.5.3 Handling of Stale Entries

It is the controller implementer's responsibility to manage stale entries. System-specific constraints often require tailored solutions. Consider the following caveats:

- When using timeouts to delete entries, deletions should occur only within the timeout handling routine. Otherwise, the implementer must ensure protection against multiple deletions of the same entry.
- In systems where timeout scanning and entry deletion/insertion occur in different threads, pointers returned by the timeout scanning function may point to invalid or different entries by the time they are processed. The implementer must address this.
    - A basic solution is to use a mutex to block deletions while scanning and handling timeouts.
- The idle timeout query function does not cause process crashes, regardless of whether it is called from single-threaded or multi-threaded environments.

7.8.9.6 IPsec

The DPL IPsec offload allows for secure IPsec communication between two endpoints. It introduces HW offloading for increased performance and reduced CPU usage. The pipeline is expressed in P4 programs. The control plane programs SA slots at runtime using the DPL RT controller API.
Supported modes:
- Transport mode - encrypts the IP payload, preserves the original IP header.
- Tunnel mode - encrypts the entire original IP packet, wraps it in a new IP header.

The offloading will take care of the encryption and decryption, and use can send and read packets normally!

7.8.9.6.1 P4 Programming

See DOCA Target Architecture documentation for more details.

7.8.9.6.2 Usage

See DOCA Target Architecture documentation for more details and full flow.

7.8.9.6.2.1 IPsec Slots

Each IPsec object is an array of IPsec slots (SAs), each slot represents an IPsec connection, user must configure each slot to align with the connection user would like to establish. Configuration is done using the APIs described in this dcoument.

Note! Each direction (Rx/Tx) requires a connection of it's own, as IPsec connections are unidirectional.

7.8.9.6.2.2 IPsec object setup and configuration

IPsec objects configuration is done in 2 phases:

1. Setting up **NvIPsec** extern in the DPL program, compiler will create and **NvIPsec** extern for user to use. This will set default values for most info stated in the extern (ICV length, implicit IV and salt are configurable for later).

2. Using `dpl_rt_controller_ipsec_configure` to configure rest of the IPsec/SA parameter (SPI, mode, etc.). User must use the correct `NvIPsec` extern ID, also created by the compiler and can be found in the `.p4info.txt` file.

We use 2 phases because, as a rule of thumb, most IPsec connections (except maybe those used for testing) negotiate values needed for connection bring up. So user sets the ground for the IPsec connections in phase 1, and passes the information needed after obtained in phase 2.

7.8.9.6.2.3 Runtime Entries

As mentioned above, some information will be obtained by user only at runtime, so for actions that require the information (mainly encapsulation), user must insert runtime entries. In order to do so, the compiler will expose a header file named `<program_name>_dpl_p4rt.h` with functions for inserting runtime entries for each table that allows it.

User must use these functions in order to insert entries that will match the IPsec configurations of each slot.

7.8.9.6.2.4 IPsec Querying

In order to get information about the configuration of each slot, including HW state such as replat window state, here also, user must use the correct extern ID.

If call returns successfully, user must destroy the returned object using `dpl_rt_controller_ipsec_config_destroy` .

7.8.9.6.2.5 Removing IPsec Connections

In order to remove or delete IPsec connection, user must override it with a new configuration.

7.8.9.6.2.6 Usage Example

See IPsec sample for full transport mode example + detailed readme file.

7.8.9.7 DPL Runtime Controller Samples

Sample controller applications for a given DPL programs are installed by the `dpl-rt-controller-samples` package at the `/opt/dpl_rt_controller/samples/` directory.

7.8.9.7.1 Running the Sample

Building the sample applications requires meson utility. Make sure it is available on your system. To install meson on Ubuntu, run:

```
sudo apt update
sudo apt install meson
```

Follow these steps to build and run a sample application:

1. Connect to the DPU.

2. Make sure the DPL Runtime Service is up and running. See DPL Container Deployment for more details.
3. Navigate to the desired sample application directory (one of the folders under **/opt/dpl_rt_controller/samples/** on the DPU).
4. Compile the provided DPL program (e.g., **hello_world.p4**) found at the sample's directory. See Compiling DPL Applications for more details.
5. Copy the DPL program compilation output directory (**_out**) to a local path on the DPU.

   The compilation output directory contains the required C header files (**<program name>_dpl_shm.h and <program name>_dpl_shm_id.h**) for compiling the sample controller application.

6. Compile the C sample application:

```
cd /opt/dpl_rt_controller/samples/<sample_name>
meson /tmp/build -Dsample_programs_out=<PATH_TO_DPL_PROGRAM_COMPILATION_OUT_FOLDER>
ninja -C /tmp/build
```

   Replace the **<PATH_TO_DPL_PROGRAM_COMPILATION_OUT_FOLDER>** with the path to the local directory containing the DPL program compilation output folder (where the required C header files (**<program name>_dpl_shm.h and <program name>_dpl_shm_id.h**) are found).

   The binary **dpl_sample_<sample_name>** is created under **/tmp/build/**.

7. Run the sample application using a privileged user. For example:

```
sudo /tmp/build/dpl_sample_basic 1000 <path-to-p4info.txt> <path-to-dplconfig>
```

7.8.9.7.2 Samples

7.8.9.7.2.1 Basic

This sample demonstrates how to manage entries on a High Update Rate table as well as a regular table.

The sample logic includes:

1. Connecting to **dpl_rtd** over gRPC.
2. Loading a DPL program.
3. Connecting to **dpl_rtd** over SHM (Shared Memory).
4. Adding entry to a regular table.
5. Adding entry to a High Update Rate table.
6. Reading entries counter.
7. Deleting entries.
8. Displaying statistics.

9. Disconnecting and destroying all structures.

References:

- **/opt/dpl_rt_controller/samples/basic/basic_sample.cc**
- **/opt/dpl_rt_controller/samples/basic/basic_main.cc**
- **/opt/dpl_rt_controller/samples/basic/hello_world.p4**
- **/opt/dpl_rt_controller/samples/basic/meson_options.txt**
- **/opt/dpl_rt_controller/samples/basic/meson.build**

7.8.9.7.2.2 gRPC Only

This sample demonstrates how to use the generic **dpl_rt_controller** APIs to manage regular P4Runtime tables when High Update Rate tables are neither required nor defined in the DPL program.

The sample logic includes:

1. Connecting to **dpl_rtd** over gRPC.
2. Loading a DPL program.
3. Creating **dpl_rt_controller** device without SHM support.
4. Adding entry to a regular table.
5. Reading entries counter.
6. Deleting entries.
7. Disconnecting and destroying all structures.

References:

- **/opt/dpl_rt_controller/samples/grpc_only/grpc_only_sample.cc**
- **/opt/dpl_rt_controller/samples/grpc_only/grpc_only_main.cc**
- **/opt/dpl_rt_controller/samples/grpc_only/grpc_only.p4**
- **/opt/dpl_rt_controller/samples/grpc_only/meson_options.txt**
- **/opt/dpl_rt_controller/samples/grpc_only/meson.build**

7.8.9.7.2.3 Idle Timeout Sample

The sample logic includes:

1. Connecting to **dpl_rtd** over gRPC.
2. Loading a DPL program.
3. Connecting to **dpl_rtd** over SHM (SHared Memory).
4. Adding 2 entries to a regular table (gRPC) with very long timeout.
5. Adding 2 entries to a High Update Rate table (SHM) very long timeout.
6. Waiting for couple of seconds to see if entries expire (shouldn't).
7. Querying for stale entries (should find any).
8. Modifying idle timeout for one gRPC entry and one SHM entry.
9. Querying for stale entries (should find two).
10. Deleting all entries.
11. Displaying statistics.

12. Disconnecting and destroying all structures.

References:

- **/opt/dpl_rt_controller/samples/idle_timeout/idle_timeout_sample.cc**
- **/opt/dpl_rt_controller/samples/idle_timeout/idle_timeout_main.cc**
- **/opt/dpl_rt_controller/samples/idle_timeout/idle_timeout.p4**
- **/opt/dpl_rt_controller/samples/idle_timeout/meson_options.txt**
- **/opt/dpl_rt_controller/samples/idle_timeout/meson.build**

7.8.9.7.2.4 TCP state (Connection Tracking) Sample

This sample demonstrates how to manage a TCP connection state, including adding entries to track connections and retrieving the connection's acknowledgement (ACK) number and sequence (SEQ) number of the last packet that hit the TCP state object.

The sample logic includes:

1. Connecting to **dpl_rtd** over gRPC.
2. Loading a DPL program.
3. Connecting to **dpl_rtd** over SHM (SHared Memory).
4. Setting up DPDK for packet processing and table entry management (addition/deletion).
5. Printing statistics and current TCP state object info.
6. Disconnecting and destroying all structures.

References:

- **/opt/dpl_rt_controller/samples/tcp_state/tcp_state.p4**
- **/opt/dpl_rt_controller/samples/tcp_state/tcp_state_sample.cc**
- **/opt/dpl_rt_controller/samples/tcp_state/tcp_state_main.cc**
- **/opt/dpl_rt_controller/samples/tcp_state/meson_options.txt**
- **/opt/dpl_rt_controller/samples/tcp_state/meson.build**

7.8.9.7.2.5 PSP Sample

This sample demonstrates a basic client-server PSP (PSP Security Protocol) implementation, featuring key exchange and encrypted traffic forwarding.

> Since the PSP key exchange is outside the scope of the DOCA Pipeline Language (DPL) itself, this sample implements a gRPC-based mechanism.

The application uses **dpl_rt_controller** to generate keys and program the DEK tables. Combined with the provided **psp.p4** program, it enables a complete end-to-end encrypted workflow.

The sample runs on two connected NICs. Both run the DPL Runtime Service and share the **psp.p4** program, acting as Client (Sender) and Server (Receiver).

> The application exchanges keys using an unencrypted gRPC channel.

In its current implementation, the PSP Sample application enables one-way communication from the client (sender) to the server (receiver).

Sample Logic

The following sequence describes the interaction between the Client and Server components.

- Server side:
  a. Connects to **dpl_rtd** via gRPC.
  b. Loads the DPL program (**psp.p4**).
  c. Listens for key-exchange requests.
  d. On Request:
    i. Generates PSP keys and assigns a SPI (Security Parameter Index).
    ii. Inserts an entry into the **decap_table** using the SPI.
    iii. Sends the generated keys back to the Client.
  e. Ready to decrypt incoming traffic.
- Client side:
  a. Connects to **dpl_rtd** via gRPC.
  b. Loads the DPL program (**psp.p4**).
  c. Requests key exchange from the Server (one key per entry).
  d. Programs the DEK using the received keys.
  e. Inserts an entry into the **encap_table** using the corresponding SPI.
  f. Ready to encrypt and send traffic.

References

- **/opt/dpl_rt_controller/samples/psp/psp.p4**
- **/opt/dpl_rt_controller/samples/psp/psp_main.cc**
- **/opt/dpl_rt_controller/samples/psp/psp_sample.cc**
- **/opt/dpl_rt_controller/samples/psp/psp_gateway_service.cc**
- **/opt/dpl_rt_controller/samples/psp/psp_gateway_service.hpp**
- **/opt/dpl_rt_controller/samples/psp/meson_options.txt**
- **/opt/dpl_rt_controller/samples/psp/meson.build**

Usage

```
./dpl_sample_psp <device_id> <p4info_path> <program_path> <mode (client|server)> <address (ip:port|port)>
<number_of_entries (optional)>

Positional Arguments:
    device_id: DPL device ID
    p4info_path: Path to P4Info file
    program_path: Path to DPL program file
    mode: Operation mode (client or server)
    address: Remote PSP gateway address in format <ip:port> for client mode (required) or <port> for server mode
(optional). Default is 50051 for server mode.
    number_of_entries: Number of entries to insert into the encap_table and decap_table. For client mode only.
Default is 5.
```

Workflow

1. Run the application in server mode on the first NIC.

```
./dpl_sample_psp 1000 <path-to-p4info.txt> <path-to-dplconfig> server
```

2. Run the application in client mode on the second NIC.

```
./dpl_sample_psp 1000 <path-to-p4info.txt> <path-to-dplconfig> client "<server_ip>:50051"
```

3. Send test traffic: On the client side, inject packets to a virtual function (VF). The sample configures default entries for the range **1.1.1.1** to **1.1.1.X**.
    - Match keys:
        - **IPv4_dst_address**: "2.2.2.2"
        - **IPv4_src_address**: "1.1.1.1" $\rightarrow$ "1.1.1.<count>"
    - Scapy command:

```
sendp(Ether(src="00:11:11:11:11:11", dst="00:22:22:22:22:22") /
      IP(src="1.1.1.1", dst="2.2.2.2") /
      Raw(load=b'\xCA\xFE\x12\x34'), iface='eth8')
```

    - Run **tcpdump** on the Server's VF interface. You should see the decrypted packet **0xCAFE1234**.

7.8.9.7.2.6 Gateway SHM Sample

This sample demonstrates how to control a VXLAN gateway program using HUR tables via SHM.

This sample requires the **json-c** package. To install it on Ubuntu, run:

```
sudo apt install libjson-c-dev
```

Sample Logic

The controller interacts with **dpl_rtd** through two distinct channels: gRPC for configuration and SHM for high-speed table updates.

1. Connect to **dpl_rtd** via gRPC.
2. Load the DPL program.
3. Connect to **dpl_rtd** via SHM.
4. Parse **gateway.entries.json** and add entries to the High Update Rate tables via SHM.
5. Print counters and entry details upon user key press.
6. Disconnect and clean up resources.

References

- **/opt/dpl_rt_controller/samples/gateway_shm/gateway_shm.p4**
- **/opt/dpl_rt_controller/samples/gateway_shm/gateway.entries.json**
- **/opt/dpl_rt_controller/samples/gateway_shm/gateway_shm_sample.cc**
- **/opt/dpl_rt_controller/samples/gateway_shm/gateway_shm_main.cc**
- **/opt/dpl_rt_controller/samples/gateway_shm/meson_options.txt**
- **/opt/dpl_rt_controller/samples/gateway_shm/meson.build**

7.8.9.7.2.7 Modify Entry Sample (Atomic Updates)

This sample demonstrates the dynamic modification of table entry action parameters in a DPL-defined data plane.

It utilizes the SDK API to update specific action data (e.g., Source Port, ECN, DSCP) atomically. Unlike standard operations that require deleting and re-inserting the flow, this method modifies the existing entry in place, ensuring efficient control plane updates and minimizing packet processing disruption.

Sample Logic

The sample logic includes:

1. Connect to **dpl_rtd** via gRPC and SHM.
2. Load the DPL program.
3. Add initial entries via both gRPC and SHM.
4. Test 1: Send packets to hit these entries (verify via **tcpdump**).
5. Update specific parameters of the existing entries via gRPC and SHM.
6. Test 2: Send the same packets again to observe the updated behavior (e.g., changed DSCP or Port).
7. Clean up and exit.

References

- **/opt/dpl_rt_controller/samples/modify_entry/atomic_action_param_update.p4**
- **/opt/dpl_rt_controller/samples/modify_entry/modify_entry_sample.cc**
- **/opt/dpl_rt_controller/samples/modify_entry/modify_entry_example_outputs.txt**
- **/opt/dpl_rt_controller/samples/modify_entry/modify_entry_main.cc**
- **/opt/dpl_rt_controller/samples/modify_entry/meson_options.txt**
- **/opt/dpl_rt_controller/samples/modify_entry/meson.build**

7.8.9.7.2.8 IPsec Sample

This sample demonstrates hardware-offloaded AES-128-GCM IPsec transport mode using strongSwan for IKE key negotiation and
DPL RT controller API for programming the pipeline.

The sample operates on both sides of the IPsec operation:
- **local** (DPU side - loads the P4 program, programs IPsec SAs, and inserts P4 table entries.)
- **remote** (peer side - configures strongSwan and logs XFRM SA events).

Sample Logic

Local Mode (DPU side)

1. Connects to `dpl_rtd` over gRPC.
2. Loads the DPL program (`ipsec.p4`).
3. Starts a Netlink XFRM listener (before IKE negotiation, to avoid missing SA events).

4. Signals readiness via FIFO, then waits for strongSwan to install both outbound (TX) and inbound (RX) XFRM SAs.
5. Extracts AES-128-GCM key + salt from the `XFRMA_ALG_AEAD` Netlink attribute.
6. Programs the DPL device using `dpl_rt_controller_ipsec_configure`:
   - **TX SA** → slot 1.
   - **RX SA** → slot 2.
7. Inserts encap_table P4 entry with the negotiated SPI.
8. Pauses for user to run test traffic.
9. Queries SA slots and prints results.

Remote Mode (peer side)

1. Starts a Netlink XFRM listener.
2. Logs negotiated SA SPIs once both outbound and inbound SAs are installed.
3. No DPL programming — strongSwan handles the IPsec stack on the peer side.

References

- **ipsec/README.md**
- **ipsec/ipsec.p4**
- **ipsec/ipsec_sample.cc**
- **ipsec/ipsec_main.cc**
- **ipsec/meson_options.txt**
- **ipsec/meson.build**
- **ipsec/run_ipsec_sample.sh**
- **ipsec/send_rx_traffic.py**

7.9 DOCA PCC Counter Tool

This document provides instruction on the usage of the PCC Counter tool.

> This tool is supported ONLY on NVIDIA® ConnectX®-7 and cannot run in parallel with DOCA Nguage or DOCA Telemetry.

7.9.1 Introduction

The PCC Counter tool is used to print PCC-related hardware counters. The output counters help debug the PCC user algorithm embedded in the DOCA PCC application.

7.9.2 Prerequisites

DOCA 2.2.0 and higher.

7.9.3 Description

If NVIDIA® BlueField®-3 is operating in DPU mode, the script must be executed on the Arm side. If BlueField-3 is operating in NIC mode, the script must be executed on the host side.

> Refer to BlueField Modes of Operation for more information on the DPU's modes of operation.

The following performance counters are supported for PCC:

- **MAD_RTT_PERF_CONT_REQ** – the number of RTT requests received in total
- **MAD_RTT_PERF_CONT_RES** – the number of RTT responses received in total
- **SX_EVENT_WRED_DROP** – the number of TX events dropped due to the CC event queue being full
- **SX_RTT_EVENT_WRED_DROP** – the number of "TX event with RTT request sent indication" dropped due to the CC event queue being full
- **ACK_EVENT_WRED_DROP** – the number of Ack events dropped due to the CC event queue being full
- **NACK_EVENT_WRED_DROP** – the number of Nack events dropped due to the CC event queue being full
- **CNP_EVENT_WRED_DROP** – the number of CNP events dropped due to the CC event queue being full
- **RTT_EVENT_WRED_DROP** – the number of RTT events dropped due to the CC event queue being full
- **HANDLED_SXW_EVENTS** – the number of handled CC events related to SXW
- **HANDLED_RXT_EVENTS** – the number of handled CC events related to RXT
- **DROP_RTT_PORT0_REQ** – the number of RTT requests dropped in total from port 0
- **DROP_RTT_PORT1_REQ** – the number of RTT requests dropped in total from port 1
- **DROP_RTT_PORT0_RES** – the number of RTT responses dropped in total from port 0
- **DROP_RTT_PORT1_RES** – the number of RTT responses dropped in total from port 1
- **RTT_GEN_PORT0_REQ** – the number of RTT requests sent in total from port 0
- **RTT_GEN_PORT1_REQ** – the number of RTT requests sent in total from port 1
- **RTT_GEN_PORT0_RES** – the number of RTT responses sent in total from port 0
- **RTT_GEN_PORT1_RES** – the number of RTT responses sent in total from port 1
- **PCC_CNP_COUNT** – the number of CNP received in total, regardless of whether it is handled or ignored

7.9.4 Execution

To use the PCC Counter:

1. Initialize all supported hardware counters. Run:

```
sudo ./pcc_counters.sh set /dev/mst/mt41692_pciconf0
```

Counters are zeroed after each **set** command.

2. Query all supported hardware counters. Run:

```
sudo ./pcc_counters.sh query /dev/mst/mt41692_pciconf0
```

The output counters are counted from the time the **set** command is executed to the time when the **query** command is issued.

Example output:

```
sudo ./pcc_counters.sh query /dev/mst/mt41692_pciconf0
-----------------PCC Counters-----------------
Counter: MAD_RTT_PERF_CONT_REQ  Value: 000000000028b85b
Counter: MAD_RTT_PERF_CONT_RES  Value: 000000000028b85a
Counter: SX_EVENT_WRED_DROP     Value: 0000000000000000
Counter: SX_RTT_EVENT_WRED_DROP Value: 0000000000000000
Counter: ACK_EVENT_WRED_DROP    Value: 0000000000ccdf4f
Counter: NACK_EVENT_WRED_DROP   Value: 0000000000000000
Counter: CNP_EVENT_WRED_DROP    Value: 0000000000000000
Counter: RTT_EVENT_WRED_DROP    Value: 0000000000000000
Counter: HANDLED_SXW_EVENTS     Value: 000000000932543a
Counter: HANDLED_RXT_EVENTS     Value: 000000000028b85c
Counter: DROP_RTT_PORT0_REQ     Value: 0000000000000000
Counter: DROP_RTT_PORT1_REQ     Value: 0000000000000000
Counter: DROP_RTT_PORT0_RES     Value: 0000000000000000
Counter: DROP_RTT_PORT1_RES     Value: 0000000000000000
Counter: RTT_GEN_PORT0_REQ      Value: 0000000000000000
Counter: RTT_GEN_PORT1_REQ      Value: 000000000028b85c
Counter: RTT_GEN_PORT0_RES      Value: 0000000000000000
Counter: RTT_GEN_PORT1_RES      Value: 000000000028b85d
Counter: PCC_CNP_COUNT          Value: 0000000000000000
```

7.10 DOCA Socket Relay

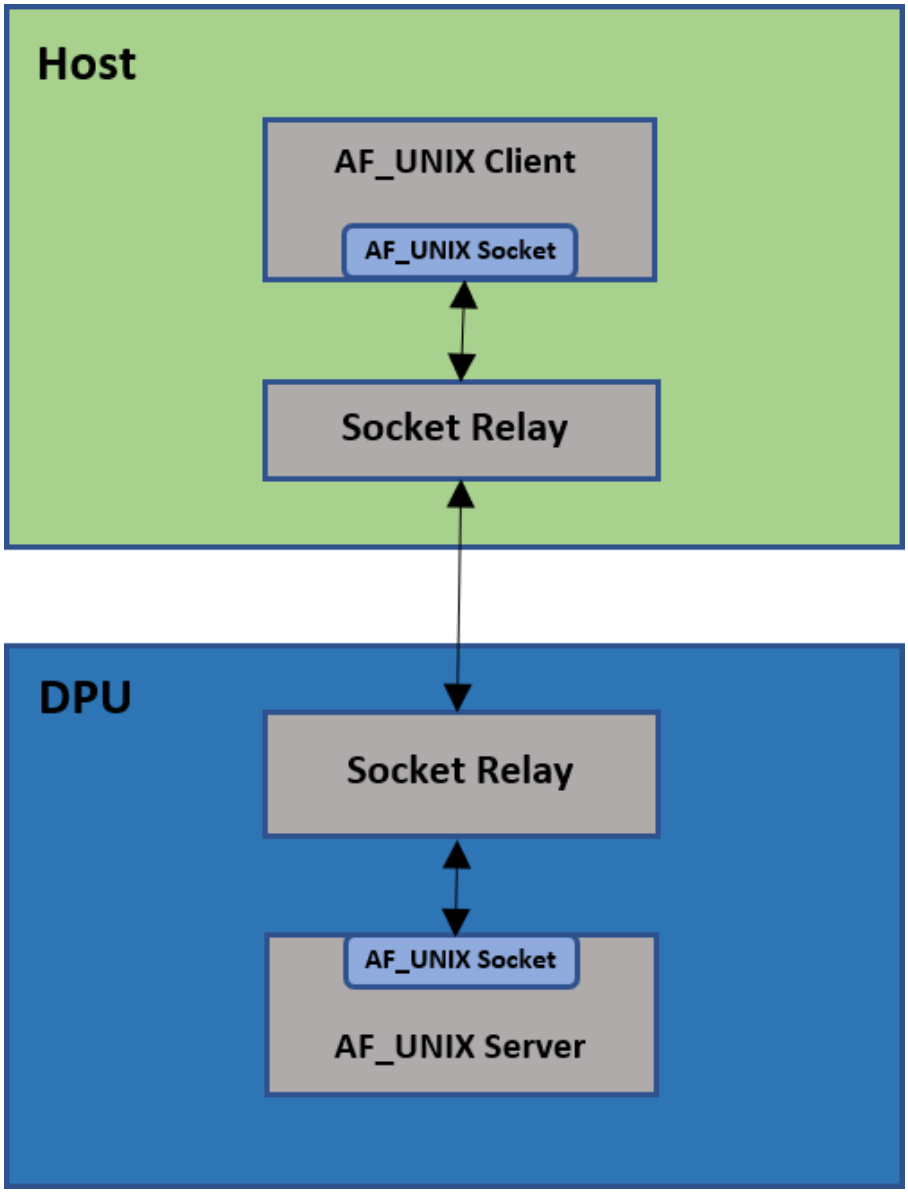
This document describes DOCA Socket Relay architecture, usage, etc.

7.10.1 Introduction

DOCA Socket Relay allows Unix Domain Socket (AF_UNIX family) server applications to be offloaded to the DPU while communication between the two sides is proxied by DOCA Comch.

Socket relay only supports SOCK_STREAM communication with a limit of 512 AF_UNIX application clients.

The tool is coupled to the client AF_UNIX server application. That is, a socket relay instance should be initiated per AF_UNIX server application.



Socket relay is transparent to the application except for the following TCP flows:

- Connection termination must be done by the host side application only
- Once a FIN packet (shutdown system call has been made) is sent by the host side application, data cannot be transferred between the DPU and the host, and the connection must be closed.

The following details the communication flow between the client and server:

- The AF_UNIX client application connects to the socket relay AF_UNIX server in the same way as in the original flow
- The AF_UNIX client application sends SOCK_STREAM packets
- The socket relay (host) AF_UNIX server receives the client application packets, and the Comm Channel client sends them on the channel

- The socket relay (DPU) Comm Channel server receives the client application packets and the AF_UNIX client sends them to the user's AF_UNIX server application

7.10.2 Prerequisites

Windows 10 build 17063 is the minimal Windows version to run DOCA Socket Relay on a Windows host.

7.10.3 Dependencies

NVIDIA® BlueField®-2 firmware version 24.35.1012 or higher.

7.10.4 Execution

To execute DOCA Socket Relay:

```
Usage: doca_socket_relay [DOCA Flags] [Program Flags]

DOCA Flags:
  -h, --help                Print a help synopsis
  -v, --version             Print program version information
  -l, --log-level           Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level           Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --log-filter              Filter logs from specific modules, separated by comma
  -j, --json <path>         Parse command line flags from an input json file

Program Flags:
  -s, --socket              Unix domain socket path, host side will bind to and DPU connect to
  -n, --cc-name             Comm Channel service name
  -p, --pci-addr            DOCA Comm Channel device PCI address
  -r, --rep-pci             DOCA Comm Channel device representor PCI address (needed only on DPU)
```

For example (DPU side):

```
doca_socket_relay -s /tmp/sr_server.socket -n cc_channel -p 03:00.0 -r b1:00.0
```

7.10.5 Arg Parser DOCA Flags

Refer to the DOCA Arg Parser for more information.

7.10.5.1 General Flags

Short Flag	Long Flag	Description
`-h`	`--help`	Prints a help synopsis and exits
`-v`	`--version`	Prints program version information and exits

Short Flag	Long Flag	Description
`-l`	`--log-level`	Sets the numeric log level for the application: • `10` – DISABLE • `20` – CRITICAL • `30` – ERROR • `40` – WARNING • `50` – INFO • `60` – DEBUG • `70` – TRACE (requires compilation with TRACE support)
N/A	`--sdk-log-level`	Sets the SDK numeric log level using the same 10-70 scale as above
N/A	`--log-filter`	Filters logs from specific modules (comma-separated list)
`-j`	`--json`	Parses command-line flags from a specified input JSON file

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

7.10.5.2 Program Flags

Short Flag	Long Flag	Description
`s`	`socket`	AF_UNIX (`SOCK_STREAM`) path. On the host, this is the path of the socket relay AF_UNIX server for the client's application to connect to. On the DPU, this is the path of the client AF_UNIX server application. ⚠ This flag is mandatory.
`n`	`cc-name`	Comm Channel service name ⚠ This flag is mandatory.
`p`	`pci-addr`	DOCA Comm Channel device PCIe address ⚠ This flag is mandatory.
`r`	`rep-pci`	DOCA Comm Channel device representor PCIe address ⚠ This flag is available and mandatory only on the DPU.

7.11 DOCA Telemetry Utils

This page provides instructions on the usage of the **doca_telemetry_utils** tool.

> Supported hardware is NVIDIA® BlueField®-3, NVIDIA® ConnectX®-7, and above.

7.11.1 Introduction

The **doca_telemetry_utils** tool can be used to display all the available counters, and to generate counter IDs which can be used in other DOCA Tools. It also can get the counter type and settings from a given counter ID, and to inform whether this counter is available on the required hardware.

This tool simplifies counter management, making it easier to identify, configure, and verify counter support for specific devices.

> **doca_telemetry_utils** relies on the **fwctl** driver and, therefore, probing hardware support will fail if run simultaneously with other tools or services that also utilize this driver.

7.11.2 Prerequisites

To utilize DOCA Telemetry Diag, your system must meet the following baseline requirements:

- Firmware: Version **>=28/32/40.43.1000** is required for ConnectX-7, BlueField-3, and ConnectX-8 devices.
- Driver: The **fwctl** driver must be fully installed and actively loaded on the system.

7.11.2.1 Verifying the fwctl Driver

To verify that the **fwctl** driver is successfully loaded, check the device directories:

```
$ ls /sys/class/fwctl/
$ ls /dev/fwctl
```

> The expected output for a standard 2-port device is **fwctl0 fwctl1**.

7.11.2.2 Manually Loading the Driver

If the directories **/sys/class/fwctl** or **/dev/fwctl** do not exist or are empty, the module may be installed but inactive.

Check for the module's presence:

```
$ grep fwctl -R /lib/modules/$(uname -r)/
```

If the output confirms the presence of **fwctl.ko** and **mlx5_fwctl.ko**, manually load the module and verify its status:

```
$ sudo modprobe mlx5_fwctl
$ lsmod | grep fwctl
```

7.11.2.3 Reinstalling the DOCA Host Package

If you cannot locate the installed **fwctl** module while manually loading the driver, or if the **modprobe** command fails to load it successfully, you must reinstall the DOCA Host package.

1. Download the package (DOCA 3.3.0 example):

   ```
   $ wget https://www.mellanox.com/downloads/DOCA/DOCA_v3.3.0/host/doca-host_3.3.0-088000-26.01-ubuntu2204_amd64.deb
   ```

2. Purge existing DOCA and OFED modules:

   ```
   $ sudo for f in $( dpkg --list | grep doca | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ;
   done
   $ sudo for f in $( dpkg --list | grep mlnx | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ;
   done
   $ sudo for f in $( dpkg --list | grep dpdk | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ;
   done
   $ sudo for f in $( dpkg --list | grep ofed | awk '{print $2}' ); do echo $f ; apt remove --purge $f -y ;
   done
   $ sudo /usr/sbin/ofed_uninstall.sh --force
   $ sudo apt-get autoremove
   ```

3. Install the new package and restart services:

   ```
   $ sudo dpkg -i doca-host_3.3.0-088000-26.01-ubuntu2204_amd64.deb
   $ sudo apt-get update
   $ sudo apt-get -y install doca-all
   $ sudo /etc/init.d/openibd restart
   ```

   Once the reinstallation is complete, confirm the module is successfully loaded according to section "Verifying the fwctl Driver".

7.11.2.4 Installing DOCA Telemetry Utils

To install **doca_telemetry_utils**:

- On deb-based distros, run:

  ```
  sudo apt-get install doca-telemetry-utils
  ```

- On RPM-based distros, run:

  ```
  sudo dnf install doca-telemetry-utils
  ```

7.11.3 Description

The **doca_telemetry_utils** tool can be used with either a counter name or a counter data ID.

7.11.3.1 Usage with Counter Name

When providing a counter name, **doca_telemetry_utils** displays the associated data ID and additional details:

The following is an example of running with the counter named **global_icmc_hit**:

```
$ doca_telemetry_utils global_icmc_hit
Data ID: 0x1180000200000000
Name: global_icmc_hit
Unit: ICMC
```

For counters requiring arguments, running the tool with only the counter name displays the options needed.

Running with **port_rx_bytes**:

```
[fill_data_id] Per-port counter 0x10200001 (port_rx_bytes) needs exactly 1 argument (local_port), 0
given.
```

In this case, you must provide the required argument(s) and re-run the command.

Example of specifying the argument **local_port**:

```
$ doca_telemetry_utils port_rx_bytes 0
Data ID: 0x1020000100000000
Name: port_rx_bytes
Unit: RX_PORT
local_port: 0
```

7.11.3.2 Usage with Data ID

When providing a data ID, **doca_telemetry_utils** displays the counter name and other details:

Example of running with data ID **0x1180000200000000**:

```
$ doca_telemetry_utils 0x1180000200000000
Data ID: 0x1180000200000000
Name: global_icmc_hit
Unit: ICMC
```

7.11.3.3 Checking Counter Support on a Device

To check whether a specific counter is supported by a particular device using the device's PCIe address.

To check if the **global_icmc_hit** counter is supported on device **08:00.0**:

```
$ doca_telemetry_utils 08:00.0 0x1180000200000000
Data ID: 0x1180000200000000
Name: global_icmc_hit
Unit: ICMC
Data ID 0x1180000200000000 is supported on device 08:00.0
```

7.11.4 Execution

To run **doca_telemetry_utils**:

```
$ doca_telemetry_utils -h
Usage:
Name to Data ID:
     doca_telemetry_utils [<device PCI>] <name> [relevant properties]
    • To get the options for 'relevant properties' run with 'name' alone.
Data ID to name:
     doca_telemetry_utils [<device PCI>] <DATA_ID>

• If the optional argument <device PCI> is provided, this device will be tested for support of this counter.
• Run with option 'get-counters' to get all the available names.

-h, --help:  Show this help message.
```

Examples:

- Show all available counters:

  ```
  doca_telemetry_utils get-counters
  ```

- Name to data ID with relevant options:

  ```
  doca_telemetry_utils port_rx_bytes 0
  ```

- Name to Data ID with device PCIe:

  ```
  doca_telemetry_utils 08:00.0 port_rx_bytes 0
  ```

- Data ID to name:

  ```
  doca_telemetry_utils 0x1020000100000000
  ```

- Data ID to Name with device PCIe:

  ```
  doca_telemetry_utils 08:00.0 0x1020000100000000
  ```

7.12 DOCA doca-hugepages Tool

This guide provides an overview of the **doca-hugepages** tool, including its purpose, usage instructions, command syntax, and integration guidelines for managing hugepage allocations in NVIDIA DOCA environments.

7.12.1 Introduction

The **doca-hugepages** tool manages hugepages configurations for various applications. It enables users to configure, reload, remove, and display hugepages settings efficiently. The tool is located at **/usr/sbin/doca-hugepages**.

All **doca-hugepages** commands begin with **doca-hugepages** and fall into the following categories:

- **config** – Adds or updates an application's configuration in the database. This command does not apply changes immediately; use the **reload** command to allocate hugepages based on the updated configuration.
- **reload** – Reloads the hugepages configuration for all applications according to the current database.
- **remove** – Removes an application's configuration from the database.
- **show** – Displays the current hugepages configuration for all applications.
- **original-config** – Manages and displays the original system configuration saved by the tool.

7.12.2 Execution

7.12.2.1 config

Adds or updates an application's configuration in the database.

```
doca-hugepages config [-h] --app APP --size SIZE --num NUM [--force]

doca-hugepages config Flags:
    --force (optional)      Forces updating the configuration without prompting.
    --app                   Name of the application.
    --size                  Hugepage size in kB.
    --num                   Number of hugepages to allocate.
```

For example:

```
doca-hugepages config --app myApp --size 2048 --num 10
```

7.12.2.2 reload

Reloads the hugepages configuration for all applications.

```
doca-hugepages reload
```

7.12.2.3 remove

Removes an application's configuration from the database.

```
doca-hugepages remove <app>

doca-hugepages remove Flags:
    --all:      Removes all configurations for the app without prompting.
```

7.12.2.4 show

Displays the current hugepages configuration for all applications.

```
doca-hugepages show
```

7.12.2.5 original-config

Manages the original system configuration captured by the tool on its first run. This snapshot serves as a reference for restoring or comparing configurations.

```
doca-hugepages original-config {update,show}

doca-hugepages original-config Flags:
    show:       Display the original system configuration.
    update:     Override the original system configuration.
```

7.12.2.6 Important Notes

- Always run **doca-hugepages reload** after updating configurations to apply changes.
- Use **--force** with caution to avoid unintended updates.
- Periodically use **show** to verify the active configuration.

7.12.3 Incompatibility with GRUB-Based Hugepages Management

The **doca-hugepages** tool is not supported on systems where hugepages are configured through GRUB (e.g., via **/etc/default/grub**). These two methods are mutually exclusive. Attempting to use both may result in unpredictable behavior or configuration conflicts.

To use **doca-hugepages**, ensure that all hugepages-related GRUB parameters are removed. Conversely, when using GRUB-based configuration, do not invoke **doca-hugepages**.

7.12.4 Integration with Packages

Custom per-application hugepages configurations can be integrated into packages by placing a configuration file in the **/etc/mellanox/hugepages.d/** directory.

Example:

```
/etc/mellanox/hugepages.d/APP.json

{
  "2048": {
    "num": 1024,
    "is_active": "inactive"
  }
}
```

In this example:

- **"2048"** – Hugepage size in kilobytes
- **"num"** – Number of hugepages to allocate
- **"is_active"** – Marks the configuration as **"active"** or **"inactive"**

NVIDIA recommends setting **"is_active": "inactive"** and using the **doca-hugepages** tool to activate the configuration during driver load or system boot. This workflow is supported only on Ubuntu 22.04 systems running on BlueField platforms.

8 DOCA Services

This section provides an overview of the various services provided by DOCA.

8.1 Introduction

DOCA services are DOCA-based products, wrapped in a container for fast and easy deployment on top of the NVIDIA BlueField DPU. DOCA services leverage DPU capabilities to offer telemetry, time synchronization, networking solutions, and more.

Services containers can be found in the NGC catalog, labeled "DOCA" and "DPU", as well as the built-in NVIDIA platform option ("DOCA") in the container catalog.

The following services are not available in the NGC catalog:

- DOCA Management Service
- NVIDIA OpenvSwitch Acceleration (OVS in DOCA)

For information on the deployment of the services, refer to the DOCA Container Deployment Guide.

8.1.1 DOCA Services Summary and Quality Levels

DOCA Service	Description	Quality Level
DOCA Platform Framework (DPF)	Provides fleet-scale provisioning and lifecycle management for BlueField DPUs and orchestrates DOCA services and compatible third-party DPU services across the fleet.	GA
DOCA Argus	Provides agentless runtime threat detection from BlueField's isolated trust domain.	Beta
DOCA Blueman	Provides a web dashboard for BlueField inventory, health, and telemetry.	GA
DOCA Firefly	Synchronizes BlueField and host clocks using Precision Time Protocol (PTP).	GA
DOCA Flow Inspector	Analyzes mirrored network traffic and sends real-time packet telemetry to DOCA Telemetry Service.	GA
DOCA HBN	Provides hardware-accelerated BGP/EVPN routing, ECMP, L4 firewall, and NAT on BlueField.	GA
DMS	Provides centralized configuration and management for BlueField and NVIDIA networking devices.	Alpha
DPL Service	Provides a P4-derived language and tools for building custom hardware-accelerated packet-processing pipelines.	Beta
SNAP Virtio-FS	Presents networked file storage to hosts as accelerated Virtio-fs devices on BlueField-3.	Beta

DOCA Service	Description	Quality Level
SNAP-4	Presents networked block storage to hosts as NVMe or virtio-blk devices on BlueField-3.	GA
SNAP-3	Presents networked block storage to hosts as NVMe or virtio-blk devices on BlueField-2.	GA
DOCA Telemetry	Collects and exports real-time telemetry from BlueField DPUs, hosts, and workloads.	GA
DOCA UROM	Offloads HPC and AI parallel-computing operations from hosts to BlueField.	GA
DOCA Virtio-net	Provides hardware-accelerated host and VM networking through BlueField-emulated virtio-net devices.	GA
OVS-DOCA	Offloads Open vSwitch packet processing to NVIDIA hardware while retaining the standard OVS control plane.	GA
DOCA XPlane	Manages failover between local and remote OVS-DOCA planes in Spectrum-X deployments.	GA

8.2 Development Lifecycle

DOCA-based containers consist of two main categories:

- DOCA Base Images: containerized DOCA environments for both runtime and development. Used either by developers for their development environment or in the process of containerizing a DOCA-based solution.
- DOCA Services: containerized DOCA-based products

8.2.1 Development

Before containerizing a product, users must first design and develop it using the same process for a bare-metal deployment on the BlueField DPU.

This process consists of the following steps:

1. Identify the requirements for the DOCA-based solution.
2. Review the feature set offered by the DOCA SDK libraries, as shown in their respective programming guides.
3. Start the development process by following the best practices detailed in the Developer Guide.
4. Test the developed solution.

After development and testing, the product can be containerized.

8.2.2 Containerization

For this process, NVIDIA recommends using DOCA's base images, available on DOCA's NGC page.

There are three image varieties:

- **base-rt** – includes the DOCA runtime, using the most basic runtime environment required by DOCA's SDK
- **full-rt** – builds on the previous image and includes the full list of runtime packages, which are all user-mode components that can be found under the **doca-runtime** package
- **devel** – builds on the previous image and adds headers and development tools for developing and debugging DOCA applications. This image is particularly useful for multi-stage builds.

All images are preconfigured to use to the DOCA repository of the matching DOCA version. This means that installing an additional DOCA package as part of a Docker file or within the development container can be done using the following commands:

```
apt update
apt install <package name>,
```

For DOCA and CUDA environments, the images follow a similar pattern, and are combined with CUDA's images:

- **base-rt** (DOCA) + **base** (CUDA)
- **full-rt** (DOCA) + **runtime** (CUDA)
- **devel** (DOCA) + **devel** (CUDA)

After development and testing, users can start profiling the containerized product in preparation for a production-grade deployment.

DOCA provides base images for both the DPU and the host. For host-related DOCA base images, refer to the image tag suffixed with "-host".

8.2.3 Profiling

As mentioned in the DOCA Container Deployment Guide, the current deployment model of containers on top of the DPU is based on kubelet-standalone. This Kubernetes-based deployment makes use of YAML files to describe the resources required by the pod such as:

- CPU
- RAM
- Huge pages

NVIDIA recommends profiling your product to estimate the resources it requires (under regular deployments, as well as under stress testing) so that the YAML file includes an accurate "resources" section. This allows administrators to better understand the requirements for deploying your service, and also enables the Kubernetes infrastructure to ensure that the service is not malfunctioning after deployment.

After the profiling is complete, the containerized DOCA-based product is ready for the final testing and deployment in production environments.

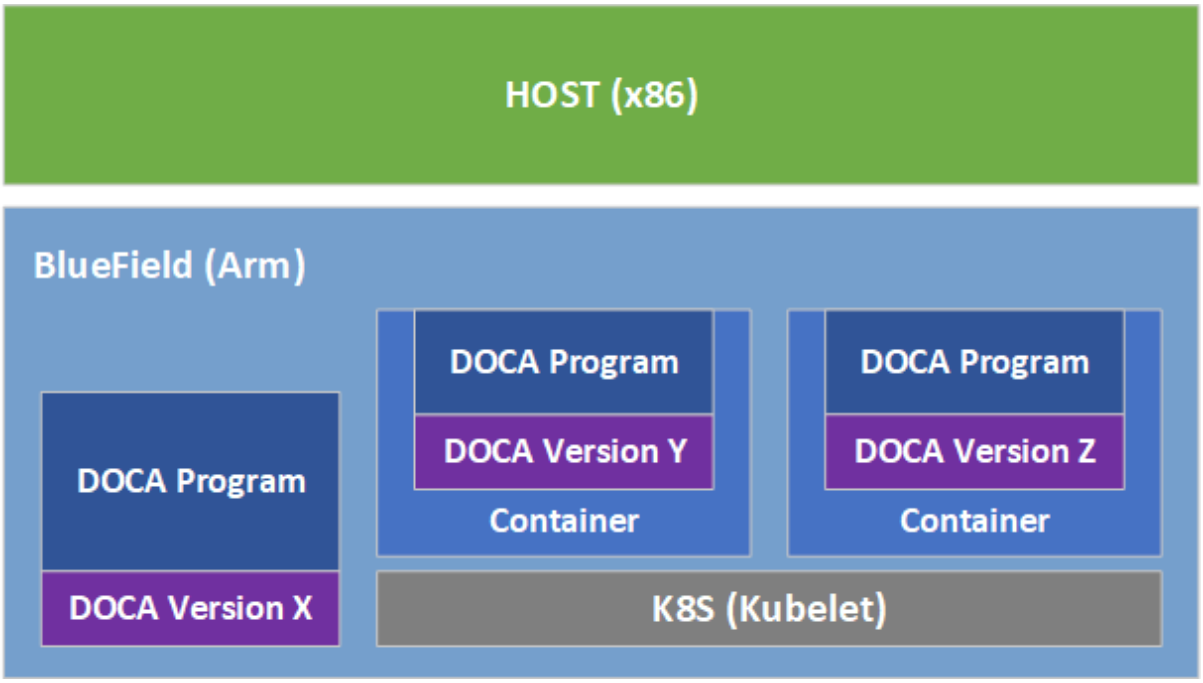
8.3 DOCA Container Deployment Guide

This guide provides an overview and deployment configuration of DOCA containers for NVIDIA® BlueField® DPU.

8.3.1 1. Introduction

DOCA containers allow for easy deployment of ready-made DOCA environments to the DPU, whether it is a DOCA service bundled inside a container and ready to be deployed, or a development environment already containing the desired DOCA version.

Containerized environments enable the users to decouple DOCA programs from the underlying BlueField software. Each container is pre-built with all needed libraries and configurations to match the specific DOCA version of the program at hand. One only needs to pick the desired version of the service and pull the ready-made container of that version from NVIDIA's container catalog.



The different DOCA containers are listed on NGC, NVIDIA's container catalog, and can be found under both the "DOCA" and "DPU" labels.

8.3.2 2. Prerequisites

- Refer to the DOCA Installation Guide for Linux for details on how to install BlueField related software
- BlueField image version required is 3.9.0 and higher

Container deployment based on **standalone** Kubelet, as presented in this guide, is currently in **alpha version** and is subject to change in future releases.

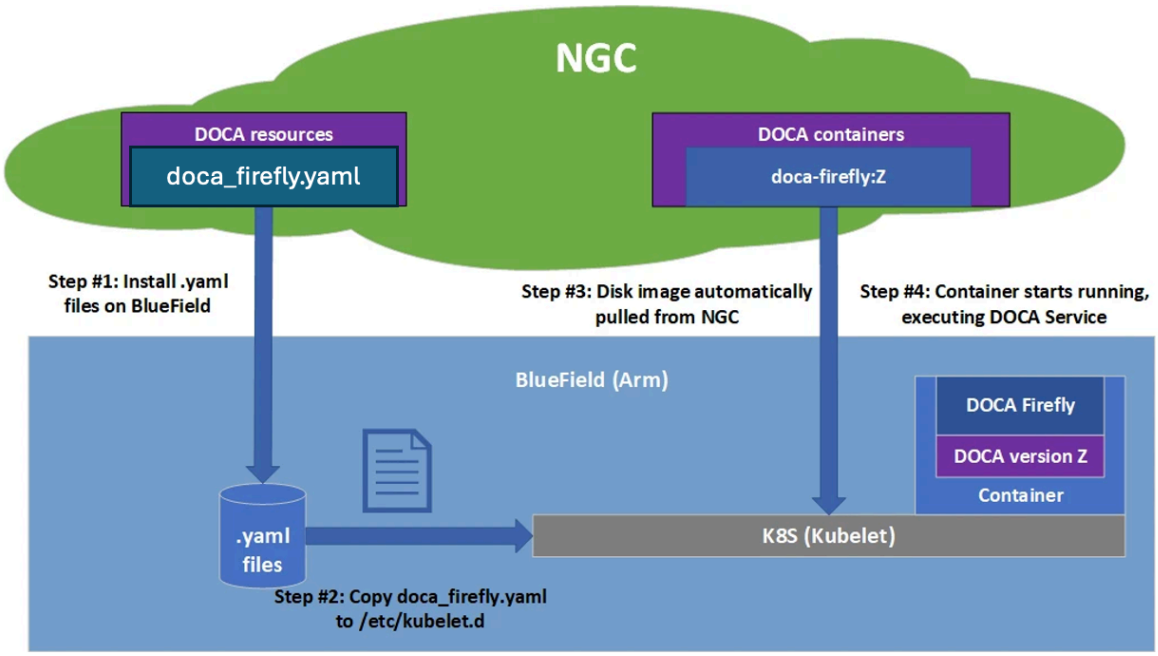
8.3.3 3. Container Deployment

Deploying containers on top of the BlueField DPU requires the following setup sequence:

1. Pull the container `.yaml` configuration files.
2. Modify the container's `.yaml` configuration file.
3. Deploy the container. The image is automatically pulled from NGC.

Some of the steps must only be performed once, while others are required before the deployment of each container.

What follows is an example of the overall setup sequence using the DOCA Firefly container as an example.



8.3.3.1 3.1. Pull Container YAML Configurations

This step pulls the `.yaml` configurations from NGC.

To pull the latest resource version, download the NVIDIA NGC CLI.

For more information regarding the NGC CLI, please refer to the *NGC Catalog User Guide*.

8.3.3.2 3.2. Container-specific Instructions

Some containers require specific configuration steps for the resources used by the application running inside the container and modifications for the **.yaml** configuration file of the container itself.

Refer to the container-specific instructions listed under the container's relevant page on NGC.

8.3.3.3 3.3. Structure of NGC Resource

Each DOCA service has a corresponding NGC resources page, referenced in the "Pull Container YAML Configurations" section. This resource contains a **configs** directory, which includes a dedicated subdirectory for each DOCA service version. The latest version contains all available **.yaml** configuration files and scripts.

DOCA Firefly example:

```
doca_firefly_1.6.13
    configs
        1.6.0
        ...
        1.7.1
          doca_firefly.yaml
```

In addition, the resource may contain a **scripts** directory, where services can provide helper scripts and additional configuration files.

Example folder structure for **scripts** directory:

```
+ doca_firefly_1.6.13
+-+ configs
| +--  ...
+-+ scripts
  | +-+ 1.6.0
  | | +-- ...
  | +-+ 1.7.0
  | | +-- ...
```

Users can deploy older DOCA service versions by accessing the relevant YAML file under **configs** and the associated scripts under the matching version directory in **scripts**.

8.3.3.4 3.4. Spawn Container

Once the desired **.yaml** file is updated, copy it to Kubelet's input folder. The following example uses **doca_firefly.yaml**, which corresponds to the DOCA Firefly service:

```
cp doca_firefly.yaml /etc/kubelet.d
```

Kubelet automatically pulls the container image from NGC and spawns a pod that runs the container. In this example, the DOCA Firefly service begins execution immediately, and its output can be viewed in the container logs.

8.3.3.5 3.5. Review Container Deployment

Follow the following steps to verify a new container deployment and troubleshoot any issues.

1. View currently active pods and their IDs:

```
sudo crictl pods
```

> It may take up to 20 seconds for the pod to start.

Look for a line matching your container, for example:

```
POD ID              CREATED             STATE   NAME                    NAMESPACE   ATTEMPT   RUNTIME
06bd84c07537e       4 seconds ago       Ready   doca-firefly-my-dpu     default     0         (default)
```

2. If no matching pod appears, view Kubelet logs for errors:

```
sudo journalctl -u kubelet --since -5m
```

Once the issue is resolved, proceed to the next steps.

> For more troubleshooting tips, refer to the *NVIDIA BlueField Platform Software Troubleshooting Guide*.

3. Verify the container image was downloaded from NGC:

```
sudo crictl images
```

Example output:

```
IMAGE                               TAG                 IMAGE ID            SIZE
k8s.gcr.io/pause                    3.9                 829e9de338bd5       268kB
nvcr.io/nvidia/doca/doca_firefly    1.1.0-doca2.0.2     134cb22f34611       87.4MB
```

4. List active containers:

```
sudo crictl ps
```

Look for a line matching your deployed container (boot time varies by image size):

```
CONTAINER           IMAGE               CREATED             STATE               NAME
ATTEMPT             POD ID              POD
b505a05b7dc23       134cb22f34611       4 minutes ago       Running             doca-firefly
0                   06bd84c07537e       doca-firefly-my-dpu
```

5. If the container is not running, list all recent container deployments:

```
sudo crictl ps -a
```

You may find that the container exited immediately due to an error:

```
CONTAINER           IMAGE               CREATED             STATE               NAME
ATTEMPT             POD ID              POD
de2361ec15b61       134cb22f34611       1 second ago        Exited              doca-firefly
1                   4aea5f5adc91d       doca-firefly-my-dpu
```

6. View container logs. Logs are available during the container's lifetime and for a short time after it exits:

```
sudo crictl logs <container-id>
```

Example:

```
$ sudo crictl logs de2361ec15b61
Starting DOCA Firefly - Version 1.1.0
...
Requested the following PTP interface: p10
Failed to find interface "p10". Aborting
```

For more information on **crictl**, refer to the Kubernetes documentation.

8.3.3.6 3.6. Stop Container

The recommended way to stop a pod and its containers is as follows:

1. Delete the YAML configuration file to instruct Kubelet to stop the pod:

```
rm /etc/kubelet.d/<file name>.yaml
```

2. Stop the pod directly (only if it still shows **Ready**)

```
sudo crictl stopp <pod-id>
```

3. Once the pod has stopped, it may also be necessary to stop the container itself:

```
sudo crictl stop <container-id>
```

8.3.4 4. Troubleshooting Common Errors

This section provides a list of common errors that may be encountered when spawning a container. These account for the vast majority of deployment errors and are easy to verify first before trying to parse the Kubelet journal log.

For more troubleshooting tips, refer to the NVIDIA BlueField Platform Software Troubleshooting Guide.

8.3.4.1 4.1. Yaml Syntax

The syntax of the **.yaml** file is extremely sensitive and minor indentation changes may cause it to stop working. The file uses spaces (' ') for indentations (two per indent). Using any other number of spaces causes an undefined behavior.

8.3.4.2 4.2. Huge Pages

The container only spawns once all the required system resources are allocated on the DPU and can be reserved for the container. The most notable resource is huge pages.

1. Before deploying the container, make sure that:
    a. Huge pages are allocated as required per container.
    b. Both the amount and size of pages match the requirements precisely.
2. Once huge pages are allocated, it is recommended to restart the container service to apply the change:

```
sudo systemctl restart kubelet.service
sudo systemctl restart containerd.service
```

3. Once the above operations are completed successfully, the container could be deployed (YAML can be copied to **/etc/kubelet.d**).

8.3.5 5. Advanced Troubleshooting

8.3.5.1 5.1. Manual Execution from Within Container - Debugging

> ⚠ The deployment described in this section requires an in-depth knowledge of the container's structure. As this structure might change from version to version, it is only recommended to use this deployment for debugging, and only after other debugging steps have been attempted.

Although most containers define the **entrypoint.sh** script as the container's ENTRYPOINT, this option is only valid for interaction-less sessions. In some debugging scenarios, it is useful to have better control of the programs executed within the container via an interactive shell session. Hence, the **.yaml** file supports an additional execution option.

Uncommenting (i.e., removing **#** from) the following 2 lines in the **.yaml** file causes the container to boot without spawning the container's entrypoint script.

```
# command: ["sleep"]
# args: ["infinity"]
```

In this execution mode, users can attach a shell to the spawned container:

```
crictl exec -it <container-id> /bin/bash
```

Once attached, users get a full shell session enabling them to execute internal programs directly at the scope of the container.

8.3.6 6. Air-gapped Container Deployment

Container deployment on the BlueField DPU can be done in air-gapped networks and does not require an Internet connection. As explained previously, per DOCA service container, there are 2 required components for successful deployment:

- Container image – hosted on NVIDIA's NGC catalog
- YAML file for the container

From an infrastructure perspective, one additional module is required:

- **k8s.gcr.io/pause** container image

8.3.6.1 6.1. Pulling Container for Offline Deployment

When preparing an air-gapped environment, users must pull the required container images in advance so they could be imported locally to the target machine:

```
docker pull <container-image:tag>
docker save <container-image:tag> > <name>.tar
```

The following example pulls DOCA Firefly **1.1.0-doca2.0.2**:

```
docker pull nvcr.io/nvidia/doca/doca_firefly:1.1.0-doca2.0.2
docker save nvcr.io/nvidia/doca/doca_firefly:1.1.0-doca2.0.2 > firefly_v1.1.0.tar
```

Some of DOCA's container images support multiple architectures, causing the **docker pull** command to pull the image according to the architecture of the machine on which it is invoked. Users may force the operation to pull an Arm image by passing the **--platform** flag:

```
docker pull --platform=linux/arm64 <container-image:tag>
```

8.3.6.2 6.2. Importing Container Image

After exporting the image from the container catalog, users must place the created ***.tar** files on the target machine on which to deploy them. The import command is as follows:

```
ctr --namespace k8s.io image import <name>.tar
```

For example, to import the firefly **.tar** file pulled in the previous section:

```
ctr --namespace k8s.io image import firefly_v1.1.0.tar
```

Examining the status of the operation can be done using the image inspection command:

```
crictl images
```

8.3.6.3 6.3. Built-in Infrastructure Support

The DOCA image comes pre-shipped with the **k8s.gcr.io/pause** image:

```
/opt/mellanox/doca/services/infrastructure/
    docker_pause_3_9.tar
    enable_offline_containers.sh
```

This image is imported by default during boot as part of the automatic activation of DOCA Telemetry Service (DTS).

> ⚠ Importing the image independently of DTS can be done using the **enable_offline_container.sh** script located under the same directory as the image's ***.tar** file.

This image can also be pulled and imported manually, using the following instructions:

- To export the image:

```
docker pull k8s.gcr.io/pause:3.9
docker save k8s.gcr.io/pause:3.9 > docker_pause_3_9.tar
```

- To import the image:

```
ctr --namespace k8s.io image import docker_pause_3_9.tar
crictl images
IMAGE                               TAG                 IMAGE ID            SIZE
k8s.gcr.io/pause                    3.9                 829e9de338bd5       268kB
```

8.3.7 7. DOCA Services for Host

A subset of the DOCA services is available for host-based deployment as well. This is indicated in those services' deployment and can also be identified by having container tags on NGC with the ***-host** suffix.

In contrast to the managed DPU environment, the deployment of DOCA services on the host is based on docker. This deployment can be extended further based on the user's own container runtime solution.

8.3.7.1 7.1. Docker Deployment

DOCA services for the host are deployed directly using Docker.

1. Make sure Docker is installed on your host. Run:

   ```
   docker version
   ```

   If it is not installed, visit the official Install Docker Engine webpage for installation instructions.
2. Make sure the Docker service is started. Run:

```
sudo systemctl daemon-reload
sudo systemctl start docker
```

3. Pull the container image directly from NGC (can also be done using the **docker run** command):
    a. Visit the NGC page of the desired container.
    b. Under the "Tags" menu, select the desired tag and click the paste icon so it is copied to the clipboard.
    c. The docker pull command will be as follows:

    ```
    sudo docker pull <NGC container tag here>
    ```

    For example:

    ```
    sudo docker pull nvcr.io/nvidia/doca/doca_firefly:1.1.0-doca2.0.2-host
    ```

    For DOCA services with deployments on both DPU and host, make sure to select the tag ending with **-host**.

4. Deploy the DOCA service using Docker:
    a. The deployment is performed using the following command:

    ```
    sudo docker run --privileged --net=host -v <host directory>:<container directory> -e <env variables> -it <container tag> /entrypoint.sh
    ```

    For more information, refer to Docker's official documentation.

    b. The specific deployment command for each DOCA service is listed in their respective deployment guide.

8.3.8 8. Container Deployment Using DMS

DOCA Management Service (DMS) is a client-server service that can run either on the host system or on BlueField Arm, and it can be accessed locally or remotely through gRPC-based management interfaces.

When deploying containers with DMS, follow the same methodology described earlier on page:

1. Pull the relevant YAML configuration
2. Apply any container-specific changes.
3. Deploy the container.
4. Verify that the pod and container started successfully.

On the DPU, this flow is based on standalone kubelet, while host-based DOCA services are deployed directly using Docker.

For more information about DMS architecture, configuration, and client commands, see the DOCA Management Service Guide.

8.4 DOCA Platform Framework (DPF)

- 8.4.1 Getting Started with DPF
- 8.4.2 Why DPF
- 8.4.3 Architecture
  - 8.4.3.1 Where DPF Fits
- 8.4.4 Deployment Models
- 8.4.5 Platform Support
- 8.4.6 DPU Services Supported by DPF
- 8.4.7 Service Function Chaining
- 8.4.8 DPF Validated Reference Deployment Guides
  - 8.4.8.1 Host Trusted
  - 8.4.8.2 Zero Trust
  - 8.4.8.3 Operations and Monitoring
- 8.4.9 NVIDIA Platform Integrations
- 8.4.10 Next Steps

NVIDIA DOCA Platform Framework (DPF) enables platform teams to provision and operate BlueField DPUs and their infrastructure services consistently at fleet scale. It is a cloud-native infrastructure orchestration platform and implementation blueprint within NVIDIA DOCA that extends cloud infrastructure control-plane operations to BlueField, turning each DPU into a programmable infrastructure services plane for networking, storage, security, and operations.

DPF provides an orchestration layer with declarative APIs for provisioning BlueField DPUs and deploying, chaining, and operating NVIDIA DOCA microservices and compatible third-party DPU services at cluster scale. DPF is intended for cloud and AI factory providers, telecommunications operators, enterprises, and infrastructure software vendors that need a consistent way to operate BlueField-enabled systems. Detailed installation, API, architecture, and operations information is available in the standalone DPF documentation.

8.4.1 Getting Started with DPF

This page provides a high-level overview of DPF. Detailed prerequisites and installation procedures are maintained in the standalone DPF documentation.

Start here

1. Review the current DPF documentation to verify the supported hardware and software combination.
2. Choose the Host Trusted or Zero Trust deployment model. See Deployment Models later on this page for an overview.
3. Follow the DPF Getting Started guide and select the installation procedure for your deployment model.

For an end-to-end validated configuration, select one of the DPF Validated Reference Deployment Guides later on this page.

8.4.2 Why DPF

BlueField offloads, accelerates, and isolates infrastructure processing from the host CPU. Operating BlueField at fleet scale requires platform teams to coordinate device provisioning, software and firmware lifecycle, service deployment, connectivity, upgrades, and health monitoring. Fragmented tools and device-by-device workflows make it difficult to keep software, configuration, and services consistent across the fleet. DPF provides a common orchestration layer for these operations.

DPF organizes these capabilities into two primary workflows:

- DPU provisioning and lifecycle management – Discovers and provisions BlueField DPUs, applies the BlueField bundle (BFB) and device configuration, integrates DPUs into a Kubernetes-managed environment, and coordinates upgrades and operational health.
- DPU service management and orchestration – Deploys, configures, updates, monitors, and removes NVIDIA DOCA microservices and compatible third-party services on BlueField DPUs.

DPF provides the following capabilities:

- Kubernetes APIs and Custom Resource Definitions (CRDs) for declarative DPU and service management
- Fleet-wide lifecycle management for BlueField software and services
- One or more dedicated DPU clusters for running infrastructure services separately from host workloads
- Coordinated deployment of multiple services through **DPUDeployment**
- Service function chaining for steering traffic through ordered NVIDIA and compatible third-party network services
- Host Trusted and Zero Trust deployment models
- Operational status, health conditions, observability integration, and troubleshooting tools
- An open-source framework and DPUService development model for ecosystem integration

8.4.3 Architecture

DPF's Kubernetes-based DPU control plane uses two cooperating cluster roles: a DPF management cluster (also called the host cluster in the standalone DPF documentation) and one or more DPU clusters. A complete DPF deployment requires both roles to provide the control and execution environments for BlueField lifecycle and DPU-service orchestration.

- The DPF management cluster runs the DPF Operator and controllers that manage BlueField provisioning, **DPUService** deployment, and service chains. In the default architecture, it also hosts the Kubernetes control-plane components for the DPU clusters.
- Each DPU cluster groups BlueField DPUs as Kubernetes nodes and provides the Kubernetes environment in which DPF-managed system components, NVIDIA DOCA microservices, and compatible third-party services operate.

DPF's native control interface is the DPF management cluster's Kubernetes API. Platform administrators can use it directly, or a higher-level infrastructure platform can translate infrastructure intent into DPF resources and consume their reported status. DPF continuously

checks and adjusts DPU clusters, BlueField DPUs, and DPU services so their actual state matches the declared state, reducing the need to manage individual DPUs during normal operations.

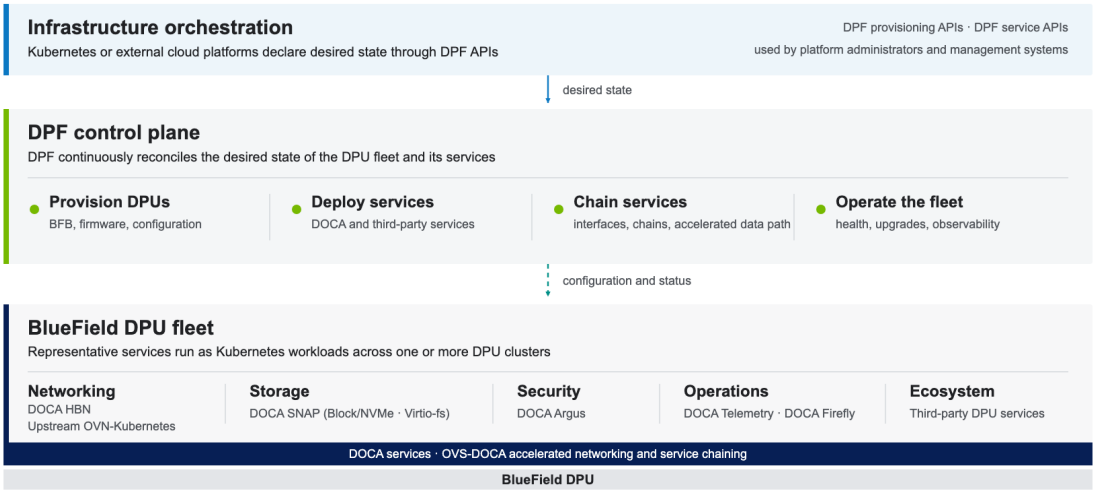


Figure 1. DPF high-level architecture showing infrastructure orchestration, the DPF control plane, and the BlueField DPU fleet.

DPF extends Kubernetes orchestration to one or more DPU clusters, coordinating BlueField lifecycle, DPU-service deployment and chaining, and fleet health. The services shown are representative; verify release-specific support in the standalone DPF documentation.

8.4.3.1 Where DPF Fits

Layer	Primary responsibility
Higher-level infrastructure platform	Owns applicable site, rack, host, tenant, cluster, and workload lifecycle. It can declare desired DPU and service state through DPF APIs.
DPF	Orchestrates BlueField provisioning and lifecycle, DPU clusters, DPU-service deployment and operations, and service-chain connectivity.
DOCA and compatible third-party services	Provide networking, storage, security, telemetry, and other runtime functions on BlueField.

For component-level details, refer to the standalone DPF documentation.

8.4.4 Deployment Models

DPF provides two deployment models so infrastructure operators can align BlueField DPU management with both their trust boundary and cloud operating model. In both models, the DPF control plane runs on Kubernetes and exposes DPF provisioning and service APIs through DPF custom resource definitions (CRDs). The models differ in where the DPF management cluster runs, whether the host is trusted, and how DPF reaches the DPU.

- Host Trusted – The DPF control plane runs on the infrastructure Kubernetes cluster that also manages the hosts, workloads, and infrastructure services. The host is a trusted

worker in that cluster, and its BlueField DPU is provisioned and managed in-band through the host. This model uses one infrastructure administration domain and software isolation.

- Zero Trust – The DPF control plane runs on a DPF management cluster separate from tenant hosts. The cloud infrastructure platform manages host lifecycle and declares the desired DPU and service state through DPF APIs. DPF provisions BlueField DPUs through the BMC using Redfish over a dedicated out-of-band (OOB) network and operates DPU services through each DPU cluster's Kubernetes control plane. Tenant hosts remain outside the DPF management plane, supporting separated infrastructure and tenant administration.

For an NVIDIA example of a higher-level cloud infrastructure integration, see NVIDIA Platform Integrations later on this page.

Deployment model	Management and orchestration path	Intended environment
Host Trusted	The DPF control plane shares the infrastructure Kubernetes cluster that manages the hosts. DPF manages each BlueField DPU in-band through its trusted host.	Single-tenant enterprise or telecommunications PaaS/CaaS environments with software isolation under a common infrastructure administrator.
Zero Trust	The DPF control plane runs on a DPF management cluster separate from tenant hosts. The cloud platform owns host lifecycle and can declare desired DPU and service state through DPF APIs. DPF orchestrates BlueField provisioning through BMC and Redfish over the dedicated OOB network and operates DPU services through each DPU cluster's Kubernetes control plane.	Multi-tenant or security-sensitive cloud infrastructure with separated infrastructure and tenant administration and an OOB DPU management path independent of tenant hosts.

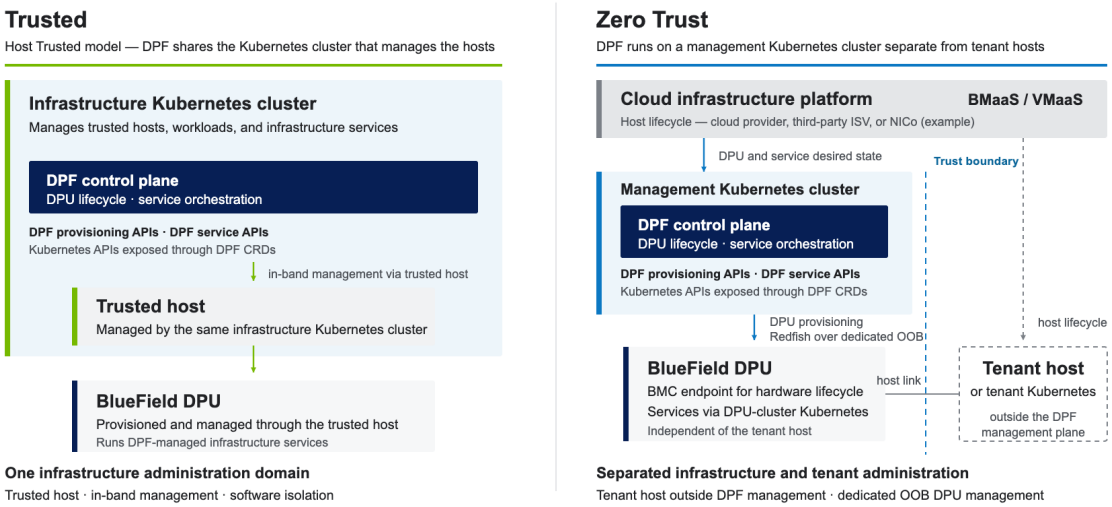


Figure 2. DPF Trusted and Zero Trust deployment models.

Both models run the DPF control plane on Kubernetes and expose the same provisioning and service APIs. In Host Trusted deployments, DPF shares the infrastructure Kubernetes cluster and manages the DPU in-band through the trusted host. In Zero Trust deployments, the cloud infrastructure platform manages host lifecycle and declares DPU and service state through DPF

APIs; DPF provisions BlueField through the BMC over a dedicated OOB network and operates services through each DPU cluster's Kubernetes control plane.

For deployment workflows and security considerations, refer to the standalone DPF documentation.

8.4.5 Platform Support

Before you deploy or upgrade

DPF and DOCA compatibility is release-specific. Open the current DPF documentation and select Platform Support to verify the qualified hardware and software combination. Then review Release Notes for service availability, limitations, and known issues.

Area	Support guidance
DPU hardware	Use a BlueField DPU generation and model listed in the current platform support matrix.
BlueField software	Use the qualified BFB, DOCA software, firmware, and NVIDIA Firmware Tools (MFT) combination for the selected DPF release.
Management stack	Use supported Kubernetes, Helm, host operating system, and container runtime versions.
Deployment models	Host Trusted and Zero Trust. Verify the prerequisites and limitations for the selected model.
DPU services	Verify service availability, quality level, and compatibility in the current DPF release notes and the corresponding DOCA service documentation.
Limitations	Review the current platform support matrix and release notes before deployment or upgrade.

8.4.6 DPU Services Supported by DPF

DPF can orchestrate NVIDIA DOCA, upstream, and compatible third-party services. The following table highlights representative services. Service availability, quality level, and compatibility are release-specific; verify them in the current DPF documentation and NVIDIA DOCA documentation.

Category	Representative services	Role in a DPF deployment
Networking and virtual networking	DOCA HBN, OVN-Kubernetes (upstream)	Provides routing and Kubernetes network offload.
Storage	DOCA SNAP: SNAP Block (NVMe) and SNAP Virtio-fs	Provides accelerated block storage through SNAP Block (NVMe) and file storage through SNAP Virtio-fs.
Security	DOCA Argus	Provides runtime threat detection from the isolated DPU trust domain.
Observability and operations	DOCA Telemetry Service · DOCA BlueMan · DOCA Firefly	Provides telemetry collection and dashboard visualization; DOCA Firefly adds PTP-based precision time synchronization for time-sensitive workloads and infrastructure services.

DPF can also orchestrate compatible third-party applications packaged as **DPUService** resources. A DPUService typically includes one or more Arm64 container images, a Helm chart, and the configuration required to deploy the application to one or more DPU clusters. A compatible third-party network service can also participate in a service chain when it exposes the required **DPUServiceInterface** endpoints.

> **Developing a DPU service?**
>
> See the DPUService Development Guide for packaging and integration requirements.

8.4.7 Service Function Chaining

Service function chaining steers traffic through an ordered set of network services on BlueField. DPF uses declarative APIs to describe the service interfaces and the connections between them:

- **DPUServiceInterface** declares a physical uplink, host-facing PF or VF, OVS patch interface, or interface exposed by a service.
- **DPUServiceChain** selects interfaces and defines how they are connected. A chain can reference interfaces exposed by NVIDIA services and compatible third-party **DPUService** resources.
- DPF service-chain controllers reconcile the configuration to the selected DPU clusters and program the required OVS ports and flows.
- OVS-DOCA is a DPF dependency included in the DOCA BFB image that DPF deploys on BlueField. It provides the accelerated data path and is not an optional chained **DPUService**.

The following conceptual host-to-fabric egress example offloads Kubernetes networking to OVN-Kubernetes, inserts an optional compatible third-party inline network function, such as a packet-inspection service, and adds DOCA HBN for BGP routing, EVPN, L4 firewall, NAT, and ECMP across two physical uplinks before traffic reaches the data-center fabric.

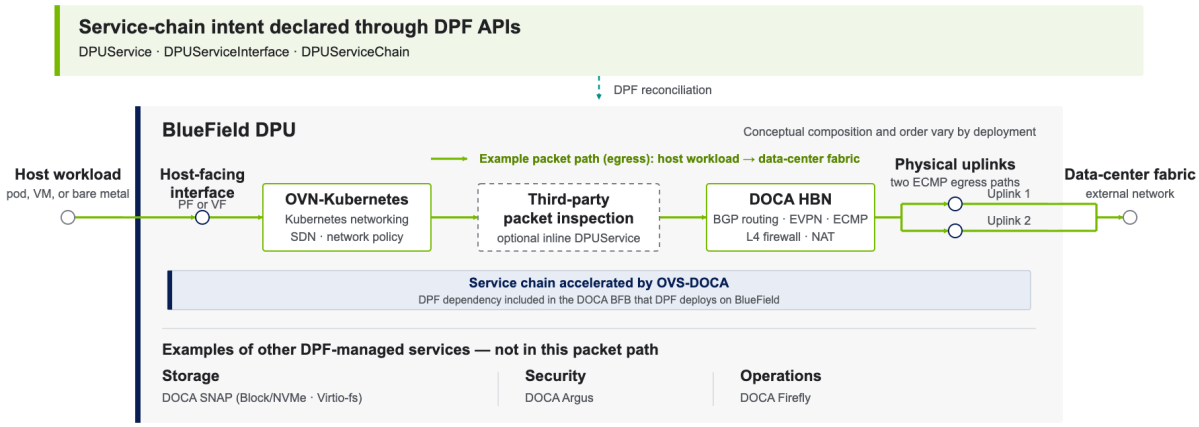


Figure 3. Conceptual DPF host-to-fabric service-chain example.

DPF reconciles the declared services, interfaces, and chain connectivity; OVS-DOCA provides the accelerated data path. The optional third-party network function represents a compatible service exposing the required **DPUServiceInterface** endpoints. DOCA HBN provides BGP/EVPN routing, L4

firewall and NAT, and ECMP across two physical uplinks. Actual service composition and order depend on the deployment.

Not every service deployed by DPF is an inline network function. Compatible third-party network services can participate in the packet path when they expose the required interfaces. Endpoint and supporting services can instead be lifecycle-managed alongside a chain. For example, DOCA SNAP presents accelerated block storage through SNAP Block (NVMe) and file storage through SNAP Virtio-fs over PCIe, DOCA Argus observes host activity from the DPU, and DOCA Firefly provides precision time synchronization. These services are not inserted into the network packet path shown in Figure 3.

8.4.8 DPF Validated Reference Deployment Guides

NVIDIA Networking Solutions publishes step-by-step Reference Deployment Guides (RDGs) for validated DPF configurations. Each RDG is a point-in-time validated deployment; follow the software versions, prerequisites, and bill of materials specified in that guide.

Choose an RDG first by deployment model, then by the service combination in its title. Refer to DPU Services Supported by DPF above for a summary of each service's role.

8.4.8.1 Host Trusted

Guide	What it demonstrates
HBN	BGP/EVPN routed networking and HBN network services.
OVN-Kubernetes	Accelerated Kubernetes pod networking on BlueField.
OVN-Kubernetes + HBN	Kubernetes pod networking with BGP/EVPN fabric routing.
HBN + SNAP Virtio-fs	HBN networking with accelerated Virtio-fs file storage.
Firefly + OVN-Kubernetes + HBN	Pod networking, fabric routing, and PTP time synchronization.
Multi-DPU: OVN-Kubernetes + HBN / HBN + SNAP Virtio-fs	Networking and storage services distributed across two DPUs.

8.4.8.2 Zero Trust

Guide	What it demonstrates
Baseline	Out-of-band BlueField provisioning through BMC and Redfish.
Argus	Zero Trust deployment with DPU-isolated runtime threat detection.
HBN	Zero Trust networking with BGP/EVPN routing.
HBN + Argus	HBN networking with DPU-isolated runtime threat detection.
DOCA Telemetry Service + BlueMan	DPU telemetry collection and dashboard visualization.
HBN + SNAP Virtio-fs	HBN networking with accelerated Virtio-fs file storage.
Multi-DPU: HBN / DOCA Telemetry Service + BlueMan	Networking and observability services distributed across two DPUs.

8.4.8.3 Operations and Monitoring

Guide	What it demonstrates
Centralized monitoring with DOCA Telemetry Service	Centralized telemetry and monitoring for the DPU fleet.

For the current catalog, refer to DPF – DOCA Platform Framework in NVIDIA Networking Solutions. For product architecture, supported configurations, and other deployment procedures, refer to the standalone DPF documentation.

8.4.9 NVIDIA Platform Integrations

DPF integrates with NVIDIA infrastructure platforms to extend BlueField lifecycle and DPU-service orchestration into broader AI infrastructure workflows.

- NVIDIA AI Enterprise – In supported NVIDIA AI Enterprise deployments, DPF provides BlueField lifecycle and DPU-service orchestration.
- NVIDIA DSX OS – In a DPF-integrated Zero Trust deployment, NVIDIA Infra Controller (NICo) orchestrates site, rack, and host lifecycle and can use DPF as the DPU provisioning strategy for selected hosts. NICo declares the required DPU and service state through DPF Kubernetes APIs; DPF orchestrates BlueField provisioning, DPU-cluster operations, and DPU-service deployments. For deployment details, refer to DPF Setup for NICo Integration.

8.4.10 Next Steps

If you want to...	Go to...
Install or get started with DPF	DPF Getting Started Guide
Verify supported versions, compatibility, limitations, and known issues	Open the current DPF documentation and select **Platform Support** and **Release Notes**.
Follow a validated deployment	DPF validated Reference Deployment Guides
Explore the services that DPF can orchestrate	Latest NVIDIA DOCA documentation
Develop or integrate a DPU service	DPUService Development Guide and NVIDIA DPF GitHub repository
Integrate Zero Trust DPF with NICo	DPF Setup for NICo Integration
Explore adjacent NVIDIA infrastructure platforms	NVIDIA AI Enterprise and NVIDIA DSX OS
Get product support	NVIDIA Enterprise Support

Additional resources: NVIDIA DOCA Software Platform · DPF product introduction · NVIDIA DSX OS overview

8.5 DOCA Argus Service Guide

This page provides installation, configuration, and usage instructions for the DOCA Argus Service.

8.5.1 Introduction

DOCA Argus is a service that runs on a supported NVIDIA® BlueField® DPU and provides real-time visibility into the attached Linux compute node. It uses hardware-level live machine introspection and NVIDIA DOCA DMA to read selected structures in volatile host memory and derive information about the host and its workloads. Because the service runs on the DPU in a trust domain separate from the host, it does not require a host-resident agent.

DOCA Argus evaluates collected activity with a policy engine and produces events and alerts for security and observability workflows. Its primary use cases include runtime workload threat detection, AI workload discovery, infrastructure mapping, and runtime observability across bare-metal, virtualized, and containerized environments. A single BlueField DPU running DOCA Argus can monitor the compute node to which it is attached.

8.5.2 Operation

DOCA Argus identifies the Linux kernel version running on the monitored host and applies a matching kernel-specific memory map to locate the structures it must inspect. The service supports hosts with x86 and Arm64 CPUs. Using DOCA DMA, it reads selected regions of host memory and decodes the raw data into logical information about processes, threads, execution states, and other system activity.

A policy engine filters the decoded activity and evaluates it against policy rules. Activity that matches one or more rules becomes meaningful activity that the service can report as an event or alert. DOCA Argus reports information about its own operational state as system activity messages.

8.5.2.1 Monitored Activity and Relationships

DOCA Argus can report the following workload and infrastructure information:

- Processes and threads, including execution state
- Executed binaries and loaded libraries, including SHA-256 hashes
- Execution paths and command-line arguments
- Process-level file access
- Process-level inbound and outbound network connections
- Exposed network services
- Workload type and deployment context, including bare-metal, virtual machine, and container environments
- Relationships among containers, Pods, virtual machines, the host operating system, processes, compute nodes, and GPUs
- Physical GPU attributes, including make, model, and serial number
- Mappings between workloads and physical or virtual GPUs
- Process-to-GPU utilization metrics and other GPU observability data

This information provides the common data foundation for the security, discovery, and observability use cases described below.

8.5.2.2 Events and Alerts

When raw activity matches one or more policy rules, DOCA Argus treats it as meaningful activity and includes it in one of the following message types:

- Event – One or more meaningful activities that describe the recorded operational state and provide situational awareness.
- Alert – One or more meaningful activities that indicate a threat or operational impact that requires investigation or response.
- System activity – Information about the operational state of the DOCA Argus service.

For details about message formats, local logging, telemetry delivery, and output fields, see the Log Types and Output sections.

8.5.3 Use Cases

DOCA Argus uses the monitored activity and relationships described above to support the following use cases.

8.5.3.1 Runtime Workload Threat Detection

Containerized AI applications are commonly instantiated from immutable images. Instances created from the same image should therefore exhibit consistent and predictable runtime behavior. DOCA Argus can compare observed activity with established behavioral profiles to identify deviations that may indicate compromise or malicious activity.

Relevant signals include executed binaries and loaded libraries, their associated hashes, execution paths, command-line arguments, process-level file access, and network connections. These signals can help detect integrity violations and suspicious behavior, including unexpected process execution, executed Bash commands, reverse shell activity, and unauthorized process execution.

External security platforms can combine DOCA Argus telemetry, events, and alerts with their own analytics, correlation logic, and threat intelligence.

8.5.3.2 AI Workload Discovery

DOCA Argus identifies how workloads are deployed and maps their relationships across the monitored compute node. It distinguishes workloads running on bare metal, in virtual machines, or in containers and can report relationships such as container-to-Pod, container-to-virtual-machine, and container-to-operating-system mappings.

DOCA Argus also reports container image hashes. External systems can correlate these hashes with public and private software repositories to help identify deployed AI software, AI models, large language models, and AI agents. DOCA Argus supplies the runtime and image-hash data used for this identification; repository correlation occurs outside the service.

Together with process, compute-node, and GPU relationship data, this information supports a current inventory and map of AI workloads and the infrastructure on which they run.

8.5.3.3 Runtime Observability

DOCA Argus provides operational visibility into how workloads execute and interact with memory, threads, the network, the file system, CPUs, and GPUs. It reports process-level network connections, file activity, exposed network services, and other runtime context.

For GPU observability, the service reports physical GPU attributes, maps workloads to physical or virtual GPUs, and provides process-to-GPU utilization metrics and related operational data. This information connects workload behavior to the compute and GPU resources used by each workload.

8.5.4 Requirements

- Operates only on BlueField-3 or later configured in DPU mode (see *BlueField Modes of Operation*).
- Requires firmware version 24.35.0388 or later.
- Supported BlueField image versions: 4.11.0 or later.
- The DOCA Argus service container must run in privileged mode to enable full-system DMA reads.

8.5.5 Limitations

- Tested against KVM-based hypervisors only.
- Supports Linux-based OSs (bare-metal, virtualization, containers).
- Kata Containers are supported only if NVIDIA-DPU support is enabled.

8.5.6 Prerequisites

1. Configure BlueField firmware. On BlueField, configure the PF BAR register:

   ```
   dpu> mlxconfig -d /dev/mst/<mst_device> s PF_BAR2_SIZE=2 PF_BAR2_ENABLE=1
   ```

   Replace **<mst_device>** with **mt41692_pciconf0** for BlueField-3.
2. Enable IOMMU passthrough only if it is not already enabled.
   a. Edit the GRUB configuration:

      ```
      host> sudo vim /etc/default/grub
      ```

   b. Add the following value to **GRUB_CMDLINE_LINUX_DEFAULT**:

      ```
      iommu=pt <intel/amd>_iommu=on
      ```

   c. Apply the changes:
      - Ubuntu:

        ```
        sudo update-grub
        ```

      - CentOS or RHEL:

```
sudo grub2-mkconfig -o /boot/grub2/grub.cfg
```

d. Reboot the host.

3. Prepare the target system. DOCA Argus automatically generates the required symbol and memory-region configuration at runtime. Complete the remaining steps only if automatic generation fails.
4. For manual profile generation, install the required debug packages on the target:
   - Ubuntu:

```
sudo tee /etc/apt/sources.list.d/ddebs.list << EOF
deb http://ddebs.ubuntu.com/ $(lsb_release -cs) main restricted universe multiverse
deb http://ddebs.ubuntu.com/ $(lsb_release -cs)-updates main restricted universe multiverse
deb http://ddebs.ubuntu.com/ $(lsb_release -cs)-proposed main restricted universe multiverse
EOF
sudo apt install ubuntu-dbgsym-keyring
sudo apt-get update
sudo apt-get install linux-image-$(uname -r)-dbgsym
```

   - CentOS or RHEL:

```
sudo yum install --enablerepo=base-debuginfo \
  kernel-devel-$(uname -r) \
  kernel-debuginfo-$(uname -r) \
  kernel-debuginfo-common-$(uname -m)-$(uname -r)
```

5. Make **doca_apsh_config.py** available on the target using one of the following methods:
   - If DOCA is installed on the target, use **/opt/mellanox/doca/tools/doca_apsh_config.py** and continue to the next step.
   - Otherwise, copy the script from BlueField:

```
scp <bf-user>@<bf-ip>:/opt/mellanox/doca/tools/doca_apsh_config.py /tmp/
```

6. Download a **dwarf2json** release binary from volatilityfoundation/dwarf2json.
7. Create the JSON files:

```
cd /opt/mellanox/doca/tools/
pip3 install psutil pdbparse
python3 doca_apsh_config.py --files memregions symbols --os <windows/linux> --path <path-to-dwarf2json>
cp /opt/mellanox/doca/tools/*.* <shared-folder>
dpu> scp <shared-folder>/* <path-to-app-shield-binary>
```

8.5.7 Service Deployment

- For DPU container deployment, see the DOCA Container Deployment Guide.
- For Argus-specific deployment, see the service container page.
- For offline deployment, see *Offline Deployment* in the DOCA Container Deployment Guide.

8.5.8 Service Configuration

Argus configuration is managed through **SERVICE_CONFIG_FILE** in the container YAML.

8.5.8.1 Service

- Immediate shutdown – Terminates immediately on **SIGINT** or **SIGTERM** and skips graceful shutdown.

- Service log level – Sets DOCA logging verbosity. The default is `50` (`INFO`). Supported values are `10` (`DISABLE`), `20` (`CRITICAL`), `30` (`ERROR`), `40` (`WARNING`), `50` (`INFO`), `60` (`DEBUG`), and `70` (`TRACE`).
- System scanner sleep time – Sets the delay between scans. Supported units are `s`, `m`, and `ms`.

8.5.8.2 DOCA Argus Configuration

- Auto Scan – Scans all available systems unless the `systems` section is defined.
- Profile generation – Automatically generates the required kernel data at runtime when no prebuilt data is available. Initialization can take longer when profile generation is required.
- Default – Defines configuration values that apply unless overridden under `systems`.
- Systems – Defines the monitored systems and their overrides.

8.5.8.3 Per-system Configurations

- Representor ID – Specifies the VU ID of the VF or PF to track.
  - PF:

    ```
    host> lspci -vv -s <PF_pci_address> | grep VU | cut -d " " -f 4
    ```

  - VF – Append `VF<x>` to the PF VU ID. For example: `MT2333XZ06YAMLNXS0D0F0VF1`.
- Memory regions path – Specifies a JSON file path, or `auto`, for the host OS memory map.
- OS symbol path – Specifies a JSON file path or directory, or `auto`.
- OS type – Specifies `Linux` or `Windows`.
- DMA device name – Matches the representor ID. List devices with:

  ```
  dpu> ibv_devinfo | grep 'hca_id' | awk '{print $2}'
  ```

- Service log level – Overrides the service logging verbosity.
- SDK log level – Sets SDK logging verbosity.
- Limits – Sets maximum values for string length, processes, file descriptors, threads, virtual memory areas, and kernel modules.
- Events
  - Container filter – Includes or excludes containerized processes.
  - Container enrichment – Enriches messages from containerized processes with container and Kubernetes Pod metadata. Enabled by default.
  - SBOM – Lists SHA signatures of approved executables and libraries.

8.5.8.4 Collection

- Events – Enables or disables each event type.

8.5.8.5 Output

- Log events to stdout – Enables logging to standard output.
- Log folder path – Specifies the directory for file logs.
- Log threshold size – Rotates logs when they reach this size.

- Log max files count – Sets the maximum number of rotated log files.
- Telemetry address – Specifies the aggregator address.
- Telemetry tag – Specifies the tag used by Fluent Bit.
- Telemetry format – Specifies **JSON** or **syslog**.
- Telemetry user data – Specifies custom metadata to include in telemetry records.

8.5.9 Log Types

8.5.9.1 Standard Output

Displays important service messages, including version information, successful startups, and startup failures.

8.5.9.2 Debug Log Output

Provides complete debug output, including partial event data, trace logs, and collection failures. These logs are stored in **/var/log/doca_argus/**.

8.5.9.3 Event Log Output

Stores the complete event log in JSON format in the configured log folder. For local storage, Linux **logrotate** manages log rotation. You can override the default configuration in **/etc/cron.d/logrotate** and **/etc/logrotate.d/argus**.

8.5.9.4 Telemetry Output

DOCA Argus can produce telemetry records in JSON or syslog format.

Telemetry is disabled by default. To enable it, set **telemetry_address** in the service configuration and ensure that **telemetry_tag** matches the tag in the telemetry pipeline configuration.

8.5.9.4.1 Fluent Bit

The DOCA Argus container page provides YAML configuration examples for Fluent Bit and Vector. Use the applicable YAML file as the starting point for your telemetry deployment.

Fluent Bit runs independently from the DOCA Argus service. To run Fluent Bit locally on the DPU, use an input configuration similar to the following:

```
[INPUT]
    Name                tcp
    Tag                 <your_preferred_tag>
    Listen              0.0.0.0
    Port                24224
    Format              json
```

For Splunk, add the following encapsulation filter:

```
[FILTER]
    Name                nest
    Match               *
    Operation           nest
    Wildcard            *
```

```
    Nest_under            event
```

The following example forwards telemetry to Elasticsearch:

```
[INPUT]
    Name                  tcp
    Tag                   elastic_forward_input
    Listen                0.0.0.0
    Port                  24224
    Format                json

[SERVICE]
    Log_Level             info

[OUTPUT]
    Name                  es
    Match                 *
    Host                  <elasticsearch_ip>
    Port                  <elasticsearch_port>
    Index                 argus
    Suppress_Type_Name    On
    Log_Level             info
```

Run Fluent Bit with the configuration:

```
docker run --rm --net=host -v <path_to_fluentbit_conf_file>:/fluent-bit/etc/fluent-bit.conf --name fluent_bit -it
fluent/fluent-bit
```

For additional output plugins, see the Fluent Bit manual.

8.5.9.4.2 Vector

Vector is an observability data pipeline that can receive DOCA Argus telemetry over TCP, decode JSON records, and forward them to supported destinations such as Elasticsearch. Vector can run independently from the DOCA Argus service or as a sidecar container in the same Kubernetes Pod.

Set **telemetry_address** to the Vector listener address, such as **127.0.0.1:24224**, and set **telemetry_format** to **json**. The following configuration receives JSON telemetry and forwards it to Elasticsearch:

```
data_dir: /vector-data-dir

sources:
  tcp_json_in:
    type: socket
    mode: tcp
    address: "127.0.0.1:24224"
    decoding:
      codec: json

sinks:
  elasticsearch_out:
    type: elasticsearch
    inputs: ["tcp_json_in"]
    endpoints:
      - "http://<elasticsearch_ip>:<elasticsearch_port>"
    bulk:
      index: "argus"
```

Start Vector with the configuration file:

```
vector --config /etc/vector/vector.yaml
```

For Kubernetes deployments, Vector can run alongside the DOCA Argus container and share a configuration volume. DOCA Argus sends telemetry to **127.0.0.1:24224**, and Vector listens on the same Pod network and forwards records to the configured sink.

```
containers:
  - name: vector
    image: timberio/vector:0.43.1-alpine
    imagePullPolicy: IfNotPresent
    args: ["--config", "/etc/vector/vector.yaml"]
    volumeMounts:
      - mountPath: /etc/vector
        name: conf-vector-config
```

Ensure that the Vector configuration volume is populated before the Vector container starts. For additional source, transform, and sink options, see the Vector documentation.

8.5.10 Reported Activities

8.5.10.1 Naming Conventions

The message schema uses the following naming conventions:

- Parameter names use lowercase letters.
- Names are explicit.
- Underscores (`_`) separate words.
- Parameter names for measurements include the unit, such as **`average_packet_size_bytes`**.
- Enum values use uppercase letters and begin with a letter.
- Path values use Linux forward slashes (**`/`**) to avoid escaping. For example: **`/Users/askywalker`**.
- Timestamps use ISO 8601 format: **`YYYY-MM-DDTHH:MM:SS.nnnnnnnnn`**.
- Empty fields use **`null`**.

8.5.10.2 Message Type

Raw activities are collected from host memory and used to describe the operational state of a workload. The policy engine filters irrelevant raw activity and reports meaningful data. A raw activity that matches one or more policy rules can be reported as an event or alert.

- Event – One or more meaningful activities that describe the recorded state and provide situational awareness.
- Alert – One or more meaningful activities that indicate a threat or impact requiring investigation or response.
- System activity – Information about the operational state of the DOCA Argus service.

8.5.10.3 Message Structure

Every reported activity includes common message metadata, BlueField system information, workload information, and activity-specific data.

8.5.10.3.1 Schema

Parameter	Type	Parent Object	Description
`message_header`	object	—	Root object containing the message metadata.
`vendor_name`	enum	**`message_header`**	Vendor name. Value: **`NVIDIA`**.

Parameter	Type	Parent Object	Description
product_name	enum	message_header	Product name. Value: DOCA_ARGUS.
product_version	string	message_header	Product version.
message_type	enum	message_header	One of EVENT, ALERT, or SYSTEM_ACTIVITY.
severity	enum	message_header	One of INFO, ERROR, WARNING, MEDIUM, HIGH, or CRITICAL.
schema_version	string	message_header	Message schema version.
message_id	string	message_header	Unique message identifier.
occurred_message_time_iso_8601_ns	string	message_header	Timestamp when the message occurred.
message_timezone	string	message_header	Time zone of the message origin.
user_data	string	message_header	Configured user data included in every message.
bluefield_system_information	object	message_header	Information about the BlueField system.
bluefield_networking_interfaces	array	bluefield_system_information	Configured BlueField interfaces, including names, IP addresses, and MAC addresses.
bluefield_network_interface_name	string	bluefield_networking_interfaces	Interface name.
bluefield_network_interface_mac_address	string	bluefield_networking_interfaces	Interface MAC address.
bluefield_network_interface_ipv4_address	string or array	bluefield_networking_interfaces	IPv4 addresses associated with the interface.
bluefield_network_interface_ipv6_address	string or array	bluefield_networking_interfaces	IPv6 addresses associated with the interface.
workload_information	object	message_header	Information about the monitored workload.
unique_identifier	string	workload_information	Workload identifier. This value is the configured system name or the VUID for automatically scanned systems.
os_version	string	workload_information	Workload OS version.
boot_uuid	string	workload_information	Unique identifier for the current kernel boot.
hostname	string	workload_information	Workload hostname. This value is not refreshed until DOCA Argus restarts.
workload_networking_interfaces	array	workload_information	Workload interfaces, including names, IP addresses, and MAC addresses.

Parameter	Type	Parent Object	Description
workload_network_interface_name	string	workload_networking_interfaces	Interface name.
workload_network_interface_mac_address	string	workload_networking_interfaces	Interface MAC address.
workload_network_interface_ipv4_address	string or array	workload_networking_interfaces	IPv4 addresses associated with the interface.
workload_network_interface_ipv6_address	string or array	workload_networking_interfaces	IPv6 addresses associated with the interface.
container_context	object	workload_information	Container and Kubernetes context when container enrichment is available.
container_id	string	container_context	Unique container identifier.
container_name	string	container_context	Container name.
image_name	string	container_context	Container image name.
image_digest	string	container_context	SHA-256 digest that identifies the container image build.
image_manifest_digest	string	container_context	SHA-256 digest of the container image manifest.
pod_name	string	container_context	Name of the Kubernetes Pod containing the container.
pod_uid	string	container_context	Unique identifier of the Kubernetes Pod.
pod_ip_address	string	container_context	IP address of the Kubernetes Pod.
enrichment_status	enum	container_context	complete or partial ; indicates whether context collection completed.
activity_data	object	message_header	Activity-specific information.
name	string	activity_data	Name of the event, alert, or system activity.
<details>	object	activity_data	Fields specific to the reported activity.

8.5.10.3.2 Example JSON Message

The following example illustrates the common message structure. Activity-specific fields vary by activity name.

```
{
  "vendor_name": "NVIDIA",
  "product_name": "DOCA_ARGUS",
  "product_version": "<version>",
  "message_type": "<EVENT | ALERT | SYSTEM_ACTIVITY>",
  "severity": "<INFO | ERROR | WARNING | MEDIUM | HIGH | CRITICAL>",
  "schema_version": "1.0",
  "message_id": "<unique_message_id>",
  "occurred_message_time_iso_8601_ns": "2024-10-31T17:20:10.123456789+02:00",
  "message_timezone": "UTC",
  "user_data": "<string>",
  "bluefield_system_information": {
    "bluefield_networking_interfaces": []
  },
  "workload_information": {
    "unique_identifier": "<identifier>",
    "os_version": "Linux kernel <version>",
    "boot_uuid": "<uuid>",
    "hostname": "<hostname>",
    "workload_networking_interfaces": [],
    "container_context": {
      "container_id": "<container_id>",
      "container_name": "<container_name>",
      "image_name": "<image_name>",
      "image_digest": "sha256:<digest>",
      "image_manifest_digest": "sha256:<digest>",
      "pod_name": "<pod_name>",
      "pod_uid": "<pod_uid>",
      "pod_ip_address": "<pod_ip_address>",
      "enrichment_status": "<complete | partial>"
    }
  },
  "activity_data": {
    "name": "<activity_name>",
    "<activity_details>": {}
  }
}
```

8.5.11 Alerts, Events, and System Activity Messages

This section is the reference for the messages that DOCA Argus can generate. The activity name is the literal value reported in **activity_data.name**. For definitions of the data objects included with each message, see Activity Data Reference.

8.5.11.1 Events and Alerts

8.5.11.1.1 Bash Shell

Message type	Severity	Activity name	Activity data	Description
EVENT	**INFO**	**Shell Command Executed**	Process; Bash shell command	Reports one or more Bash commands that were executed.
ALERT	**HIGH**	**Shell History Cleared**	Process; Bash shell command	Reports that the Bash shell history was cleared.
ALERT	**HIGH**	**Shell History Disabled**	Process; Bash shell command	Reports that Bash shell history was disabled.

Only the **bash** shell is supported.

8.5.11.1.2 Containers

Message type	Severity	Activity name	Activity data	Description
EVENT	INFO	Container Started	Process	Reports that a container instance started.
EVENT	INFO	Container Terminated	Process	Reports that a container instance terminated.

When container enrichment is enabled, container and Kubernetes metadata is reported in **workload_information.container_context**. See Message Structure.

8.5.11.1.3 File Descriptors

Message type	Severity	Activity name	Activity data	Description
EVENT	INFO	File Descriptor Open	Process; File descriptor	Reports that a file descriptor was opened.
EVENT	INFO	File Descriptor Close	Process; File descriptor	Reports that a file descriptor was closed.
ALERT	MEDIUM	File Descriptor File Content Change	Process; File descriptor	Reports that a regular-file descriptor opened with write permissions was closed and the file content might have changed.

8.5.11.1.4 GPU Telemetry

Message type	Severity	Activity name	Activity data	Description
EVENT	INFO	GPU Identity	GPU telemetry	Reports the make and model of each GPU on the AI node. By default, DOCA Argus generates one event per GPU every minute.
EVENT	INFO	GPU Metrics	GPU telemetry	Reports operational telemetry for each GPU. By default, DOCA Argus generates one event per GPU every second.
EVENT	INFO	GPU Topology	GPU telemetry	Reports how GPUs and other devices are connected. By default, DOCA Argus generates one event every minute.

8.5.11.1.5 Kernel Modules

Message type	Severity	Activity name	Activity data	Description
EVENT	INFO	Kernel Module Changed	Kernel module	Reports a change to the parameters of a loaded kernel module.
EVENT	INFO	Kernel Module Loaded	Kernel module	Reports loaded kernel modules during initialization and each newly loaded kernel module.
EVENT	INFO	Kernel Module Unloaded	Kernel module	Reports that a kernel module was unloaded.

8.5.11.1.6 Network Connections

Message type	Severity	Activity name	Activity data	Description
EVENT	INFO	Network Connection Created	Process; Network connection; File descriptor; Thread	Reports that a TCP connection was created.
EVENT	INFO	Network Connection Terminated	Process; Network connection; File descriptor; Thread	Reports that a TCP connection terminated.
ALERT	LOW	TCP Connection Excessive Data	Process; Network connection; File descriptor; Thread	Reports that incoming or outgoing data on a TCP connection exceeded its configured threshold. Incoming and outgoing thresholds are configured separately.
ALERT	LOW	TCP Long Lasting Connection	Process; Network connection; File descriptor; Thread	Reports that the duration of a TCP connection exceeded its configured threshold.
EVENT	INFO	TCP Network Connection State Change	Process; Network connection; File descriptor; Thread	Reports a TCP state change, such as SYN_SENT or SYN_RECEIVED.
EVENT	INFO	TCP Network Connections Status	Process; Network connection; File descriptor; Thread	Periodically reports open TCP connections for each process, including packet and byte counts. Disabled by default.
ALERT	HIGH	Reverse Shell Detected	Process; Network connection; File descriptor; Thread	Reports a process whose standard input is redirected to a remotely connected socket.

8.5.11.1.7 Process Memory

Message type	Severity	Activity name	Activity data	Description
EVENT	INFO	Process Memory Created	Process; Process memory	Reports that a virtual memory area, such as a heap, stack, or executable area, was created. Disabled by default.
EVENT	INFO	Process Memory Terminated	Process; Process memory	Reports that a virtual memory area is no longer present. Disabled by default.
EVENT	WARNING	New Executable Anonymous Memory Mapped	Process; Process memory	Reports that an executable anonymous memory area was mapped.
ALERT	MEDIUM	Executable Permissions Added	Process; Process memory	Reports that executable permissions were added to a memory area.
ALERT	MEDIUM	Executable Permissions Removed	Process; Process memory	Reports that executable permissions were removed from a memory area.
EVENT	INFO	New File Mapped	Process; Process memory; Runtime manifest	Reports that a file was mapped into process memory.
EVENT	INFO	File Unmapped	Process; Process memory	Reports that a memory-mapped file was unmapped.

8.5.11.1.8 Processes

Message type	Severity	Activity name	Activity data	Description
EVENT	INFO	Process Created	Process	Reports that a process was created.
EVENT	INFO	Process Terminated	Process	Reports that a process terminated.
EVENT	WARNING	Process Zombie	Process	Reports a process in the zombie state.
ALERT	HIGH	Process Hidden	Process	Reports a process in a hidden state.

8.5.11.1.9 Runtime Manifest

Message type	Severity	Activity name	Activity data	Description
ALERT	HIGH	Foreign Binary Executed	Process; Process memory; Runtime manifest	Reports execution of a binary that was not included in the original container image or that differs from the original binary.
ALERT	HIGH	Binary Executed Not as Intended	Process; Process memory; Runtime manifest; Expected behavior	Reports execution of an original-image binary with command-line arguments or from a path that does not match its expected behavior.
ALERT	HIGH	Foreign Binary Executed File Size Mismatch	Process; Process memory; Runtime manifest; Expected behavior	Reports that an executed binary differs in size from the corresponding binary in the original container image.
ALERT	HIGH	Foreign Library Loaded	Process; Process memory; Runtime manifest	Reports loading of a library that was not included in the original container image or that differs from the original library.
ALERT	HIGH	Foreign Library Loaded File Size Mismatch	Process; Process memory; Runtime manifest; Expected behavior	Reports that a loaded library differs in size from the corresponding library in the original container image.

8.5.11.1.10 Threads

Message type	Severity	Activity name	Activity data	Description
EVENT	INFO	Thread Created	Process; Thread	Reports that a thread was created.
EVENT	INFO	Thread Terminated	Process; Thread	Reports that a thread terminated.

8.5.11.2 System Activity Messages

The following messages use the `SYSTEM_ACTIVITY` message type and report the operational state of the DOCA Argus service.

8.5.11.2.1 Service Lifecycle

Severity	Activity name	Description
INFO	Service Initialization Started	The DOCA Argus initialization stage started.

Severity	Activity name	Description
INFO	Service Initialization Successful	The DOCA Argus initialization stage completed successfully.
ERROR	Service Initialization Failed	DOCA Argus initialization failed.
ERROR	Service Runtime Failure	A critical internal service error occurred, and DOCA Argus is offline.
INFO	Service Gracefully Shutdown	DOCA Argus shut down gracefully after a user request.

8.5.11.2.2 Host Initialization and OS Discovery

Severity	Activity name	Description
INFO	Host Initialization Started	Workload detection started.
INFO	Host Initialization Successful	Workload detection completed successfully.
ERROR	Host Initialization Failed	Workload detection failed.
INFO	OS Identifier Discovery Extended	Workload OS discovery is taking longer than expected.
INFO	OS Identifier Found	DOCA Argus detected the workload OS.
ERROR	Unable to Determine Target OS	DOCA Argus could not detect the workload OS.

8.5.11.2.3 OS Profile Processing

Severity	Activity name	Description
INFO	Load Memory Regions Failed	DOCA Argus failed to load memory-region information.
INFO	Load Memory Regions Successful	DOCA Argus loaded memory-region information.
INFO	Loading Profile Candidate	DOCA Argus identified an OS profile candidate.
ERROR	No Matching Profile Found	No matching OS profile was found.
ERROR	Profile Parsing Failed	DOCA Argus could not parse an OS profile.
INFO	Profile Verification Successful	DOCA Argus initialized successfully with the selected OS profile.
ERROR	Profile Verification Failed	Verification of an OS profile failed. DOCA Argus continues with subsequent profile candidates.

8.5.11.2.4 Resource Limits

Severity	Activity name	Description
MEDIUM	File Descriptor Limit Reached	The configured file descriptor limit was reached.
MEDIUM	Kernel Module Limit Reached	The configured kernel module limit was reached.
MEDIUM	Process Limit Reached	The configured process limit was reached.
MEDIUM	Process Memory Limit Reached	The configured virtual memory area limit was reached.
MEDIUM	Threads Limit Reached	The configured thread limit was reached.

8.5.11.2.5 Configuration and Data Collection

Severity	Activity name	Description
ERROR	Details Gathering Failed	DOCA Argus failed to collect required information.
ERROR	Invalid SBOM Configuration File	The SBOM configuration file is invalid.
ERROR	Invalid SBOM Configuration Folder	The configured SBOM folder is invalid.

8.5.12 Activity Data Reference

The `activity_data` object contains information specific to the reported event, alert, or system activity. The fields included in a message depend on its activity name. See Alerts, Events, and System Activity Messages for the data objects associated with each message.

8.5.12.1 Bash Shell Command Data

Object: `shell_command_details`

Attribute	Description
`shell_command_execution_history`	JSON array of command records. Each record contains `command_execution_time_iso_8601_ns` and `command_line`.
`command_execution_time_iso_8601_ns`	Time when the command was executed, in ISO 8601 format with nanosecond precision.
`command_line`	Executed command line.

8.5.12.2 Container Context

Container and Kubernetes metadata is reported in `workload_information.container_context`, not in a separate activity-data object. See Message Structure.

8.5.12.3 File Descriptor Data

Object: `file_descriptor_details`

Attribute	Description
`file_descriptor_index`	File descriptor identifier.
`file_descriptor_name`	File descriptor name.
`file_descriptor_file_type`	File type, such as a socket, regular file, or pipe.
`file_descriptor_file_mode`	Access mode and status flags, such as `READ_WRITE` or `NON_BLOCKING`.
`file_descriptor_inode_number`	Identifier of the inode associated with the file descriptor.
`file_descriptor_inode_device_identifier`	Major and minor device numbers identifying the file system that contains the inode.
`file_descriptor_inode_permissions`	File permissions.
`file_descriptor_inode_uid`	User ID of the file owner.
`file_descriptor_inode_gid`	Group ID of the file owner.
`file_descriptor_inode_size_bytes`	File size, in bytes.
`file_descriptor_inode_deleted`	Indicates whether the file was deleted.
`file_descriptor_inode_hash_sha1`	SHA-1 hash of the file. Calculated only for `File Descriptor File Content Change` alerts.
`file_descriptor_inode_hash_sha256`	SHA-256 hash of the file. Calculated only for `File Descriptor File Content Change` alerts.
`file_descriptor_inode_hash_md5`	MD5 hash of the file. Calculated only for `File Descriptor File Content Change` alerts.
`file_descriptor_thread_id_array`	Identifiers of threads that use the file descriptor.
`file_descriptor_inode_change_time_iso_8601_ns`	Time when the inode metadata last changed, in ISO 8601 format with nanosecond precision.
`file_descriptor_inode_modify_time_iso_8601_ns`	Time when the file content last changed, in ISO 8601 format with nanosecond precision.
`file_descriptor_inode_access_time_iso_8601_ns`	Time when the file content was last accessed, in ISO 8601 format with nanosecond precision.

8.5.12.4 GPU Telemetry Data

Field-level definitions for GPU telemetry messages are not currently documented in this section.

8.5.12.5 Kernel Module Data

Object: `kernel_module_details`

Attribute	Description
`kernel_module_name`	Name registered with the Linux kernel.
`kernel_module_size`	Size of the loaded module in memory, in bytes.
`kernel_module_state`	Operational state, such as loaded, live, or unloading.
`kernel_module_address`	Kernel virtual memory address at which the module is loaded.
`kernel_module_parameters`	Module parameters and their current values.
`kernel_module_used_by`	Modules or kernel subsystems that depend on or reference the module, including the reference count when available.
`kernel_module_taints`	Linux kernel taint flags associated with the module.
`kernel_module_source`	Origin of the module, such as its file path, package, or provider.

8.5.12.6 Network Connection Data

Object: `network_connection_details`

Attribute	Description
`file_descriptor_id`	Identifier of the file descriptor associated with the socket.
`connection_state`	TCP connection state.
`protocol`	Network protocol.
`address_family`	Address family that defines how the kernel interprets the network address.
`local_address`	IP address of the local endpoint.
`local_port`	Port of the local endpoint.
`peer_address`	IP address of the remote endpoint.

Attribute	Description
`peer_port`	Port of the remote endpoint.
`tcp_bytes_in`	Number of data bytes received.
`tcp_bytes_out`	Number of data bytes sent.
`tcp_segments_in`	Number of TCP segments received.
`tcp_segments_out`	Number of TCP segments sent.
`workload_network_interface_name`	Workload interface name.
`workload_network_interface_mac_address`	Workload interface MAC address.
`workload_network_interface_ipv4_address`	IPv4 addresses associated with the workload interface.
`workload_network_interface_ipv6_address`	IPv6 addresses associated with the workload interface.
`tcp_connection_creation_time_iso_8601_ns`	Time when the connection was first observed, in ISO 8601 format with nanosecond precision.
`tcp_connection_overall_duration_ns`	Total connection duration, in nanoseconds.
`tcp_average_bytes_in`	Average number of bytes received per packet.
`tcp_average_bytes_out`	Average number of bytes sent per packet.

8.5.12.7 Process Data

Object: `process_details`

Attribute	Description
`process_id`	Unique process identifier.
`process_executable_path`	Full path of the executable file.
`process_name`	Name of the executable file.
`process_self_exec_id`	Thread-group change indicator that increments on `exec` calls.
`process_parent_process_id`	Identifier of the parent process.
`process_cpu_clock_cycles`	Number of CPU clock cycles consumed by the process.
`process_real_group_id`	Real group ID of the process owner.
`process_real_user_id`	Real user ID of the process owner.
`process_effective_group_id`	Effective group ID of the process owner.
`process_effective_user_id`	Effective user ID of the process owner.

Attribute	Description
process_command_line_arguments	Command-line arguments used to start the process.
process_creation_time_iso_8601_ns	Process creation time, in ISO 8601 format with nanosecond precision.
process_state	Current process state.
process_pid_namespace	Process ID namespace.
process_mount_points_namespace	Mount namespace.
process_network_namespace	Network namespace.
process_hash_sha256	SHA-256 hash of the executable file.
process_hash_sha1	SHA-1 hash of the executable file.
process_hash_md5	MD5 hash of the executable file.
process_file_size_bytes	Executable file size, in bytes.
process_current_working_directory	Working directory from which the process was started.
process_container_id	Container identifier when the process belongs to a container.

8.5.12.8 Process Memory Data

Object: process_memory_details

Attribute	Description
process_id	Identifier of the associated process.
virtual_memory_area_start_address	Start address of the virtual memory area.
virtual_memory_area_end_address	End address of the virtual memory area.
memory_permissions	Permissions assigned to the virtual memory area.
is_main_process_executable	Indicates whether the virtual memory area belongs to the process executable.
file_path	Full path of the file associated with the virtual memory area.
file_name	Name of the file associated with the virtual memory area.

8.5.12.9 Runtime Manifest Data

Object: process_attestation_details

Attribute	Description
elf_file_inode_number	Inode number of the ELF file.

Attribute	Description
`elf_file_name`	Name of the ELF file.
`elf_file_folder_path`	Directory containing the ELF file.
`elf_file_hash_sha256`	SHA-256 hash of the ELF file.
`elf_file_hash_sha1`	SHA-1 hash of the ELF file.
`elf_file_size_bytes`	ELF file size, in bytes.
`is_main_process_executable`	Indicates whether the file is the main executable for the process.
`elf_file_type`	ELF file type.
`elf_file_hash_md5`	MD5 hash of the ELF file.

8.5.12.9.1 Expected Behavior Data

Object: `expected_behavior`

The following fields are included with **Binary Executed Not as Intended**, **Foreign Binary Executed File Size Mismatch**, and **Foreign Library Loaded File Size Mismatch** alerts.

Attribute	Description
`process_hash_sha256`	Expected SHA-256 hash of the executable or library.
`process_command_line_arguments`	Expected command-line arguments.
`process_folder_path`	Expected directory of the executable or library.
`process_name`	Expected name of the executable or library.
`process_file_size_bytes`	Expected file size, in bytes.

8.5.12.10 Thread Data

Object: `thread_details`

Attribute	Description
`thread_id`	Unique thread identifier.
`thread_self_exec_id`	Thread-group change indicator that increments on `exec` calls.
`thread_exit_state`	Thread exit state.

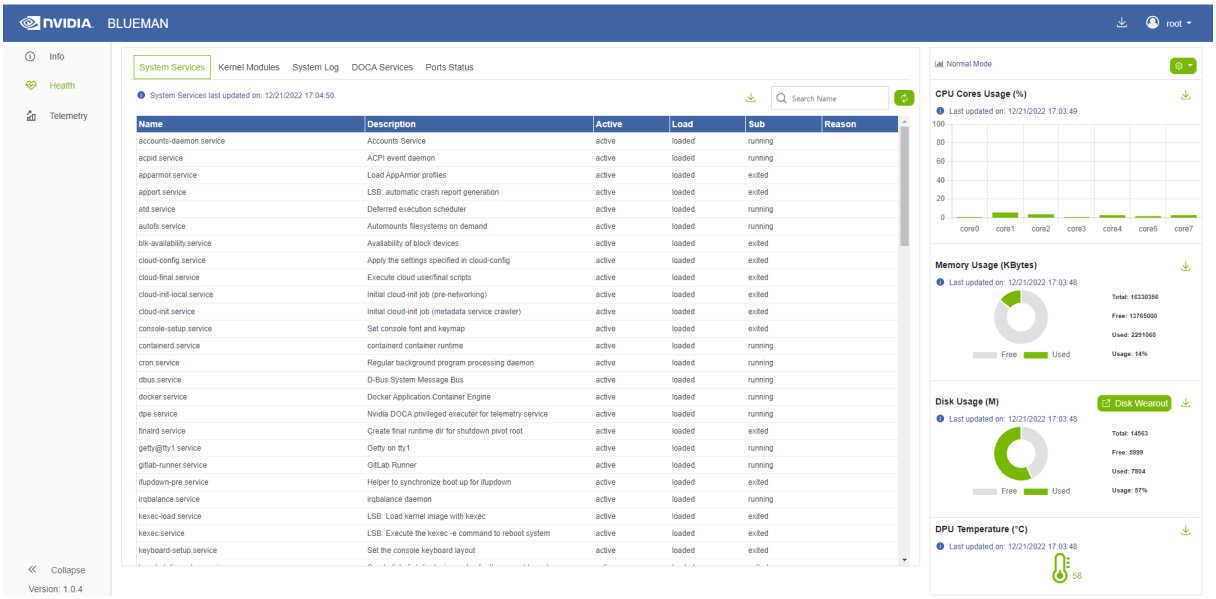
8.6 DOCA BlueMan Service Guide

This guide provides instructions on how to use the DOCA BlueMan service on top of NVIDIA® BlueField® DPU.

8.6.1 Introduction

DOCA BlueMan runs in the DPU as a standalone web dashboard and consolidates all the basic information, health, and telemetry counters into a single interface.

All the information that BlueMan provides is gathered from the DOCA Telemetry Service (DTS), starting from DTS version 1.11.1-doca1.5.1.



8.6.2 Requirements

- BlueField image version 3.9.3.1 or higher
- DTS and the DOCA Privileged Executer (DPE) daemon must be up and running

8.6.2.1 Verifying DTS Status

All the information that BlueMan provides is gathered from DTS.

Verify that the state of the DTS pod is **ready**:

```
$ crictl pods --name doca-telemetry-service
```

Verify that the state of the DTS container is **running**:

```
$ crictl ps --name doca-telemetry-service
```

8.6.2.2 Verifying DPE Status

All the information that DTS gathers for BlueMan is from the the DPE daemon.

Verify that the DPE daemon is **active**:

```
$ systemctl is-active dpe.service
active
```

If the daemon is inactive, activate it by starting the **dpe.service**:

```
$ systemctl start dpe.service
```

8.6.3 Service Deployment

For information about the deployment of DOCA containers on top of the BlueField DPU, refer to the NVIDIA DOCA Container Deployment Guide.

8.6.3.1 DOCA Service on NGC

BlueMan is available on NGC, NVIDIA's container catalog. Service-specific configuration steps and deployment instructions can be found under the service's container page.

8.6.3.2 Default Deployment – BlueField BSP

BlueMan service is located under **/opt/mellanox/doca/services/blueman**/.

The following is a list of the files under the BlueMan directory:

```
doca_blueman_fe_service_<version>-doca<version>_arm64.tar
doca_blueman_conv_service_<version>-doca<version>_arm64.tar
doca_blueman_standalone.yaml
bring_up_doca_blueman_service.sh
```

8.6.3.2.1 Enabling BlueMan Service

8.6.3.2.1.1 Using Script

Run **bring_up_doca_blueman_service.sh**:

```
$ chmod +x /opt/mellanox/doca/services/blueman/bring_up_doca_blueman_service.sh
$ /opt/mellanox/doca/services/blueman/bring_up_doca_blueman_service.sh
```

8.6.3.2.1.2 Manual Procedure

1. Import images to crictl images:

```
$ cd /opt/mellanox/doca/services/blueman/
$ ctr --namespace k8s.io image import doca_blueman_fe_service_<version>-doca<version>_arm64.tar
```

```
$ ctr --namespace k8s.io image import doca_blueman_conv_service_<version>-doca<version>_arm64.tar
```

2. Verify that the DPE daemon is active:

```
$ systemctl is-active dpe.service
active
```

If the daemon is inactive, activate it by starting the **dpe.service**:

```
$ systemctl start dpe.service
```

3. Copy **blueman_standalone.yaml** to **/etc/kubelet.d/**:

```
$ cp doca_blueman_standalone.yaml /etc/kubelet.d/
```

8.6.3.3 Verifying Deployment Success

1. Verify that the DPE daemon is active:

```
$ systemctl is-active dpe.service
```

2. Verify that the state of the DTS container is **running**:

```
$ crictl ps --name doca-telemetry-service
```

3. Verify that the state of the BlueMan service container is **running**:

```
$ crictl ps --name doca-blueman-fe
$ crictl ps --name doca-blueman-conv
```

Configuration

The configuration of the BlueMan back end is located under **/opt/mellanox/doca/services/telemetry/config/blueman_config.ini**. Users can interact with the **blueman_config.ini** file which contains the default range values of the Pass, Warning, and Failed categories which are used in the health page. Changing these values gets reflected in the BlueMan webpage within 60 seconds.

Example of **blueman_config.ini**:

```
;Health Cpu usages Pass, warning, Failed
[Health:CPU_Usages:Pass]
range = 0,80
[Health:CPU_Usages:Warning]
range = 80,90
[Health:CPU_Usages:Failed]
range = 90,100
```

8.6.4 Collected Data

- Info
  - General info – OS name, kernel, part number, serial number, DOCA version, driver, board ID, etc.
  - Installed packages – list of all installed packages on the DPU including their version
  - CPU info – vendor, cores, model, etc.

        - FW info – all the mlxconfig parameters with default/current/next boot data
        - DPU operation mode
- Health
        - System service
        - Kernel modules
        - Dmesg
        - DOCA services
        - Port status of the PF and OOB
        - Core usage and processes running on each core
        - Memory usage
        - Disk usage
        - Temperature
- Telemetry – all telemetry counters that come from DTS according to the enabled providers displayed on tables
        - Users have the ability to build graphs of specific counters

8.6.5 Connecting to BlueMan Web Interface

To log into BlueMan, enter the IP address of the DPU's OOB interface (**http://<DPU_OOB_IP>**) to a web browser located in the same network as the DPU.

The login credentials to use are the same pair used for the SSH connection to the DPU.

8.6.6 Troubleshooting

For general troubleshooting, refer to the DOCA Troubleshooting.

For container-related troubleshooting, refer to the "Troubleshooting" section in the NVIDIA DOCA Container Deployment Guide.

The following are additional troubleshooting tips for DOCA BlueMan:

- The following error message in the login page signifies a failure to connect to the DPE daemon: "The service is currently unavailable. Please check server up and running."
  a. Restart the DPE daemon:

  ```
  $ systemctl restart dpe.service
  ```

  b. Verify that DTS is up and running by following the instructions in section "Verifying DTS Status".
- If the message "Invalid Credentials" appears in the login page, verify that the username and password are the same ones used to SSH to the DPU.
- If all of the above is configured as expected and there is still some failure to log in, it is recommended to check if there are any firewall rules that block the connection.
- For other issues, check the **/var/log/syslog** and **/var/log/doca/telemetry/blueman_service.log** log file.

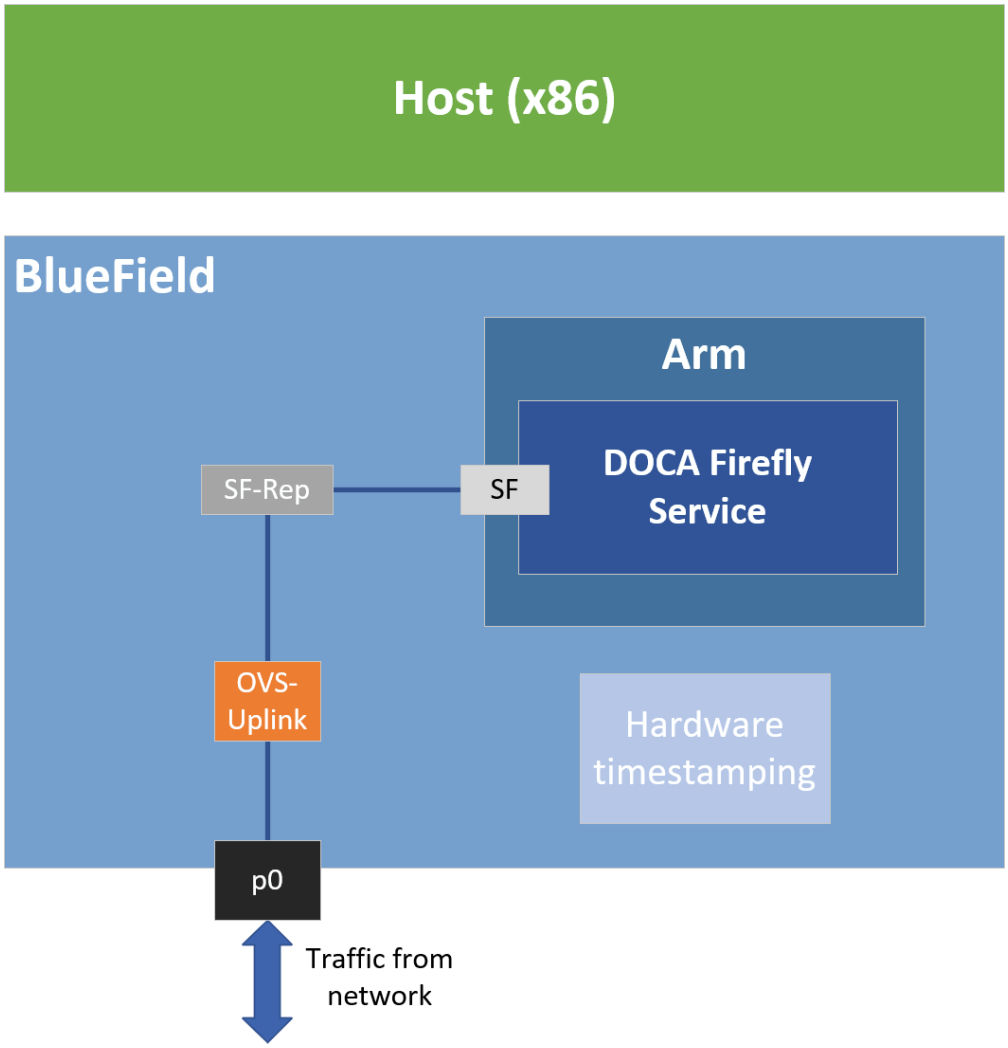
8.7 DOCA Firefly Service Guide

This guide provides instructions on how to use the DOCA Firefly service container on top of NVIDIA® BlueField® DPU.

8.7.1 Introduction

DOCA Firefly Service provides precision time protocol (PTP) based time syncing services to the BlueField DPU.

PTP is a protocol used to synchronize clocks in a network. When used in conjunction with hardware support, PTP is capable of sub-microsecond accuracy, which is far better than is what is normally obtainable with network time protocol (NTP). PTP support is divided between the kernel and user space. The ptp4l program implements the PTP boundary clock and ordinary clock. With hardware time stamping, it is used to synchronize the PTP hardware clock to the master clock.



8.7.2 Requirements

Some of the features provided by Firefly require specific BlueField DPU hardware capabilities:

- PTP – Supported by all BlueField DPUs
- PPS – Requires BlueField DPU with PPS capabilities
- SyncE - Requires converged card BlueField DPUs

Failure to run PPS due to missing hardware support will be noted in the service's output. However, the service will continue to run the timing services it can provide on the provided hardware.

8.7.2.1 Firmware Version

Firmware version must be 24.34.1002 or higher.

8.7.2.2 BlueField BSP Version

Supported BlueField image versions are 3.9.0 and higher.

8.7.2.3 Embedded Mode

8.7.2.3.1 Configuring Firmware Settings on DPU for Embedded Mode

1. Set the DPU to embedded mode (default mode):

```
sudo mlxconfig -y -d 03:00.0 s INTERNAL_CPU_MODEL=1
```

2. Enable the real time clock (RTC):

```
sudo mlxconfig -d 03:00.0 set REAL_TIME_CLOCK_ENABLE=1
```

3. Graceful shutdown and power cycle the DPU to apply the configuration.
4. You may check the DPU mode using the following command:

```
sudo mlxconfig -d 03:00.0 q | grep INTERNAL_CPU_MODEL
# Example output
         INTERNAL_CPU_MODEL                  EMBEDDED_CPU(1)
```

8.7.2.3.2 Ensuring OVS Hardware Offload

DOCA Firefly requires that hardware offload is activated in Open vSwitch (OVS). This is enabled by default as part of the BFB image installed on the DPU.

To verify the hardware offload configuration in OVS:

```
sudo ovs-vsctl get Open_vSwitch . other_config | grep hw-offload
# Example output
        {hw-offload="true"}
```

If inactive:

1. Activate hardware offloading by running:

```
sudo ovs-vsctl set Open_vSwitch . other_config:hw-offload=true;
```

2. Restart the OVS service:

```
sudo /etc/init.d/openvswitch-switch restart
```

3. Graceful shutdown and power cycle the DPU to apply the configuration.

8.7.2.3.3 Helper Scripts

Firefly's deployment contains a script to help with the configuration steps required for the network interface in embedded mode:

- **scripts/doca_firefly/<firefly-version>/prepare_for_embedded_mode.sh**

- **scripts/doca_firefly/<firefly-version>/set_new_sf.sh**

The latest DOCA Firefly version is **1.7.0**.

Both scripts are included as part of DOCA's container resource which can be downloaded according to the instructions in the NVIDIA DOCA Container Deployment Guide. For more information about the structure of the DOCA container resource, refer to section "Structure of NGC Resource" in the deployment guide.

Due to technical limitations of the NGC resource, both scripts are provided without execute (+x) permissions. This could be resolved by running the following command:

```
chmod +x scripts/doca_firefly/<firefly-version>/*.sh
```

8.7.2.3.3.1 prepare_for_embedded_mode.sh

This script automates all the steps mentioned in section "Setting Up Network Interfaces for Embedded Mode" and configures a freshly installed BFB image to the settings required by DOCA Firefly.

Notes:

- The script deletes all previous OVS settings and creates a single OVS bridge that matches the definitions in section "Setting Up Network Interfaces for Embedded Mode"
- The script should only be run once when connecting to the DPU for the first time or after a power cycle
- The only manual step required after using this script is configuring the IP address for the created network interface (step 5 in section "Setting Up Network Interfaces for Embedded Mode")
- The script automatically uses port 0 (**p0**). Configurations for port 1 should be done manually based on the commands listed in sections " set_new_sf.sh" and " Setting Up Network Interfaces for DPU Mode".

Script arguments:

- SF number (checks if already exists)

Examples:

- Prepare OVS settings using an SF indexed 4:

```
chmod +x ./*.sh
./prepare_for_embedded_mode.sh 4
```

The script makes use of **set_new_sf.sh** as a helper script.

8.7.2.3.3.2 set_new_sf.sh

Creates a new trusted SF and marks it as "trusted".

Script arguments:

- PCIe address
- SF number (checks if already exists)
- MAC address (if absent, a random address is generated)

Examples:

- Create SF with number "4" over port 0 of the DPU:

```
./set_new_sf.sh 0000:03:00.0 4
```

- Create SF with number "5" over port 0 of the DPU and a specific MAC address:

```
./set_new_sf.sh 0000:03:00.0 5 aa:bb:cc:dd:ee:ff
```

- Create SF with number "4" over port 1 of the DPU:

```
./set_new_sf.sh 0000:03:00.1 4
```

The first two examples should work out of the box for a BlueField-2 device and create SF4 and SF5 respectively.

8.7.2.3.4 Setting Up Network Interfaces for DPU Mode

1. Create a trusted SF to be used by the service according to the Scalable Function Setup Guide.

   > ⚠ The following instructions assume that the SF has been created using index 4.

2. Create the required OVS setting as is shown in the architecture diagram:

```
$ sudo ovs-vsctl add-br uplink
$ sudo ovs-vsctl add-port uplink p0
$ sudo ovs-vsctl add-port uplink en3f0pf0sf4
# This port is needed to ensure we have traffic host<->network as well
$ sudo ovs-vsctl add-port uplink pf0hpf
```

3. Verify the OVS settings:

```
sudo ovs-vsctl show
    Bridge uplink
        Port pf0hpf
            Interface pf0hpf
        Port en3f0pf0sf4
            Interface en3f0pf0sf4
        Port p0
            Interface p0
        Port uplink
            Interface uplink
                type: internal
```

4. Enable TX timestamping on the SF interface (not the representor):

```
# tx port timestamp offloading
sudo ethtool --set-priv-flags enp3s0f0s4 tx_port_ts on
```

5. Enable the interface and set an IP address for it:

```
# configure ip for the interface:
```

```
sudo ifconfig enp3s0f0s4 <ip-addr> up
```

6. Configure OVS to support TX timestamping over this SF and multicast traffic in general:

```
# Multicast-related definitions
$ sudo ovs-vsctl set Bridge uplink mcast_snooping_enable=true
$ sudo ovs-vsctl set Bridge uplink other_config:mcast-snooping-disable-flood-unregistered=true
$ sudo ovs-vsctl set Port p0 other_config:mcast-snooping-flood=true
$ sudo ovs-vsctl set Port p0 other_config:mcast-snooping-flood-reports=true
# PTP-related definitions
$ sudo ovs-ofctl add-flow uplink in_port=en3f0pf0sf4,udp,tp_src=319,actions=output:p0
$ sudo ovs-ofctl add-flow uplink in_port=p0,udp,tp_src=319,actions=output:en3f0pf0sf4
$ sudo ovs-ofctl add-flow uplink in_port=en3f0pf0sf4,udp,tp_src=320,actions=output:p0
$ sudo ovs-ofctl add-flow uplink in_port=p0,udp,tp_src=320,actions=output:en3f0pf0sf4
```

If your OVS bridge uses a name other than **uplink**, make sure that the used name is reflected in the **ovs-vsctl** and **ovs-ofctl** commands. For instance:

```
$ sudo ovs-vsctl set Bridge <bridge-name> mcast_snooping_enable=true
```

8.7.2.4 Separated Mode

8.7.2.4.1 Configuring Firmware Settings on DPU for Separated Mode

1. Set the BlueField mode of operation to "Separated":

```
sudo mlxconfig -y -d 03:00.0 s INTERNAL_CPU_MODEL=0
```

2. Enable RTC:

```
sudo mlxconfig -d 03:00.0 set REAL_TIME_CLOCK_ENABLE=1
```

3. Graceful shutdown and power cycle the DPU to apply the configuration.
4. You may check the BlueField's operation mode using the following command:

```
sudo mlxconfig -d 03:00.0 q | grep INTERNAL_CPU_MODEL
# Example output
         INTERNAL_CPU_MODEL                  SEPARATED_HOST(0)
```

8.7.2.4.2 Setting Up Network Interfaces for Separated Mode

1. Make sure that that **p0** is not connected to an OVS bridge:

```
sudo ovs-vsctl show
```

2. Enable TX timestamping on the **p0** interface:

```
# TX port timestamp offloading (assuming PTP interface is p0)
sudo ethtool --set-priv-flags p0 tx_port_ts on
```

3. Enable the interface and set an IP address for it:

```
# Configure IP for the interface
sudo ifconfig p0 <ip-addr> up
```

8.7.2.5 Host-based Deployment

Host-based deployment requires the same configuration described under section " Separated Mode".

8.7.3 Service Deployment

8.7.3.1 DPU Deployment

For information about the deployment of DOCA containers on top of the BlueField DPU, refer to NVIDIA DOCA Container Deployment Guide.

Service-specific configuration steps and deployment instructions can be found under the service's container page.

> ⚠ DOCA Firefly can also be deployed on DPUs not connected to the Internet. For instructions, refer to the relevant section in the NVIDIA DOCA Container Deployment Guide.

8.7.3.2 Host Deployment

DOCA Firefly has a version adapted for host-based deployments. For more information about the deployment of DOCA containers on top of a host, refer to the NVIDIA BlueField DPU Container Deployment Guide.

The following is the docker command for deploying DOCA Firefly on the host:

```
sudo docker run --privileged --net=host -v /var/log/doca/firefly:/var/log/firefly -v /etc/firefly:/etc/firefly -e
PTP_INTERFACE='eth2' -it nvcr.io/nvidia/doca/doca_firefly:1.7.0-doca3.0.0-host /entrypoint.sh
```

Where:

- Additional YAML configs may be passed as environment variables as additional `-e` key-value pairs as done with `PTP_INTERFACE` above
- The exact container tag should be the desired tag as chosen on DOCA Firefly's NGC page

8.7.4 Configuration

All modules within the service have configuration files that allow customizing various settings, both general and PTP-related.

8.7.4.1 Built-In Config File

Each profile has its own base PTP configuration file for `ptp4l`. For example, the Media profile PTP configuration file is `ptp4l-media.conf`.

The built-in PTP configuration files can be found in section "PTP Profile Default Config Files". For ease-of-use, those files are provided as part of DOCA's container resource as downloaded from NGC and are placed under Firefly's **configs** directory (**scripts/doca_firefly/<firefly version>/configs**).

> ⚠ When using a built-in configuration file, Firefly uses the files as stored within the container itself in the **/etc/linuxptp** directory. The configuration files included in the NGC resource are only provided for ease of access. Modifying them does not impact the configuration used in practice by the container. Instead, updates to the configuration should be done as described in the following sections.

8.7.4.2 Custom Config File

Instead of using a profile's base config file, users can create a file of their own, for each of the modules.

To set a custom config file, users should locate their config file in the directory **/etc/firefly** and set the config file name in DOCA Firefly's YAML file.

For example, to set a custom **linuxptp** config file, the user can set the parameter **PTP_CONFIG_FILE** in the YAML file:

```
- name: PTP_CONFIG_FILE
  value: my_custom_ptp.conf
```

In this example, **my_custom_ptp.conf** should be placed at **/etc/firefly/my_custom_ptp.conf** .

> ⚠ A config file must not define values for the UDS-related ports (**/var/run/ptp4l** and **/var/run/ptp4lro**), as those will impact internal container behavior. Such settings will prompt a warning and will be ignored when preparing the finalized configuration (See more in the next sections).

8.7.4.3 Overriding Specific Config File Parameters

Instead of replacing the entire config file, users may opt to override specific parameters. This can be done using the following variable syntax in the YAML file:
CONF_<TYPE>_<SECTION>_<PARAMETER_NAME> .

- **TYPE** – either **PTP** , **MONITOR** , **PHC2SYS** , SYNCE, or **SERVO**
- **SECTION** – the section in the config file that the parameter should be placed in

  > ⚠ If the specified section does not already exist in the config file, a new section is created unless it refers to a PTP network interface that has not been included in the **PTP_INTERFACE** YAML field.

- **PARAMETER_NAME** – the config parameter name as should be placed in the config file

> ⚠ If the parameter name already exists in the config file, then the value is changed according to the value provided in the `.yaml` file. If the parameter name does not already exist in the config file, then it is added.

For example, the following variable in the YAML file definition changes the value of the parameter `priority1` under section `global` in the PTP config file to `64`.

```
- name: CONF_PTP_global_priority1
  value: "64"
```

> ⚠ Configuring `unicast_master_table` through the YAML file is not supported due to the structure of the table (i.e., multiple entries sharing the same key).

8.7.4.4 Ensuring and Debugging Correctness of Config Files

The previous sections describe 2 layers for the configuration file definitions:

- Basic configuration file – either a built-in config file or a custom config file
- Adding/overriding values to/from the YAML file

In practice, there are slightly more layers in place, and the precedence is as follows (presented in increasing order):

- Default configuration values of the PTP program (ptp4l for instance) – holds values of all available configuration options
- Your chosen configuration file – contains a subset of options
- Definitions from the YAML file – narrower subset
- Firefly mandatory values

When combining the supplied configuration file with the definitions from the YAML file, Firefly goes over those definitions and checks them against a predefined set of configuration options:

- Warning only – warns if a certain value leads to known issues in a supported deployment scenario
- Override – container-internal definitions that should not be set by the user and will be overridden by Firefly

Suitable log messages are provided in either case:

```
# Example for a warning
2023-01-31 11:55:13 - Firefly - Config - INFO    - Missing explicit definition "fault_reset_interval", verifying
default value instead: "4"
2023-01-31 11:55:13 - Firefly - Config - WARNING - Value "4" for definition "fault_reset_interval" will be invalid
in Embedded Mode, expected a value lesser or equal to "1"
2023-01-31 11:55:13 - Firefly - Config - WARNING - Continuing with invalid value
# Example for an override
2023-01-31 11:21:00 - Firefly - Config - WARNING - Invalid value "/var/run/ptp4l2" for definition "uds_address",
expected "/var/run/ptp4l"
2023-01-31 11:21:00 - Firefly - Config - INFO    - Setting definition "uds_address" value to the following: "/var/
run/ptp4l"
```

At the end of this process, an updated configuration file is generated by Firefly to be used later by the various time providers. To avoid accidental modification of a user-supplied configuration file or permission issues, the finalized file is generated within the container under the **/tmp** directory.

For instance, if using a custom configuration file named **my_custom_ptp.conf** under the **/etc/firefly** directory on the DPU, the updated file will reside within the container at the following path: **/tmp/my_custom_ptp.conf**.

For troubleshooting possible issues with the configuration file, one can do one of the following:

- Connect to the container directly as is explained in the debugging finalized configuration file bullet under " Troubleshooting".
- Map the container's **/tmp** directory to the DPU using the built-in support in the YAML file:
  - Before the change:

```
    # Uncomment when debugging the finalized configuration files used - Part #1
    #- name: debug-firefly-volume
    #  hostPath:
    #    path: /tmp/firefly
    #    type: DirectoryOrCreate
  containers:
    ...
      volumeMounts:
      - name: logs-firefly-volume
        mountPath: /var/log/firefly
      - name: conf-firefly-volume
        mountPath: /etc/firefly
      # Uncomment when debugging the finalized configuration files used - Part #2
      #- name: debug-firefly-volume
      #  mountPath: /tmp
```

  - After the change:

```
    # Uncomment when debugging the finalized configuration files used - Part #1
    - name: debug-firefly-volume
      hostPath:
        path: /tmp/firefly
        type: DirectoryOrCreate
  containers:
    ...
      volumeMounts:
      - name: logs-firefly-volume
        mountPath: /var/log/firefly
      - name: conf-firefly-volume
        mountPath: /etc/firefly
      # Uncomment when debugging the finalized configuration files used - Part #2
      - name: debug-firefly-volume
        mountPath: /tmp
```

The finalized configuration file keeps the sections and config options in the same order as they appear in the original file, yet the file is stripped from spare new lines or comment lines. This should be taken into considerations when directly accessing it during a debugging session.

8.7.5 Description

8.7.5.1 Providers

DOCA Firefly Service uses the following third-party providers to provide time syncing services:

- Linuxptp - Version v4.2
  - **PTP** – PTP service, provided by the PTP4L program
  - **PHC2SYS** – OS time calibration, provided by the PHC2SYS program

- Testptp
  - **PPS** - PPS settings service

In addition, DOCA Firefly Service also makes use of the following NVIDIA modules:

- SyncE
  - **SYNCE** – Synchronous Ethernet Deamon (**synced**)
- Firefly
  - **MONITOR** - Firefly PTP Monitor
- Firefly
  - **SERVO** - Firefly PTP Servo

Each of the providers can be enabled, disabled, or set to use the setting defined by the configuration profile:

- YAML setting – **<provider name>_STATE**
- Supported values – **enable** , **disable** , **defined_by_profile**

For the default profile settings per provider, refer to the table under section " Profiles".

An example YAML setting for specifically disabling the **phc2sys** provider is the following:

```
- name: PHC2SYS_STATE
  value: "disable"
```

The **defined_by_profile** setting is only available for well-defined profiles. As such, it cannot be used when the **custom** profile is selected. For more information about the profile settings, refer to the table under section " Profiles".

8.7.5.2 Profiles

DOCA Firefly Service includes profiles which represent common use cases for the Firefly service that provide a different default configuration per profile:

	Default	Media	Telco (L2)	Custom
Purpose	Any user that requires PTP	Media productions	Telco networks	Custom configuration for a dedicated user scenario
PTP	Enabled	Enabled	Enabled	No default. Enable/disable should be set by the user.
PTP profile	PTP default profile	SMPTE 2059-2	G.8275.1	Set by the user
PTP Client/ Server	Both	Client-only	Both	Set by the user
PHC2SYS	Enabled	Enabled	Enabled	No default. Enable/disable should be set by the user.
PPS (in/ out)	Enabled	Enabled	Enabled	No default. Enable/disable should be set by the user.

	Default	Media	Telco (L2)	Custom
PTP Monitor	Disabled	Disabled	Disabled	No default. Enable/disable should be set by the user.
SyncE	Disabled	Disabled	Enabled	No default. Enable/disable should be set by the user.
Servo	Disabled	Disabled	Disabled	No default. Enable/disable should be set by the user.

We don't have a way to export this macro.

8.7.5.3 Outputs

8.7.5.3.1 Container Output

While running, the full output of the DOCA Firefly Service container can be viewed using the following command:

```
sudo crictl logs <CONTAINER-ID>
```

Where **CONTANIER-ID** can be retrieved using the following command:

```
sudo crictl ps
```

For example, in the following output, the container ID is **8f368b98d025b**.

```
$ sudo crictl ps
CONTAINER           IMAGE               CREATED             STATE               NAME                ATTEMPT
    POD ID              POD
8f368b98d025b       289809f312b4c       2 seconds ago       Running             doca-firefly        0
    5af59511b4be4       doca-firefly-some-computer-name
```

The output of the container depends on the services supported by the hardware and enabled by configuration and the selected profile. However, note that any of the configurations runs PTP, so when DOCA FireFly is running successfully expect to see the line "**Running ptp4l**".

The following is an example of the expected container output when running the default profile on a DPU that supports PPS:

```
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Starting DOCA Firefly - Version 1.4.0
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Selected features:
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] PTP     - Enabled - ptp4l will be used
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] MONITOR - Enabled - PTP Monitor will be used
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] PHC2SYS - Enabled - phc2sys will be used
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [-] SyncE   - Disabled
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [-] SERVO   - Disabled
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] PPS     - Enabled - testptp will be used (if supported by
hardware)
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Going to analyze the configuration files
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Requested the following PTP interface: p0
2023-09-07 14:04:23 - Firefly
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Starting PPS configuration
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] PPS is supported by hardware
2023-09-07 14:04:23 - Firefly - Init    - INFO     - set pin function okay
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] PPS in - Activated
2023-09-07 14:04:23 - Firefly - Init    - INFO     - set pin function okay
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] PPS out - Activated
2023-09-07 14:04:23 - Firefly - Init    - INFO     - name mlx5_pps0 index 0 func 1 chan 0
```

```
2023-09-07 14:04:23 - Firefly - Init    - INFO     - name mlx5_pps1 index 1 func 2 chan 0
2023-09-07 14:04:23 - Firefly - Init    - INFO     - periodic output request okay
2023-09-07 14:04:23 - Firefly
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Running ptp4l
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Running Firefly PTP Monitor
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Running phc2sys
```

The following is an example of the expected container output when running the default profile on a DPU that does not support PPS:

```
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Starting DOCA Firefly - Version 1.3.0
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Selected features:
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] PTP     - Enabled - ptp4l will be used
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] MONITOR - Enabled - PTP Monitor will be used
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] PHC2SYS - Enabled - phc2sys will be used
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [-] SyncE   - Disabled
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [-] SERVO   - Disabled
2023-09-07 14:04:23 - Firefly - Init    - INFO     - [+] PPS     - Enabled - testptp will be used (if supported by
hardware)
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Going to analyze the configuration files
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Requested the following PTP interface: p0
2023-09-07 14:04:23 - Firefly
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Starting PPS configuration
2023-09-07 14:04:23 - Firefly - Init    - WARNING  - [-] PPS capability is missing, seems that the card doesn't
support PPS
2023-09-07 14:04:23 - Firefly - Init    - INFO     - capabilities:
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   50000000 maximum frequency adjustment (ppb)
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   0 programmable alarms
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   0 external time stamp channels
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   0 programmable periodic signals
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   0 pulse per second
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   0 programmable pins
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   0 cross timestamping
2023-09-07 14:04:23 - Firefly
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Running ptp4l
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Running Firefly PTP Monitor
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Running phc2sys
```

8.7.5.3.2 Firefly Output

On top of the container's log, Firefly defines an additional, non-volatile log that can be found in **/var/log/doca/firefly/firefly.log**.

This file contains the same output described in section " Container Output" and is useful for debugging deployment errors should the container stop its execution.

> ⚠ To avoid disk space issues, the **/var/log/doca/firefly/firefly.log** file only contains the log from Firefly's initialization, and not the logs of the rest of the modules (ptp4l, phc2sys, etc.) or that of the PTP monitor. The latter is still included in the container log and can be inspected using the command **sudo crictl logs <CONTAINER-ID>**.

8.7.5.3.3 ptp4l Output

The ptp4l output can be found in the file **/var/log/doca/firefly/ptp4l.log**.

Example output:

```
ptp4l[192710.691]: rms 1 max 1 freq -114506 +/- 0 delay -15 +/- 0
ptp4l[192712.692]: rms 6 max 9 freq -114501 +/- 3 delay -15 +/- 0
ptp4l[192714.692]: rms 7 max 9 freq -114511 +/- 3 delay -13 +/- 0
ptp4l[192716.692]: rms 5 max 7 freq -114502 +/- 1 delay -13 +/- 0
ptp4l[192718.693]: rms 4 max 6 freq -114509 +/- 2 delay -13 +/- 0
ptp4l[192720.693]: rms 3 max 3 freq -114506 +/- 2 delay -13 +/- 0
ptp4l[192722.694]: rms 4 max 6 freq -114510 +/- 3 delay -12 +/- 0
ptp4l[192724.694]: rms 5 max 7 freq -114510 +/- 5 delay -12 +/- 1
ptp4l[192726.695]: rms 4 max 5 freq -114508 +/- 3 delay -11 +/- 0
ptp4l[192728.695]: rms 6 max 9 freq -114504 +/- 4 delay -11 +/- 0
```

8.7.5.3.4 phc2sys Output

The phc2sys output can be found in the file **/var/log/doca/firefly/phc2sys.log**.

Example output:

```
phc2sys[1873325.928]: reconfiguring after port state change
phc2sys[1873325.928]: selecting CLOCK_REALTIME for synchronization
phc2sys[1873325.928]: selecting enp3s0f0s4 as the master clock
phc2sys[1873325.928]: CLOCK_REALTIME phc offset      1378 s2 freq -165051 delay    255
phc2sys[1873326.928]: CLOCK_REALTIME phc offset      1378 s2 freq -163673 delay    240
phc2sys[1873327.928]: port 62b785.fffe.0c9369-1 changed state
phc2sys[1873327.929]: CLOCK_REALTIME phc offset        14 s2 freq -164624 delay    255
phc2sys[1873328.936]: CLOCK_REALTIME phc offset        89 s2 freq -164545 delay    240
```

8.7.5.3.5 SyncE Output

The SyncE output can be found in the file **/var/log/doca/firefly/synced.log**.

Example output:

```
INFO     [05/09/2023 05:11:01.493414]: SyncE Group #0: is in TRACKING holdover acquired mode on p0, frequency_diff:
0 (ppb)
INFO     [05/09/2023 05:11:02.502963]: SyncE Group #0: is in TRACKING holdover acquired mode on p0, frequency_diff:
-113 (ppb)
INFO     [05/09/2023 05:11:03.512491]: SyncE Group #0: is in TRACKING holdover acquired mode on p0, frequency_diff:
37 (ppb)
```

The verbosity of the output from the **SYNCE** module is limited by default. To set the output to be more verbose, set the **verbose** option to **1** (True).

Before:

```
# Example #4 - Overwrite the value of verbose in the [global] section of the SyncE configuration file.
#- name: CONF_SYNCE_global_verbose
#  value: "1"
```

After:

```
# Example #4 - Overwrite the value of verbose in the [global] section of the SyncE configuration file.
- name: CONF_SYNCE_global_verbose
  value: "1"
```

8.7.5.3.6 Firefly Servo Output

The Firefly servo output can be found in the file **/var/log/doca/firefly/servo.log**.

Example output:

```
2024-03-18 09:04:22 - Firefly - SERVO  - INFO     - offset   +8 +/- 2  freq   -5.66 +/- 0.41  delay -48 +/- 2
2024-03-18 09:04:24 - Firefly - SERVO  - INFO     - offset   +4 +/- 2  freq   -6.35 +/- 0.36  delay -47 +/- 2
2024-03-18 09:04:26 - Firefly - SERVO  - INFO     - offset   +2 +/- 2  freq   -6.75 +/- 0.41  delay -47 +/- 1
2024-03-18 09:04:28 - Firefly - SERVO  - INFO     - offset   +0 +/- 2  freq   -6.97 +/- 0.35  delay -47 +/- 1
2024-03-18 09:04:30 - Firefly - SERVO  - INFO     - offset   +0 +/- 3  freq   -7.30 +/- 0.60  delay -47 +/- 1
2024-03-18 09:04:33 - Firefly - SERVO  - INFO     - offset   +1 +/- 2  freq   -6.93 +/- 0.41  delay -47 +/- 1
2024-03-18 09:04:35 - Firefly - SERVO  - INFO     - offset   +1 +/- 2  freq   -6.81 +/- 0.48  delay -47 +/- 1
2024-03-18 09:04:37 - Firefly - SERVO  - INFO     - offset   +2 +/- 2  freq   -6.76 +/- 0.52  delay -48 +/- 2
```

8.7.5.4 Tx Timestamping Support on DPU Mode

When the BlueField is operating in DPU mode, additional OVS configuration is required as mentioned in step 6 of section " Setting Up Network Interfaces for DPU Mode". This configuration achieves the following:

- Proper support for incoming/outgoing multicast traffic
- Enabling Tx timestamping

Firefly only gets the packet timestamping for outgoing PTP messages (Tx timestamping) when they are offloaded to the hardware. As such, when working with OVS, users must ensure this traffic flow is properly recognized and offloaded. If offloading does not take place, Firefly gets stuck in a fault loop while waiting to receive the Tx timestamp events:

```
ptp4l[2912.797]: timed out while polling for tx timestamp
ptp4l[2912.797]: increasing tx_timestamp_timeout may correct this issue, but it is likely caused by a driver bug
ptp4l[2912.797]: port 1 (enp3s0f0s4): send sync failed
ptp4l[2923.528]: timed out while polling for tx timestamp
ptp4l[2923.528]: increasing tx_timestamp_timeout may correct this issue, but it is likely caused by a driver bug
ptp4l[2923.528]: port 1 (enp3s0f0s4): send sync failed
```

The solution to this issue:

- Activation of hardware offloading in OVS
- OpenFlow rules that ensure OVS properly recognizes the traffic and offloads it to the hardware
- Modification to the **fault_reset_interval** configuration value to ensure timely recovery from the fault induced by the first packet being always treated by software (until the rule is offloaded to hardware). As such, Firefly requires that the **fault_reset_interval** value is 1 or less. Proper warnings are raised if an improper value is detected. The value is updated accordingly in the built-in profiles.

When these configurations are in order, Firefly includes a report for a single fault during boot, but recovers from it and continues as usual:

```
ptp4l[3715.687]: timed out while polling for tx timestamp
ptp4l[3715.687]: increasing tx_timestamp_timeout may correct this issue, but it is likely caused by a driver bug
ptp4l[3715.687]: port 1 (enp3s0f0s4): send delay request failed
```

8.7.5.4.1 Troubleshooting Tx Timestamp Issues

As explained earlier, there are several layers required to ensure Tx timestamping works as necessary by Firefly. The following is a list of commands to debug the state of each layer:

1. Inspect the OpenFlow rules:

   ```
   $ sudo ovs-ofctl dump-flows uplink
   cookie=0x0, duration=4075.576s, table=0, n_packets=2437, n_bytes=209582, udp,in_port=en3f0pf0sf4,tp_src=319
   actions=output:p0
   cookie=0x0, duration=4075.549s, table=0, n_packets=1216, n_bytes=109420, udp,in_port=p0,tp_src=319
   actions=output:en3f0pf0sf4
   cookie=0x0, duration=4075.521s, table=0, n_packets=13, n_bytes=1242, udp,in_port=en3f0pf0sf4,tp_src=320
   actions=output:p0
   cookie=0x0, duration=4074.604s, table=0, n_packets=3034, n_bytes=297376, udp,in_port=p0,tp_src=320
   actions=output:en3f0pf0sf4
   cookie=0x0, duration=4075.856s, table=0, n_packets=184, n_bytes=12901, priority=0 actions=NORMAL
   ```

2. Inspect hardware TC rules while DOCA Firefly is deployed (the rules age out after 10 seconds without traffic):

```
$ sudo tc -s -d filter show dev en3f0pf0sf4 egress
filter ingress protocol ip pref 4 flower chain 0
filter ingress protocol ip pref 4 flower chain 0 handle 0x1
  eth_type ipv4
  ip_proto udp
  src_port 320
  ip_flags nofrag
  in_hw in_hw_count 1
    action order 1: mirred (Egress Redirect to device p0) stolen
    index 3 ref 1 bind 1 installed 7 sec used 7 sec
    Action statistics:
    Sent 0 bytes 0 pkt (dropped 0, overlimits 0 requeues 0)
    backlog 0b 0p requeues 0
    cookie bec8bd6ede4e86341e9045a6edb58ca2
    no_percpu

filter ingress protocol ip pref 4 flower chain 0 handle 0x2
  eth_type ipv4
  ip_proto udp
  src_port 319
  ip_flags nofrag
  in_hw in_hw_count 1
    action order 1: mirred (Egress Redirect to device p0) stolen
    index 4 ref 1 bind 1 installed 6 sec used 6 sec
    Action statistics:
    Sent 0 bytes 0 pkt (dropped 0, overlimits 0 requeues 0)
    backlog 0b 0p requeues 0
    cookie c568d97efd400de98608fbbf86ccdf3c
    no_percpu
```

If no TC rules are present when Firefly is running, this usually indicates that hardware offloading is disabled at the OVS level, in which case it should be activated as explained under " Ensuring OVS Hardware Offload".

8.7.5.5 PTP

Firefly uses the **ptp4l** utility to handle the Precision Time Protocol (IEEE 1588).

Through the YAML file, users can configure the network interfaces used for the protocol:

```
# Network interfaces to be used (For multiple interfaces use a space (" ") separated list)
- name: PTP_INTERFACE
  # Set according to used interfaces on the local setup
  value: "p0"
```

Before the deployment of the container, users should configure this field to point at the desired network interface(s) configured in the previous steps.

8.7.5.6 PHC2SYS

Firefly uses the **phc2sys** utility to synchronize the OS's clock to the accurate time stamps received by **ptp4l**.

Through the YAML file, users can configure the command-line arguments used by the **phc2sys** program:

```
- name: PHC2SYS_ARGS
  value: "-a -r"
```

Firefly adds the following command-line arguments on top of the user-selected flags:

- Use of chosen configuration file (empty configuration file by default, or user-supplied file if specified in the YAML file)
- Redirection of output to a log file using the **-m** command line option

⚠ **phc2sys** must use the same **domainNumber** setting used by **ptp4l**. If the same **domainNumber** is not set by the user, Firefly does that automatically.

⚠ **phc2sys** is only able to accurately sync the clock of the hosting environment (usually the DPU, but may also be the host if deployed there) if other timing services, such as NTP, are disabled.

So, for instance, on Ubuntu 22.04, users must ensure that the NTP timing service is disabled by running:

```
systemctl stop systemd-timesyncd
```

8.7.5.7 SYNCE

Firefly uses the proprietary **synced** utility to implement the Synchronous Ethernet protocol, aimed at ensuring synchronization of the clock's frequency with the reference clock. Once achieved, both clocks are declared as "syntonized".

Through the YAML file, users can configure the network interfaces used for the protocol:

```
# Network interfaces to be used (For multiple interfaces use a space (" ") separated list)
- name: SYNCE_INTERFACE
  # Set according to used interfaces on the local setup
  value: "p0"
```

Before the deployment of the container, one should configure this field to point at the desired network interface(s) configured in the previous steps.

DOCA includes synced support for the **"dpll"** backend (default) which adds support for SFs and VFs. The **"dpll"** backend is the default backend used. If DOCA detects the system does not support it, it will automatically falls back to the **"mft"** backend.

⚠ In versions older than kernel 6.8 or BlueField Platform Software 2.8.0, only PFs are supported and only using the **"mft"** backend.

The backend option can be explicitly set using the YAML file by uncommenting the following lines:

Before

```
# Example #5 - Explicitly specify the used backend in the [global] section of the SyncE configuration file.
#- name: CONF_SYNCE_global_backend
#  # Options are "mft"/"dpll". If nothing is specified in YAML, "dpll" is taken as the default
#  value: "mft"
```

After

```
# Example #5 - Explicitly specify the used backend in the [global] section of the SyncE configuration file.
- name: CONF_SYNCE_global_backend
  # Options are "mft"/"dpll". If nothing is specified in YAML, "dpll" is taken as the default
  value: "mft"
```

The following is an example for the OVS commands required to route the SyncE-related traffic when using a SF on top of the **"dpll"** backend:

```
$ sudo ovs-ofctl add-flow uplink dl_dst=01:80:c2:00:00:02,in_port=en3f0pf0sf4,actions=p0
$ sudo ovs-ofctl add-flow uplink dl_dst=01:80:c2:00:00:02,in_port=p0,actions=en3f0pf0sf4
$ sudo ovs-ofctl add-flow uplink dl_dst=01:80:c2:00:00:02,actions=controller
```

This example uses the same OVS settings used earlier in the guide:

- **uplink** – bridge name
- **en3f0pf0sf4** – SF representor
- **p0** – PF interface we are working (port 0)

If your deployment uses different values make sure to adjust the above commands accordingly.

If the kernel version does not yet support this feature, and SF/VF are used, the following error is printed:

```
...
mlx5 DPLL kernel support appears to be missing
Falling back to MFT tools backend
...
```

If this error is shown, only PFs can be used, and **synced** falls back to using the **"mft"** backend.

8.7.5.8 PTP Monitor

PTP monitor periodically queries for various PTP-related information and prints it to the container's log.

The following is a sample output of this tool:

```
gmIdentity:                 48:B0:2D:FF:FE:5C:4D:24 (48b02d.fffe.5c4d24)
portIdentity:               48:B0:2D:FF:FE:5C:53:44 (48b02d.fffe.5c5344-1)
port_state:                 Active
domainNumber:               2
master_offset:              avg: 1   max:    -8  rms:    3
gmPresent:                  true
ptp_stable:                 Recovered
UtcOffset:                  37
timeTraceable:              0
frequencyTraceable:         0
grandmasterPriority1:       128
gmClockClass:               248
gmClockAccuracy:            0x6
grandmasterPriority2:       128
gmOffsetScaledLogVariance:  0xffff
ptp_time (TAI):             Thu Sep  7 11:22:50 2023
ptp_time (UTC adjusted):    Thu Sep  7 11:22:13 2023
system_time (UTC):          Thu Sep  7 11:22:13 2023
ptp_ports:                  48:B0:2D:FF:FE:5C:53:44 (48b02d.fffe.5c5344-1) - Slave
error_count:                1
last_err_time (UTC):        Thu Sep  7 09:55:48 2023
```

Among others, this monitoring provides the following information:

- Details about the Grandmaster the DPU is syncing with
- Current PTP timestamp
- Health information such as connection errors during execution and whether they have been recovered from

PTP monitoring is disabled by default and can be activated by replacing the **disable** value with the IP address for the monitor server to use:

```
- name: MONITOR_STATE
  Value: "<IP address for the monitoring server>"
```

Once activated, the information can be viewed from the container using the following command:

```
sudo crictl logs --tail=<MONITOR_LINE_NUMBER> <CONTAINER-ID>
```

MONITOR_LINE_NUMBER is equal to 20 lines + the number of PTP interfaces supplied via the **PTP_INTERFACE** field.

It is recommended to use the following **watch** command to actively monitor the PTP state:

```
sudo watch -n 1 crictl logs --tail=<MONITOR_LINE_NUMBER> <CONTAINER-ID>
```

When triaging deployment issues, additional logging information can be found in the monitor's developer logs at **/var/log/doca/firefly/firefly_monitor_dev.log**.

The monitoring feature connects to ptp4l's local UDS server to query the necessary information. This is why the configuration manager prevents users from modifying the **uds_address** and **uds_ro_address** fields used by ptp4l within the container.

8.7.5.8.1 Configuration

The PTP monitor supports configuration options which are passed through a dedicated configuration file like the rest of DOCA Firefly's modules. The built-in monitor configuration file can be found in the section " PTP Monitor". For ease of use, the file is also provided as part of DOCA's container resource as downloaded from NGC.

" Firefly Modules Configuration Options" contains a complete explanation of each of the configuration options alongside their default values.

To set a custom config file, users should locate their config file in the directory **/etc/firefly** and set the config file name in DOCA Firefly's YAML file.

```
- name: MONITOR_CONFIG_FILE
  value: my_custom_monitor.conf
```

In this example, **my_custom_monitor.conf** should be placed at **/etc/firefly/my_custom_monitor.conf**.

8.7.5.8.2 Time Representations (PTP Time vs System Time)

Under most deployment scenarios, the PTP time shown by the monitor is presented according to the International Atomic Time (TAI) standard, while the system time would most commonly use the Coordinated Universal Time (UTC). Due to the differences between these time representation models, the monitor provides 2 different time readings (each marked accordingly):

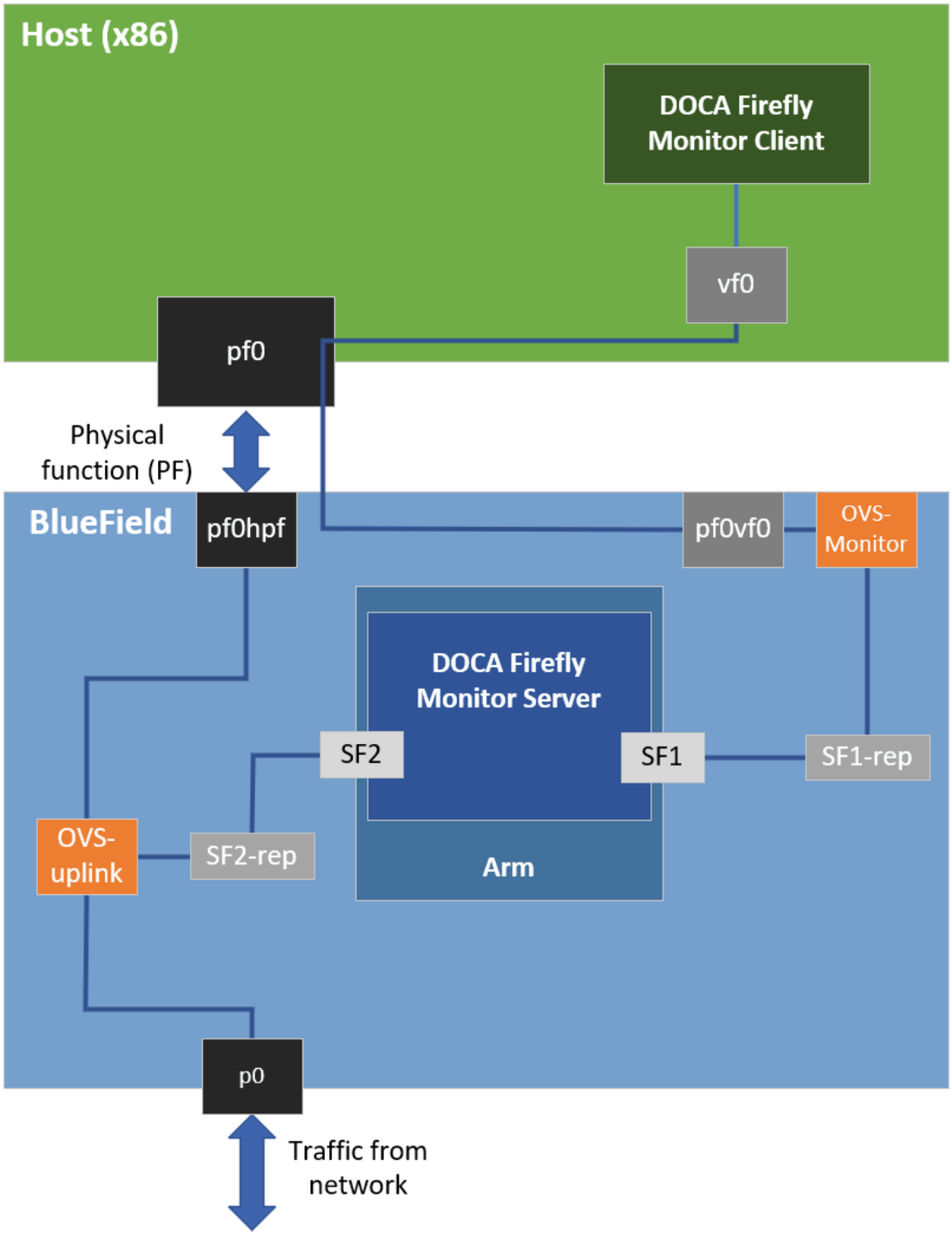
```
...
UtcOffset:                37
...
ptp_time (TAI):           Thu Sep  7 11:22:50 2023
ptp_time (UTC adjusted):  Thu Sep  7 11:22:13 2023
system_time (UTC):        Thu Sep  7 11:22:13 2023
```

This difference (37 seconds in the above example) is intentional and stems from the amount of leap seconds since epoch. This is indicated by the **UtcOffset** field that is also included in the monitor's report.

8.7.5.8.3 Monitor Server

In addition to printing the monitoring data to the container's standard output available through the container logs, the monitoring data is also exposed through a gRPC server that clients can subscribe to. This allows a monitoring client on the host to subscribe to monitor events from the service running on top of the DPU, thus providing better visibility.

The following diagram presents the recommended deployment architecture for connecting the monitoring client (on the host) to the monitor server (on the DPU).



Based on the above, when activating the monitor feature, the user must provide the IP address to be used by the monitor server:

```
- name: MONITOR_STATE
  value: "<IP address for the monitoring server>"
```

Users can choose to only view the monitoring events through the container logs without connecting to the monitoring server. In this case, it is recommended to configure the local host IP address (127.0.0.1) in the YAML file to avoid exposing it to an unwanted network.

8.7.5.8.4 Monitor Client

The required files for the monitor client are available under the service's dedicated NGC resource "scripts" directory.

To run Monitor Client you need to provide:

```
- name: MONITOR_STATE
  value: "<IP address for the monitoring server>"
- name: MONITOR_CLIENT_TYPE
  value: "vanilla"/"phc2sys"
```

vanilla is used for only monitoring the Server, while **phc2sys** is used for also running phc2sys on the host to synchronize the host's system clock with the Server's clock.

> ⚠ The monitor client **vanilla** type can run on both DPU and host, while the **phc2sys** type can only run on the host.

If you are using **phc2sys** you must also provide the interface the server is running on.

```
- name: MONITOR_CLIENT_PHC2SYS_INTERFACE
  value: "<interface_name>"
```

The monitor phc2sys client output will also include the following additional lines:

```
Host information:
Current system time:         Tue Apr  8 10:28:10 2025
Current phc time (Timezone): Tue Apr  8 10:28:10 2025
Current phc time (UTC):      2025-4-8 10:28:10.409982752 UTC
```

Example command line for executing the monitor client inside a container on host:

```
sudo docker run --privileged --net=host -v /var/log/doca/firefly:/var/log/firefly -v /etc/firefly:/etc/firefly -e
PTP_INTERFACE='eth2' -e MONITOR_STATE='192.168.0.1' -e MONITOR_CLIENT_TYPE='vanilla' -it nvcr.io/nvidia/doca/
doca_firefly:1.5.0-doca2.9.0-host /entrypoint.sh
```

Example command line for executing the python-based monitor client from a Linux host:

```
$ sudo pip3 install click protobuf grpcio
$ ./doca_firefly_monitor_client.py <ip-address-for-the-monitoring-server>
```

> ⚠ Reference source files and the **.proto** file used for Firefly's monitor are placed under the **src/** within the NGC resource.

8.7.5.8.5 Telemetry Export

On top of allowing clients to subscribe to the feed of monitoring events, the PTP Monitor server also supports an active export of the events to DOCA Telemetry Service (DTS). In such a scenario,

users should deploy DTS to the machine on which DOCA Firefly is deployed (host/DPU) and then activate "telemetry export" using the following steps:

1. Enable the needed file mounts through the YAML file:

Before

```
...
    # Uncomment when using the telemetry features with DTS - Part #1
    #- name: ipc-sockets-volume
    #  hostPath:
    #    path: /opt/mellanox/doca/services/telemetry/ipc_sockets
    #    type: DirectoryOrCreate
    #- name: shared-memory
    #  hostPath:
    #    path: /dev/shm/telemetry
    #    type: DirectoryOrCreate
  containers:
    - name: doca-firefly
...
      # Uncomment when debugging the finalized configuration files used - Part #2
      #- name: debug-firefly-volume
      #  mountPath: /tmp
      # Uncomment when using the telemetry features with DTS - Part #2
      #- name: ipc-sockets-volume
      #  mountPath: /opt/mellanox/doca/services/telemetry/ipc_sockets
      #- name: shared-memory
      #  mountPath: /dev/shm
...
```

After

```
...
    # Uncomment when using the telemetry features with DTS - Part #1
    - name: ipc-sockets-volume
      hostPath:
        path: /opt/mellanox/doca/services/telemetry/ipc_sockets
        type: DirectoryOrCreate
    - name: shared-memory
      hostPath:
        path: /dev/shm/telemetry
        type: DirectoryOrCreate
  containers:
    - name: doca-firefly
...
      # Uncomment when debugging the finalized configuration files used - Part #2
      #- name: debug-firefly-volume
      #  mountPath: /tmp
      # Uncomment when using the telemetry features with DTS - Part #2
      - name: ipc-sockets-volume
        mountPath: /opt/mellanox/doca/services/telemetry/ipc_sockets
      - name: shared-memory
        mountPath: /dev/shm
...
```

2. Pass a configuration value of `1` to `telemetry_export` through the configuration. This could easily be done directly through the YAML file:

Before

```
...
       # Example #7 - Activate the monitor telemetry export feature (export through DTS).
       #- name: CONF_MONITOR_global_telemetry_export
       #  value: "1"
...
```

After

```
...
       # Example #7 - Activate the monitor telemetry export feature (export through DTS).
       - name: CONF_MONITOR_global_telemetry_export
         value: "1"
...
```

Once active, a log message should indicate the availability of the export feature (which uses the DOCA Telemetry Exporter library):

```
...
2024-09-05 06:22:08 - Firefly - MONITOR - INFO     - Monitor records will also be exported via DOCA
Telemetry Exporter
...
```

The following is a sample output of the telemetry export:


```
"ptp_time_str": "Tue Apr 22 17:26:16 2025",
"adjusted_ptp_time_str": "Tue Apr 22 17:25:39 2025",
"sys_time_str": "Tue Apr 22 17:25:39 2025",
"last_error_time_str": "Tue Apr 22 12:40:47 2025",
"gm_identity": "18:08:31:FF:FE:5A:F1:EE (180831.fffe.5af1ee)",
"port_identity": "90:D7:6B:FF:FE:96:92:F8 (90d76b.fffe.9692f8-1)",
"ptp_ports": "90:D7:6B:FF:FE:96:92:F8 (90d76b.fffe.9692f8-1) - Slave",
"ptp_stability": 2,
"ptp_time_raw": 1745342776089236500,
"adjusted_ptp_time_raw": 1745342739089236500,
"sys_time_raw": 1745342739,
"error_count": 1,
"last_error_time_raw": 1745325647,
"master_offset_max": 1,
"master_offset_avg": 1,
"master_offset_rms": 0,
"utc_offset": 37,
"gm_priority1": 127,
"gm_clock_class": 248,
"gm_clock_accuracy": 254,
"gm_priority2": 127,
"gm_scaled_offset": 65535,
"domain_number": 0,
"port_state": 2,
"source_id": "c-237-153-80-p88-00-0-bf2.mtl.labs.mlnx",
"timestamp": 1745342739263351,
"data_type": "Firefly_monitor"
```


- **ptp_ports** – a list of all existing PTP ports and there state according to PTP protocol states.
- **ptp_stability** – represent the stability state of the PTP. It has 3 options:
  - 0 – PTP is in a stable state
  - 1 – PTP is currently out of sync
  - 2 – PTP managed to recover from a sync error
- **error_count** – represent the the number of errors we encountered thus far.
- **port_state** – represent the effective PTP port state of the most active port. It has 3 options:
  - 0 – PTP port is in an inactive state
  - 1 – PTP port is active, but uncalibrated
  - 2 – PTP port is active and calibrated

When locally debugging the telemetry information through DTS, it is important to remember to activate the data writer (storage output), as mentioned in the DOCA Telemetry Service Guide.

For more information about visualizing the information once exported through DTS, please refer to the example about using Grafana alongside DTS.

8.7.5.9 Firefly Servo

Firefly's Servo module can be seen as an extension to the built-in set of servos offered by **linuxptp**. When active, **linuxptp** is automatically set to "free running" and the control over the physical hardware clock (PHC) is handed over to Firefly's own servo.

The following is a sample output of this tool when using the **l2-telco** profile (16 messages per seconds):

```
2024-03-18 07:46:45 - Firefly - SERVO  - INFO     - Detected new master clock: 48b02d.fffe.5c4d24-1
2024-03-18 07:46:45 - Firefly - SERVO  - INFO     - Transition from servo state IDLE to FREE_RUNNING
```

```
2024-03-18 07:46:47 - Firefly - SERVO  - INFO     - Estimated a logSyncInterval of: -4
2024-03-18 07:46:47 - Firefly - SERVO  - INFO     - Measured offset      18691      delay -47
2024-03-18 07:46:48 - Firefly - SERVO  - INFO     - Transition from servo state FREE_RUNNING to LOCKED
2024-03-18 07:46:50 - Firefly - SERVO  - INFO     - offset +164 +/- 164 freq   -1.50 +/- 0.00  delay -48 +/- 1
2024-03-18 07:46:52 - Firefly - SERVO  - INFO     - Transition from servo state LOCKED to LOCKED_STABLE
2024-03-18 07:46:52 - Firefly - SERVO  - INFO     - offset   +0 +/- 1  freq   -1.41 +/- 0.47  delay -48 +/- 1
2024-03-18 07:46:54 - Firefly - SERVO  - INFO     - offset   -8 +/- 4  freq   -4.21 +/- 1.40  delay -47 +/- 1
2024-03-18 07:46:57 - Firefly - SERVO  - INFO     - offset  -12 +/- 2  freq   -5.46 +/- 0.73  delay -47 +/- 1
2024-03-18 07:46:59 - Firefly - SERVO  - INFO     - offset  -13 +/- 2  freq   -6.13 +/- 0.65  delay -47 +/- 1
2024-03-18 07:47:01 - Firefly - SERVO  - INFO     - offset  -13 +/- 3  freq   -6.19 +/- 1.23  delay -47 +/- 2
2024-03-18 07:47:03 - Firefly - SERVO  - INFO     - offset  -19 +/- 2  freq   -8.04 +/- 0.96  delay -47 +/- 1
2024-03-18 07:47:06 - Firefly - SERVO  - INFO     - offset  -14 +/- 3  freq   -6.46 +/- 1.11  delay -47 +/- 1
2024-03-18 07:47:08 - Firefly - SERVO  - INFO     - offset  -16 +/- 2  freq   -7.32 +/- 0.78  delay -48 +/- 2
2024-03-18 07:47:10 - Firefly - SERVO  - INFO     - offset  -15 +/- 2  freq   -7.11 +/- 0.87  delay -47 +/- 2
2024-03-18 07:47:12 - Firefly - SERVO  - INFO     - offset  -14 +/- 1  freq   -6.74 +/- 0.57  delay -47 +/- 2
2024-03-18 07:47:15 - Firefly - SERVO  - INFO     - offset  -12 +/- 3  freq   -6.20 +/- 1.01  delay -48 +/- 1
2024-03-18 07:47:17 - Firefly - SERVO  - INFO     - offset  -13 +/- 2  freq   -6.40 +/- 0.89  delay -47 +/- 1
2024-03-18 07:47:19 - Firefly - SERVO  - INFO     - offset  -11 +/- 2  freq   -5.98 +/- 0.86  delay -48 +/- 1
2024-03-18 07:47:21 - Firefly - SERVO  - INFO     - offset  -10 +/- 2  freq   -5.75 +/- 0.87  delay -46 +/- 1
2024-03-18 07:47:24 - Firefly - SERVO  - INFO     - offset   -8 +/- 1  freq   -5.15 +/- 0.42  delay -47 +/- 1
```

As can be seen, the servo's behavior is similar to that of **linuxptp**'s **ptp4l** and consists of a state machine that tracks the state of the active PTP port (**FREE_RUNNING** , **LOCKED** , **LOCKED_STABLE** , etc).

Firefly's Servo is disabled by default (in all profiles) and can be activated by replacing the **define_by_profile** value with **enable** :

```
# Activation status
- name: SERVO_STATE
  # Options are "enable"/"disable"/"defined_by_profile"
  value: "enable"
```

Once activated, the information can be viewed from the module's log file **/var/log/doca/firefly/servo.log** .

8.7.5.9.1 Firefly Servo Configuration

Firefly's Servo is currently aimed for telco-related deployments, using the **l2-telco** profile including the use of SyncE. As such, the default values in the built-in configuration file are optimized for those scenarios.

The servo supports configuration options which are passed through a dedicated configuration file like the rest of DOCA Firefly's modules. The built-in servo configuration file can be found in the section " Firefly Servo". For ease of use, the file is also provided as part of DOCA's container resource as downloaded from NGC.

"Firefly Modules Configuration Options" contains a complete explanation of each of the configuration options alongside their default values.

To set a custom config file, users should locate their config file in the directory **/etc/firefly** and set the config file name in DOCA Firefly's YAML file.

```
- name: SERVO_CONFIG_FILE
  value: my_custom_servo.conf
```

In this example, **my_custom_servo.conf** should be placed at **/etc/firefly/my_custom_servo.conf** .

8.7.5.9.2 Dynamic Packet Rate Support

The servo has the ability to dynamically detect the packet rate used by the PTP grandmaster clock, so to calibrate itself accordingly incase it differs from the recommended 16 packets per seconds.

```
2024-03-18 07:46:45 - Firefly - SERVO  - INFO     - Transition from servo state IDLE to FREE_RUNNING
2024-03-18 07:46:47 - Firefly - SERVO  - INFO     - Estimated a logSyncInterval of: -4
2024-03-18 07:46:47 - Firefly - SERVO  - INFO     - Measured offset      18691      delay -47
```

In a case the message rate is constant and known in advance, the dynamic estimation can be disabled, in favor of a provided message rate:

```
- name: CONF_SERVO_global_servo_const_log_sync_interval
  value: "-2"
```

In the above example, a fixed message rate of 4 packets per seconds will be used (**logSyncInterval** of **-2**).

> While the servo was tested to produce stable results with various packets rates (2, 4, 8, 16, 32, 64, 128), it is only officially recommended for use in deployments using a packet rate of 16 packets per second.

8.7.5.10 VLAN Tagging

DOCA Firefly natively supports VLAN-tagging-enabled network interfaces.

8.7.5.10.1 Separated Mode

The name of the VLAN-enabled network interface should be the one passed through the YAML file in the **PTP_INTERFACE** field.

8.7.5.10.2 Embedded Mode

In addition to passing on the VLAN-enabled interface through the YAML as listed in the previous section, the user is also required to configure the network routing within the DPU to support the VLAN tagging:

1. The following example configures a VLAN tag of 10 to the **enp3s0f0s4** interface:

   ```
   $ sudo ip link add link enp3s0f0s4 name enp3s0f0s4.10 type vlan id 10
   $ sudo ip link set up enp3s0f0s4.10
   $ sudo ifconfig enp3s0f0s4.10 192.168.104.1 up
   ```

   In this example, **enp3s0f0s4.10** is the interface to be passed to DOCA Firefly.
2. Additional commands to route the traffic within the DPU:

   ```
   $ sudo ovs-ofctl add-flow uplink in_port=en3f0pf0sf4,dl_vlan=10,actions=output:p0
   $ sudo ovs-ofctl add-flow uplink in_port=p0,dl_vlan=10,actions=output:en3f0pf0sf4
   ```

8.7.5.11 Multiple Interfaces

DOCA Firefly can support multiple network interfaces through the following YAML file syntax:

```
- name: PTP_INTERFACE
  value: "<space (' ') separated list of interface names>"
```

For example:

```
- name: PTP_INTERFACE
  value: "p0 p1"
```

> The monitoring feature is supported for multiple interfaces only when the **clientOnly** configuration is enabled.

> For Firefly versions lower than 1.7.0, automatic mode (**-a**) for **phc2sys** is not supported when working with multiple interfaces. It is recommended to disable **phc2sys** in this mode.

8.7.6 Troubleshooting

When troubleshooting container deployment issues, it is highly recommended to follow the deployment steps and tips in the "Review Container Deployment" section of the NVIDIA DOCA Container Deployment Guide.

To debug the finalized configuration file used by Firefly, users can connect to the container as follows:

1. Open a shell session on the running container using the container ID:

   ```
   sudo crictl exec -it <container-id> /bin/bash
   ```

2. Once connected to the container, the finalized configuration file can be found under the **/tmp** directory using the same filename as the original configuration file.

   > More information regarding the configuration files can be found under section "Ensuring and Debugging Correctness of Config File".

8.7.6.1 Pod is Marked as "Ready" and No Container is Listed

8.7.6.1.1 Error

When deploying the container, the pod's STATE is marked as **Ready** , an image is listed, however no container can be seen running:

```
$ sudo crictl pods
```

```
POD ID              CREATED             STATE               NAME                                NAMESPACE
ATTEMPT             RUNTIME
06bd84c07537e       4 seconds ago       Ready               doca-firefly-my-dpu                 default
0                   (default)

$ sudo crictl images
IMAGE                                TAG                 IMAGE ID            SIZE
k8s.gcr.io/pause                     3.2                 2a060e2e7101d       251kB
nvcr.io/nvidia/doca/doca_firefly     1.1.0-doca2.0.2     134cb22f34611       87.4MB

$ sudo crictl ps
CONTAINER           IMAGE               CREATED             STATE               NAME                ATTEMPT
POD ID              POD
```

8.7.6.1.2 Solution

In most cases, the container did start, but immediately exited. This could be checked using the following command:

```
$ sudo crictl ps -a
CONTAINER           IMAGE               CREATED             STATE               NAME                ATTEMPT
POD ID              POD
556bb78281e1d       134cb22f34611       7 seconds ago       Exited              doca-firefly        1
          06bd84c07537e       doca-firefly-my-dpu
```

Should the container fail (i.e., state of **Exited**) it is recommended to examine Firefly's main log at **/var/log/doca/firefly/firefly.log**.

In addition, for a short period of time after termination, the container logs could also be viewed using the the container's ID:

```
$ sudo crictl logs 556bb78281e1d
Starting DOCA Firefly - Version 1.1.0
...
Requested the following PTP interface: p10
Failed to find interface "p10". Aborting
```

8.7.6.2 Custom Config File is Not Found

8.7.6.2.1 Error

When DOCA Firefly is deployed using a custom configuration file, a deployment error occurs and the following log message appears:

```
...
2023-09-07 14:04:23 - Firefly - Init    - ERROR    - Custom config file not found: my_file.conf. Aborting
...
```

8.7.6.2.2 Solution

Check the custom file name written in the YAML file and make sure that you properly placed the file with that name under the **/etc/firefly/** directory of the DPU.

8.7.6.3 Profile is Not Supported

8.7.6.3.1 Error

When DOCA Firefly is deployed, a deployment error occurs and the following log message appears:

```
...
2023-09-07 14:04:23 - Firefly - Init    - ERROR    - profile <name> is not supported. Aborting
...
```

8.7.6.3.2 Solution

Verify that the profile selected in the YAML file matches one of the supported profiles as listed in the profiles table.

> ⚠ The profile name is case sensitive. The name must be specified in lower-case letters.

8.7.6.4 PPS Capability is Missing

8.7.6.4.1 Error

When DOCA Firefly is deployed and configured to use the **PPS** module, a deployment error occurs and the following log message appears:

```
...
2023-09-07 14:04:23 - Firefly - Init    - INFO     - Starting PPS configuration
2023-09-07 14:04:23 - Firefly - Init    - WARNING  - [-] PPS capability is missing, seems that the card doesn't
support PPS
2023-09-07 14:04:23 - Firefly - Init    - INFO     - capabilities:
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   50000000 maximum frequency adjustment (ppb)
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   0 programmable alarms
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   0 external time stamp channels
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   0 programmable periodic signals
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   0 pulse per second
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   0 programmable pins
2023-09-07 14:04:23 - Firefly - Init    - INFO     -   0 cross timestamping
...
```

8.7.6.4.2 Solution

This log indicates that the DPU hardware does not support PPS. However, PTP can still run on this hardware, and you should see the line **Running ptp4l** in the container log, indicating that PTP is running successfully.

8.7.6.5 Timed Out While Polling for Tx Timestamp

8.7.6.5.1 Error

When the BlueField is operating in DPU mode, DOCA Firefly gets stuck in a fault loop while waiting to receive the Tx timestamp events:

```
ptp4l[2912.797]: timed out while polling for tx timestamp
ptp4l[2912.797]: increasing tx_timestamp_timeout may correct this issue, but it is likely caused by a driver bug
ptp4l[2912.797]: port 1 (enp3s0f0s4): send sync failed
ptp4l[2923.528]: timed out while polling for tx timestamp
ptp4l[2923.528]: increasing tx_timestamp_timeout may correct this issue, but it is likely caused by a driver bug
ptp4l[2923.528]: port 1 (enp3s0f0s4): send sync failed
```

> ℹ DOCA Firefly has a known gap leading to this error appearing once, after which ptp4l recovers from it. This section only covers the case in which there is a fault loop and no recovery occurs.

8.7.6.5.2 Solution

DOCA Firefly's configurations were already adjusted to accommodate for Tx port timestamping. For more information about the reason for this error and for the designed recovery mechanism from it, refer to section " Tx Timestamping Support on DPU Mode".

8.7.6.6 Warning – Time Jumped Backwards

8.7.6.6.1 Error

When using Firefly's Servo module, the following warning log message is encountered on start:

```
 2024-01-01 14:04:23 - Firefly - SERVO   - WARNING  - Clock is going to jump backwards in time - this might have a
system-wide impact
```

8.7.6.6.2 Solution

This warning message indicates that the system's time jumped backwards with a value of at least one minute. This event is logged by Firefly given that such jumps might have system-wide implications. For more information, refer to section "Failed to Reserve Sandbox Name" in the DOCA Troubleshooting .

Such jumps can only happen during Firefly's boot, before the Servo achieves initial time synchronization with the reference clock.

8.7.7 PTP Profile Default Config Files

8.7.7.1 Media Profile

```
#
# This config file contains configurations for media & entertainment alongside
# DOCA Firefly specific adjustments.
#

[global]
domainNumber            127
priority1               128
priority2               127
use_syslog                1
logging_level             6
tx_timestamp_timeout     30
hybrid_e2e                1
dscp_event               46
dscp_general             46
logAnnounceInterval      -2
announceReceiptTimeout    3
logSyncInterval          -3
logMinDelayReqInterval   -3
delay_mechanism         E2E
network_transport     UDPv4
# Value lesser or equal to 1 is required for Embedded Mode
fault_reset_interval      1
# Required for multiple interfaces support
boundary_clock_jbod       1
```

8.7.7.2 Default Profile

```
#
# This config file extends linuxptp default.cfg config file with DOCA Firefly
# specific adjustments.
#
```

```
[global]
# Value lesser or equal to 1 is required for Embedded Mode
fault_reset_interval                1
# Required for multiple interfaces support
boundary_clock_jbod                 1
```

8.7.7.3 Telco (L2) Profile

```
#
# This config file extends linuxptp G.8275.1.cfg config file with DOCA Firefly
# specific adjustments.
#

[global]
dataset_comparison                        G.8275.x
G.8275.defaultDS.localPriority                 128
maxStepsRemoved                                255
logAnnounceInterval                             -3
logSyncInterval                                 -4
logMinDelayReqInterval                          -4
G.8275.portDS.localPriority                    128
ptp_dst_mac                     01:80:C2:00:00:0E
network_transport                              L2
domainNumber                                   24
# Value lesser or equal to 1 is required for Embedded Mode
fault_reset_interval                            1
# Required for multiple interfaces support
boundary_clock_jbod                             1
```

8.7.8 Firefly Modules Configuration Options

8.7.8.1 PTP Monitor

8.7.8.1.1 monitor-default.conf

```
#
# Default values for all of Firefly's PTP monitor configuration values.
#

[global]
# General
report_interval                    1000
# Debugging & Logging
doca_logging_level                   50
telemetry_export                      0
```

8.7.8.1.2 Configuration Options

- **report_interval** – The time interval (in msecs) for when the monitor should publish a report to all defined output providers (standard output, gRPC clients, etc). Default: 1000 (1 second).
- **doca_logging_level** – Logging level for the module, based on DOCA's logging levels. Default is 50 (INFO). Valid options:
  - 10=DISABLE
  - 20=CRITICAL
  - 30=ERROR
  - 40=WARNING
  - 50=INFO
  - 60=DEBUG
- **telemetry_export** – Indicates whether monitor information should be exported through DOCA Telemetry Service. Valid options:
  - 0=Disabled (default)

- 1=Enabled

8.7.8.2 Firefly Servo

8.7.8.2.1 servo-default.conf

```
#
# Default values for all of Firefly's servo configuration values
#

[global]
# Time thresholds
init_offset_from_master_threshold   10000000
offset_from_master_min_threshold       -1500
offset_from_master_max_threshold        1500
init_max_time_adjustment                   0
max_time_adjustment                     1500
step_adjustment_threshold                  0
hold_over_timer                            0
# Sampling Window & servo logic
warmup_period                           1500
sync_filter_length                         6
delay_request_filter_length                6
servo_adjustment_interval                  4
servo_init_adjustment_interval            24
servo_const_log_sync_interval           0xFF
servo_window_min_samples                   2
servo_num_offset_values                    5
servo_pi_cutoff_frequency             0.0159
servo_pi_dumping_factor                 7.85

# Debugging & Logging
summary_interval                        2000
doca_logging_level                        50
free_running                               0
```

8.7.8.2.2 Configuration Options

- **init_offset_from_master_threshold** – Minimal threshold (in nsecs) for switching to PI servo on init. Default is 10000000 (10 msecs).
- **offset_from_master_min_threshold** – Minimal threshold (in nsecs) for declaring time offset from the master clock as "stable". Default is -1500 (-1.5 µsecs).
- **offset_from_master_max_threshold** – Maximal threshold (in nsecs) for declaring time offset from the master clock as "stable". Default is +1500 (+1.5 µsecs).
- **init_max_time_adjustment** – When active, defines the maximal allowed time (step) adjustment (in nsecs) before the servo reaches the "locked" state. Default is 0 (disabled).
- **max_time_adjustment** – When active, defines the maximal allowed reference time adjustment (in nsecs) after the servo has reached the "locked" state. Default is 1500 (1.5 µsecs).
- **step_adjustment_threshold** – When active, defines the thresholds above which a time (step) adjustment (in nsecs) would be allowed, even after the servo has reached the "locked" state. Default is 0 (disabled).
- **hold_over_timer** – When active, defines the time duration (in seconds) in which the servo stays in "hold over" mode, until reverting back to "free running". Default is 0 ("hold over" state is disabled).
- **warmup_period** – Time span (in msecs) during which samples are collected to estimate the **logSyncInterval** value (packet rate). Default is 1500 (1.5 seconds).
- **sync_filter_length** – Number of **SYNC** messages in the servo's history buffer. Default is 6.
- **delay_request_filter_length** – Number of **DELAY_REQUEST** messages in the servo's history buffer. Default is 6 messages.

- **servo_adjustment_interval** – Number of **SYNC** messages after which the PHC is updated once the servo has reached the "locked" state at least once. Default is 4 messages.
- **servo_init_adjustment_interval** – Number of **SYNC** messages after which the PHC is updated before the servo has ever reached the "locked" state. Default is 24 messages.
- **servo_const_log_sync_interval** – Known fixed value to be used as the **logSyncInterval** instead of trying to estimate it at runtime. Default is 0xFF (disabled).
- **servo_window_min_samples** – Minimal number of samples needed for a servo calculation. Default is 2 messages.
- **servo_num_offset_values** – Number of consecutive timestamps within the "offset from master" threshold that are required so to transition from the "locked" state and to the "locked stable" state. Default is 5 offset values.
- **servo_pi_cutoff_frequency** – The PI servo's cutoff frequency value. Default is 0.0159.
- **servo_pi_dumping_factor** – The PI servo's dumping factor value. Default is 7.85.
- **summary_interval** – The time interval (in msecs) for when the servo should publish a report log event. Default is 2000 (2 seconds).
- **doca_logging_level** – Logging level for the module, based on DOCA's logging levels. Default is 50 (INFO). Valid options:
  - 10=DISABLE
  - 20=CRITICAL
  - 30=ERROR
  - 40=WARNING
  - 50=INFO
  - 60=DEBUG
- **free_running** – Tell the servo to only log the operations, without actually adjusting the PHC. Default is 0 (disabled).

8.8 DOCA Flow Inspector Service Guide

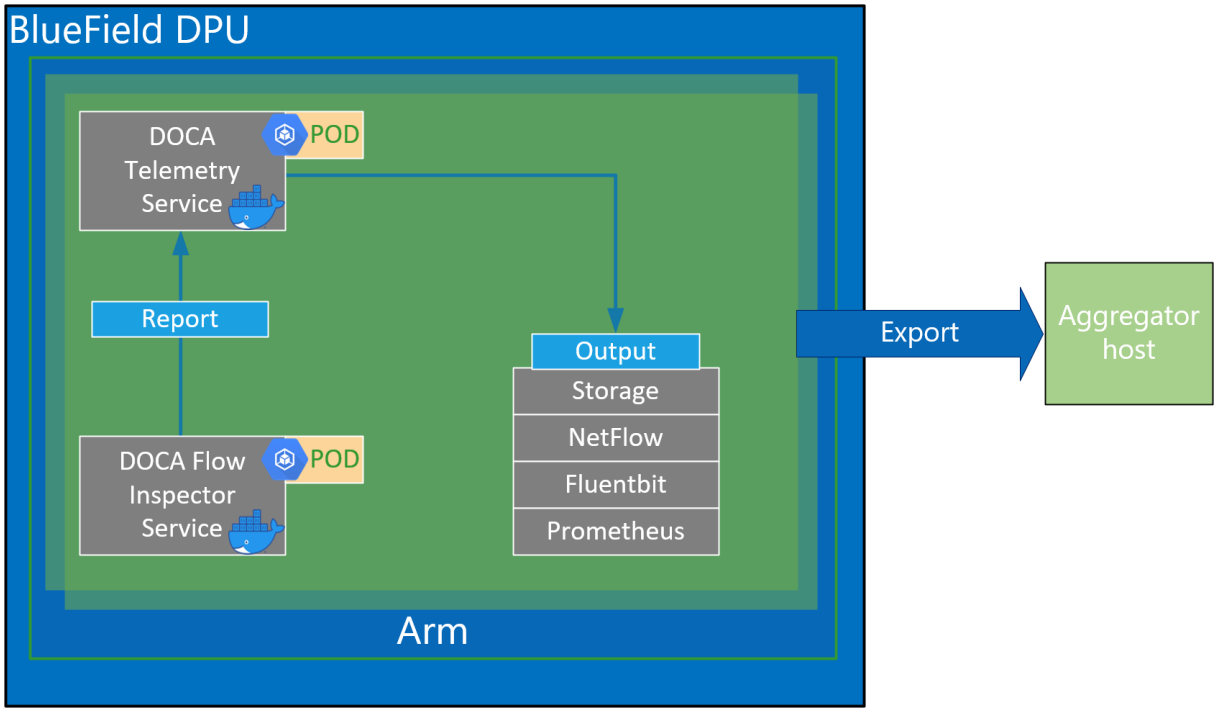
This guide provides instructions on how to use the DOCA Flow Inspector service container on top of NVIDIA® BlueField® DPU.

8.8.1 Introduction

DOCA Flow Inspector service enables real-time data monitoring and extraction of telemetry components. These components can be leveraged by various services, including those focused on security, big data, and other purposes.

DOCA Flow Inspector service is linked to DOCA Telemetry Service (DTS). It receives mirrored packets from the user parses the data, and forwards it to the DTS, which aggregates predefined statistics from various providers and sources. The service utilizes the DOCA Telemetry Exporter API to communicate with the DTS, while the DPDK infrastructure facilitates packet acquisition at a user-space layer.

DOCA Flow Inspector operates within its dedicated Kubernetes pod on BlueField, aimed at receiving mirrored packets for analysis. The received packets are parsed and transmitted, in a predefined structure, to a telemetry collector that manages the remaining telemetry aspects.



8.8.1.1 Service Flow

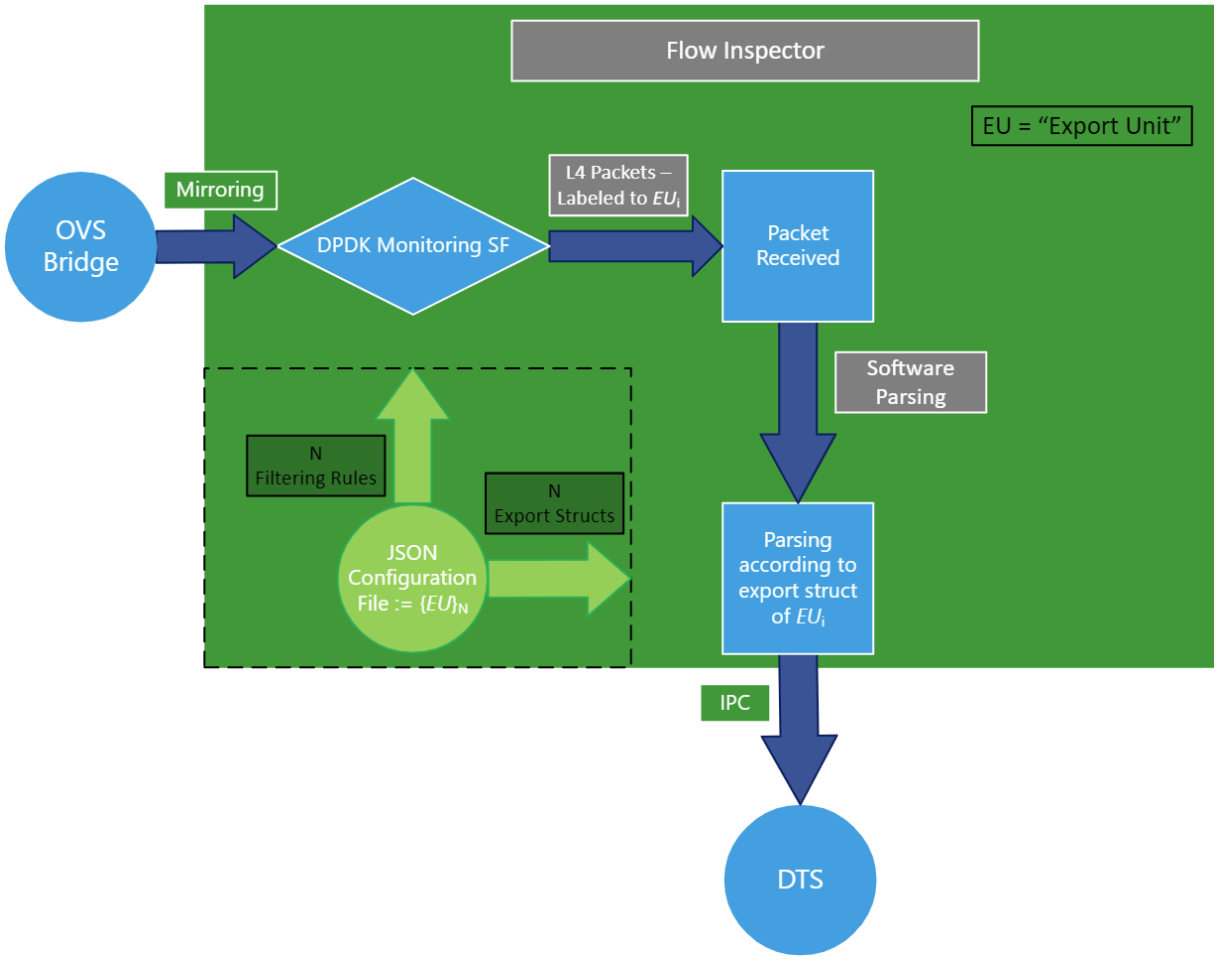
The DOCA Flow Inspector receives a configuration file in a JSON format which includes which of the mirrored packets should be filtered and which information should be sent to DTS for inspection.

The configuration file can include several export units under the "export-units" attribute. Each one is comprised of a "filter" and an "export". Each packet that matches one filter (based on the protocol and ports in the L4 header) is then parsed to the corresponding requested struct defined in the export. That information only is sent for inspection. A packet that does not match any filter is dropped.

In addition, the configuration file could contain FI optional configuration flags, see JSON format and example in the Configuration section.

The service watches for changes in the JSON configuration file in runtime and for any change that reconfigures the service.

The DOCA Flow Inspector runs on top of DPDK to acquire L4. The packets are then filtered and HW-marked with their export unit index. The packets are then parsed according to their export unit and export struct, and then forwarded to the telemetry collector using IPC.



Configuration phase:

1. A JSON file is used as input to configure the export units (i.e., filters and corresponding export structs).
2. The filters are translated to HW rules on the SF (scalable function port) using the DOCA Flow library.
3. The connection to the telemetry collector is initialized and all export structures are registered to DTS.

Inspection phase:

1. Traffic is mirrored to the relevant SF.
2. Ingress traffic is received through the configured SF.
3. Non-L4 traffic and packets that do not match any filter are dropped using hardware rules.
4. Packets matching a filter are marked with the export unit index they match and are passed to the software layer in the Arm cores.
5. Packets are parsed to the desired struct by the index of export unit.
6. The telemetry information is forwarded to the telemetry agent using IPC.
7. Mirrored packets are freed.
8. If the JSON file is changed, run the configuration phase with the updated file.

8.8.2 Requirements

Before deploying the flow inspector container, ensure that the following prerequisites are satisfied:

1. Create the needed files and directories. Folders should be created automatically. Make sure the **.json** file resides inside the folder:

```
$ touch /opt/mellanox/doca/services/flow_inspector/bin/flow_inspector_cfg.json
```

   Validate that DTS's configuration folders exist. They should be created automatically when DTS is deployed.

```
$ sudo mkdir -p /opt/mellanox/doca/services/telemetry/config
$ sudo mkdir -p /opt/mellanox/doca/services/telemetry/ipc_sockets
$ sudo mkdir -p /opt/mellanox/doca/services/telemetry/data
```

2. Allocate huge pages as needed by DOCA Flow:

```
$ echo '1024' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
$ sudo mkdir /mnt/huge
$ sudo mount -t hugetlbfs -o pagesize=2M nodev /mnt/huge
```

   Deploy a scalable function according to BlueField Scalable Functions and mirror packets accordingly using the Open vSwitch command.
   For example:

   a. Mirror packets from **p0** to **sf4**:

```
$ ovs-vsctl add-br ovsbr1
$ ovs-vsctl add-port ovsbr1 p0
$ ovs-vsctl add-port ovsbr1 en3f0pf0sf4
$ ovs-vsctl -- --id=@p1 get port en3f0pf0sf4 \
           -- --id=@p2 get port p0 \
           -- --id=@m create mirror name=m0 select-dst-port=@p2 select-src-port=@p2 output-
port=@p1 \
           -- set bridge ovsbr1 mirrors=@m
```

   b. Mirror packets from **pf0hpf** or **p0** that pass through **sf4**:

```
$ ovs-vsctl add-br ovsbr1
$ ovs-vsctl add-port ovsbr1 pf0hpf
$ ovs-vsctl add-port ovsbr1 p0
$ ovs-vsctl add-port ovsbr1 en3f0pf0sf4
$ ovs-vsctl -- --id=@p1 get port en3f0pf0sf4 \
           -- --id=@p2 get port pf0hpf \
           -- --id=@m create mirror name=m0 select-dst-port=@p2 select-src-port=@p2 output-
port=@p1 \
           -- set bridge ovsbr1 mirrors=@m
$ ovs-vsctl -- --id=@p1 get port en3f0pf0sf4 \
           -- --id=@p2 get port p0 \
           -- --id=@m create mirror name=m0 select-dst-port=@p2 select-src-port=@p2 output-
port=@p1 \
           -- set bridge ovsbr1 mirrors=@m
```

   The output of last command (creating the mirror) should output a sequence of letters and numbers similar to the following:

```
0d248ca8-66af-427c-b600-af1e286056e1
```

   ⚠ The designated SF must be created as a trusted function. Additional details can be found in the BlueField Scalable Functions.

8.8.3 Service Deployment

For information about the deployment of DOCA containers on top of the BlueField DPU, refer to NVIDIA DOCA Container Deployment Guide.

DTS is available on NGC, NVIDIA's container catalog. Service-specific configuration steps and deployment instructions can be found under the service's container page.

> ⚠ The order of running DTS and DOCA Flow Inspector is important. You must launch DTS, wait a few seconds, and then launch DOCA Flow Inspector.

8.8.4 Configuration

8.8.4.1 JSON Input

The DOCA Flow Inspector configuration file should be placed under **/opt/mellanox/doca/services/flow_inspector/bin/<json_file_name>.json** and be built in the following format:

```
{
    /* Optional param, time period to check for changes in JSON config file (in seconds) and flush telemetry buffer
if enabled (default is 60 seconds) */
    "config-sample-rate": <time>,

    /* Optional param, telemetry buffer size in bytes (default is 60KB) */
    "telemetry-buffer-size": <size>,

    /* Optional param, enable periodic telemetry buffer flush and defining the period time (in seconds) */
    "telemetry-flush-rate": <numeric value in seconds>,

    /* Mandatory param, Flow Inspector export units */
    "export-units":
    [

        /* Export Unit 0 */
        {
            "filter":
            {     "protocols": [<L4 protocols separated by comma>], # What L4 protocols are allowed
                  "ports":
                    [
                            [<source port>, <destination port>],
                            [<source ports range>, <destination ports range>],
                            <... more pairs of source, dest ports>
                    ]
            },
            "export":
            {
                    "fields": [<fields to be part of export struct, separated by comma>] # the Telemetry event will
contain these fields.

            }
        },
        <... More Export Units>
    ]
}
```

8.8.4.1.1 Export Unit Attributes

Allowed protocols:

- **"TCP"**
- **"UDP"**

Port range:

- It is possible to insert a range of ports for both source and destination
- Range should include borders **[start_port-end_port]**

Allowed ports:

- All ports in range `0` - `65535` as a string
- Or `*` to indicate any ports

Allowed fields in export struct:

- **`timestamp`** – timestamp indicating when it was received by the service
- **`host_ip`** – the IP of the host running the service
- **`src_mac`** – source MAC address
- **`dst_mac`** – destination MAC address
- **`src_ip`** – source IP
- **`dst_ip`** – destination IP
- **`protocol`** – L4 protocol
- **`src_port`** – source port
- **`dst_port`** – destination port
- **`flags`** – additional flags (relevant to TCP only)
- **`data_len`** – data payload length
- **`data_short`** – short version of data (payload sliced to first 64 bytes)
- **`data_medium`** – medium version of data (payload sliced to first 1500 bytes)
- **`data_long`** – long version of data (payload sliced to first 9*1024 bytes)

JSON example:

```
{

    /* Optional param, time period to check for changes in JSON config file (in seconds) and flush telemetry buffer
if enabled (default is 60 seconds) */
    "config-sample-rate": 30,

    /* Optional param, telemetry maximum buffer size in bytes */
    "telemetry-buffer-size": 70000,

    /* Optional param, enable periodic telemetry buffer flush and defining the period time (in seconds) */
    "telemetry-flush-rate": 1.5,

   /* Mandatory param, Flow Inspector export units */
    "export-units":
    [

        /* Export Unit 0 */
        {
            "filter":
            {
                "protocols": ["tcp", "udp"],
                "ports":
                    [
                        ["*","433-460"],
                        ["20480","28341"],
                        ["28341","20480"],
                        ["68", "67"],
                        ["67", "68"]
                    ]
            },
            "export":
            {
                "fields": ["timestamp", "host_ip", "src_mac", "dst_mac", "src_ip", "dst_ip", "protocol",
"src_port",
                    "dst_port", "flags", "data_len", "data_long"]
            }
        },

        /* Export Unit 1 */
        {
            "filter":
            {
                "protocols": ["tcp"],
                "ports":
                    [
                        ["5-10","422"],
                        ["80","80"]
                    ]
            },
            "export":
            {
```

```
                "fields": ["timestamp","dst_ip", "host_ip", "data_len", "flags", "data_medium"]
            }
        }
    ]
}
```

If a packet header contains L4 ports or L4 protocol which are not specified in any filter, they are filtered out.

8.8.4.2 Yaml File

The **.yaml** file downloaded from NGC can be easily edited according to your needs.

```
env:
  # Set according to the local setup
  - name: SF_NUM_1
    value: "2"   # Additional EAL flags, if needed
  - name: EAL_FLAGS
    value: ""   # Service-Specific command line arguments
  - name: SERVICE_ARGS
    value: "--policy /flow_inspector/flow_inspector_cfg.json -l 60"
```

- The **SF_NUM_1** value can be changed according to the SF used in the OVS configuration and can be found using the command in BlueField Scalable Functions.
- The **EAL_FLAGS** value must be changed according to the DPDK flags required when running the container.
- The **SERVICE_ARGS** are the runtime arguments received by the service:
  - **-l** , **--log-level <value>** – sets the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  - **-p** , **--policy <json_path>** – sets the JSON path inside the container

8.8.4.3 Verifying Output

Enabling write to data in the DTS allows debugging the validity of the DOCA Flow Inspector.

To allow DTS to write locally, uncomment the following line in **/opt/mellanox/doca/services/telemetry/config/dts_config.ini** :

```
#output=/data
```

Any changes in **dts_config.ini** necessitate restarting the pod for the new settings to apply.

The schema folder contains JSON-formatted metadata files which allow reading the binary files containing the actual data. The binary files are written according to the naming convention shown in the following example:

Requires installing the **tree** runtime utility (**apt install tree**).

```
$ tree /opt/mellanox/doca/services/telemetry/data/
/opt/mellanox/doca/services/telemetry/data/
    {year}
```

```
        {mmdd}
            {hash}
                {source_id}
                    {source_tag}{timestamp}.bin
                {another_source_id}
                    {another_source_tag}{timestamp}.bin
    schema
        schema_{MD5_digest}.json
```

New binary files appear when:

- The service starts
- When the binary file's max age/size restriction is reached
- When JSON file is changed and new schemas of telemetry are created
- An hour passes

If no schema or no data folders are present, refer to the Troubleshooting section in DOCA Telemetry Service Guide.

source_id is usually set to the machine hostname. **source_tag** is a line describing the collected counters, and it is often set as the provider's name or name of user-counters.

Reading the binary data can be done from within the DTS container using the following command:

```
crictl exec -it <Container-ID> /opt/mellanox/collectx/bin/clx_read -s /data/schema /data/path/to/datafile.bin
```

The data written locally should be shown in the following format assuming a packet matching Export Unit 1 from the example has arrived:

```
{
    "timestamp": 1656427771076130,
    "host_ip": "10.237.69.238",
    "src_ip": "11.7.62.4",
    "dst_ip": "11.7.62.5",
    "data_len": 1152,
    "data_short": "Hello World"
}
```

8.8.5 Troubleshooting

When troubleshooting container deployment issues, it is highly recommended to follow the deployment steps and tips in the "Review Container Deployment" section of the NVIDIA DOCA Container Deployment Guide.

8.8.5.1 Pod is Marked as "Ready" and No Container is Listed

8.8.5.1.1 Error

When deploying the container, the pod's STATE is marked as **Ready**, an image is listed, however no container can be seen running:

```
$ sudo crictl pods
POD ID              CREATED             STATE               NAME
NAMESPACE           ATTEMPT             RUNTIME
3162b71e67677       4 seconds ago       Ready               doca-flow-inspector-my-dpu                      default
0                   (default)

$ sudo crictl images
IMAGE                                 TAG                 IMAGE ID            SIZE
k8s.gcr.io/pause                      3.2                 2a060e2e7101d       487kB
nvcr.io/nvidia/doca/doca_flow_inspector   1.1.0-doca2.0.2     2af1e539eb7ab       86.8MB
```

```
$ sudo crictl ps
CONTAINER           IMAGE               CREATED             STATE               NAME                ATTEMPT
POD ID              POD
```

8.8.5.1.2 Solution

In most cases, the container did start, but immediately exited. This could be checked using the following command:

```
$ sudo crictl ps -a
CONTAINER           IMAGE               CREATED             STATE               NAME                ATTEMPT
POD ID              POD
556bb78281e1d       2af1e539eb7ab       6 seconds ago       Exited              doca-flow-inspector      1
          3162b71e67677       doca-flow-inspector-my-dpu
```

Should the container fail (i.e., state of **Exited**), it is recommended to examine the Flow Inspector's main log at **/var/log/doca/flow_inspector/flow_inspector_fi_dev.log**.

In addition, for a short period of time after termination, the container logs could also be viewed using the container's ID:

```
$ sudo crictl logs 556bb78281e1d
...
2023-10-04 11:42:55 - flow_inspector - FI     - ERROR    - JSON file was not found <config-file-path>.
```

8.8.5.2 Pod is Not Listed

8.8.5.2.1 Error

When placing the container's YAML file in the Kubelet's input folder, the service pod is not listed in the list of pods:

```
$ sudo crictl pods
POD ID              CREATED             STATE               NAME
NAMESPACE           ATTEMPT             RUNTIME
```

8.8.5.2.2 Solution

In most cases, the pod does not start due to the absence of the requested hugepages. This can be verified using the following command:

```
$ sudo journalctl -u kubelet -e. . .
Oct 04 12:12:19 <my-dpu> kubelet[2442376]: I1004 12:12:19.905064 2442376 predicate.go:103] "Failed to admit pod,
unexpected error while attempting to recover from admission failure" pod="default/doca-flow-inspector-<my-dpu>"
 err="preemption: error finding a set of pods to preempt: no set of running pods found to reclaim resources: [(res:
hugepages-2Mi, q: 104563999874), ]"
```

8.9 DOCA HBN Service Guide

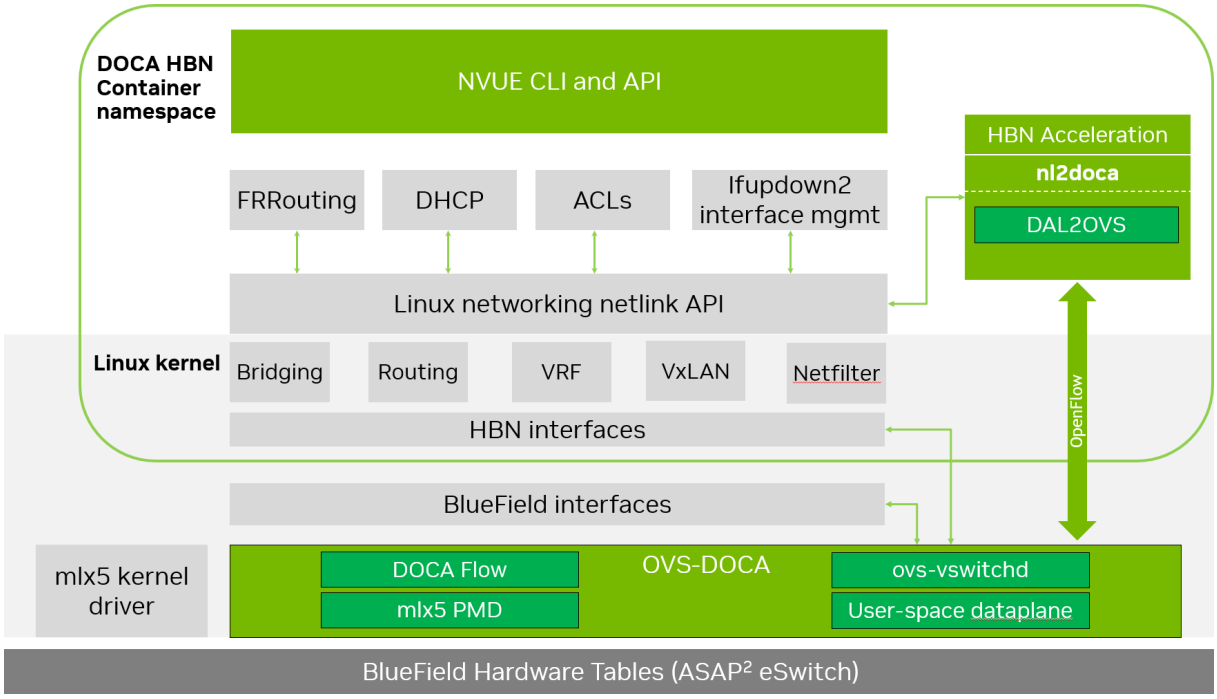
This guide provides instructions on how to use the DOCA HBN Service container on top of NVIDIA® BlueField® networking platform.

8.9.1 Introduction

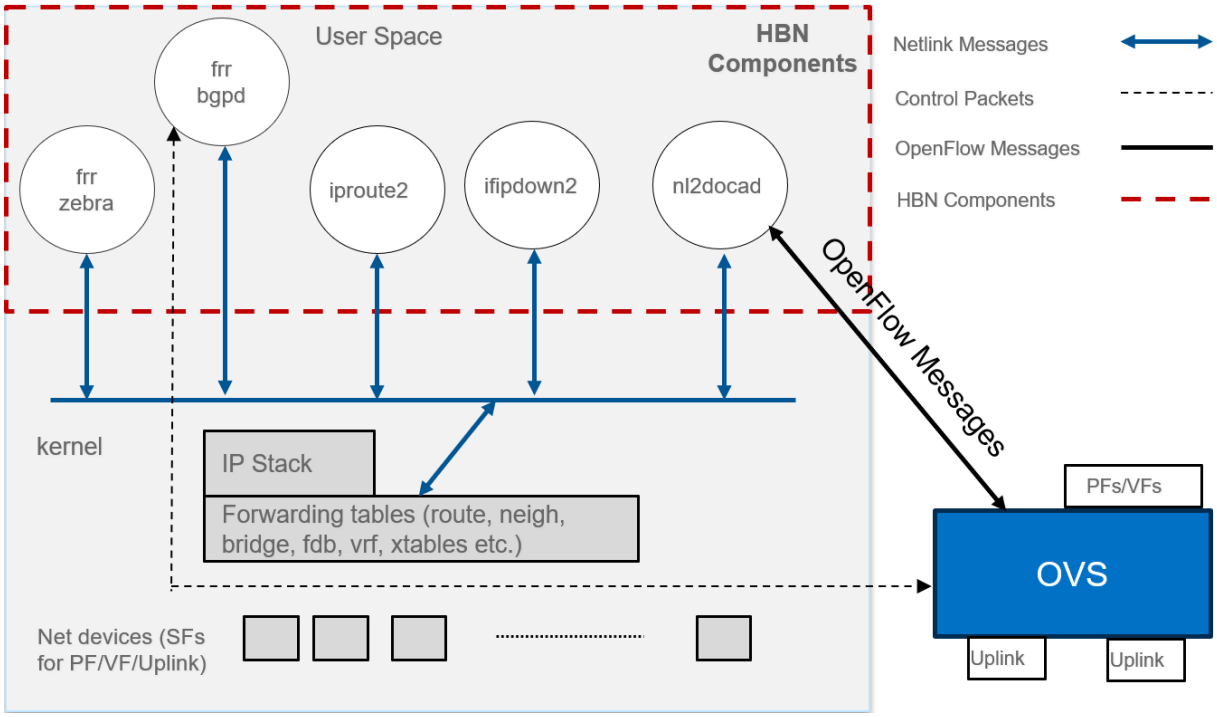
Host-based Networking (HBN) is a DOCA service that allows network architects to design networks based on layer-3 (L3) protocols. HBN enables routing to run on the server side by using BlueField as a BGP router. The HBN solution encapsulates a set of network functions inside a container, which is deployed as a service pod on BlueField's Arm cores.

The HBN solution packages a set of network functions inside a container which, itself, is packaged as a service pod to be run on BlueField Arm. At the core of HBN is the Linux networking BlueField acceleration driver Netlink-to-DOCA, or nl2docad. This daemon seamlessly accelerates Linux networking using DOCA APIs to program specific packet processing rules in BlueField hardware.

The driver mirrors the Linux kernel routing and bridging tables into the BlueField hardware tables by discovering the configured Linux networking objects using the Linux Netlink API. Dynamic network flows, as learned by the Linux kernel networking stack, are also programmed by the driver into BlueField hardware by listening to Linux kernel networking events.



The following diagram captures an overview of HBN and the interactions between various components of HBN.



- ifupdown2 is the interface manager which pushes all the interface related states to kernel
- The routing stack is implemented in FRR and pushes all the control states (EVPN MACs and routes) to kernel via netlink
- Kernel maintains the whole network state and relays the information using netlink. The kernel is also involved in the punt path and handling traffic that does not match any rules in the eSwitch.
- nl2docad listens for the network state via netlink and invokes the DOCA interface to accelerate the flows in BlueField hardware tables. nl2docad also offloads these flows to eSwitch.

8.9.2 Service Function Chaining

HBN is a "bump-in-the-wire" service and requires specific network configuration on BlueField called service function chaining (SFC). SFC configuration is used to redirect network traffic, which is originated from or forwarded to the host or BlueField itself via the HBN data plane.

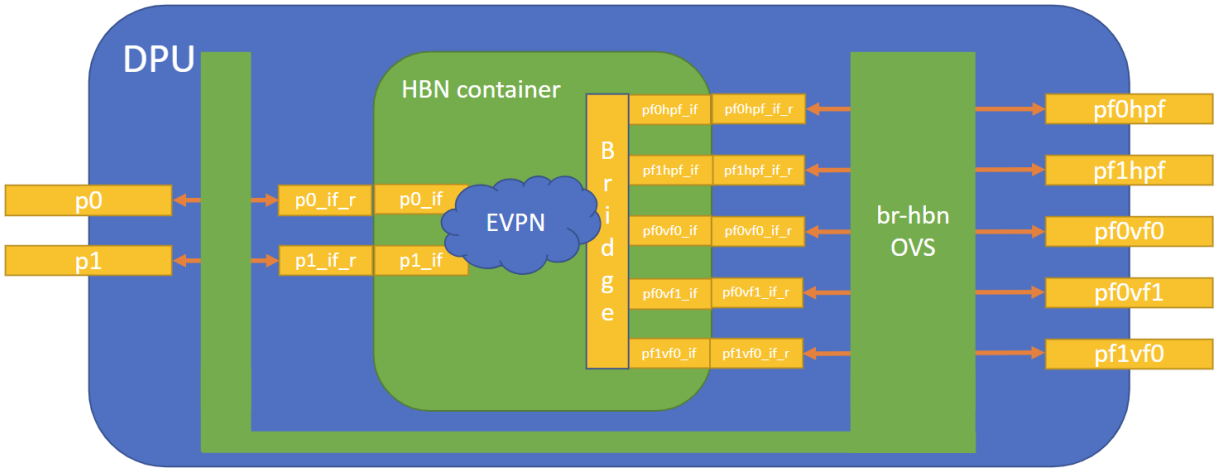
The diagram below shows the fully detailed default configuration for HBN with SFC.

In this setup, the HBN container is configured to use sub-function ports (SFs) instead of the actual uplinks, PFs and VFs. To illustrate, for example:

- Uplinks – use **p0_if** instead of **p0**
- PF – use **pf0hpf_if** instead of **pf0hpf**
- VF – use **pf0vf0_if** instead of **pf0vf0**

The indirection layer between the SF and the actual ports is managed via a **br-hbn** OVS bridge automatically configured when the BFB image is installed on BlueField with HBN enabled. This

indirection layer allows other services to be chained to existing SFs and provide additional functionality to transit traffic.



8.9.3 HBN Service Release Notes

The following subsections provide information about new features, interoperability, known issues, and bug fixes.

8.9.3.1 Changes and New Features

HBN 3.5.0 offers the following new features and updates:

- Upgraded FRR to version 10.03.
- Added support for B3220-SH Storage Controller
- Introduced resource scaling profiles to optimize CPU and memory usage. These profiles define specific resource limits based on scale targets, enabling efficient coexistence with other DPU services on NVIDIA® BlueField®-2 and NVIDIA® BlueField®-3 platforms.

> ⚠ Starting with HBN 3.5.0, the HBN container will fail to start if it detects an invalid startup configuration, rather than falling back to the last known good configuration. This validation occurs exclusively during container startup. You can find detailed error information (including the specific configuration that failed validation) in the container logs.

You can upgrade to this version from the previous HBN version.

8.9.3.2 Supported Platforms and Interoperability

8.9.3.2.1 Supported BlueField Networking Platforms

This version has been validated on the following NVIDIA BlueField networking platforms:

- BlueField-2 DPUs:

        - BlueField-2 P-Series DPU 25GbE Dual-Port SFP56; PCIe Gen4 x8; Crypto Enabled; 16GB on-board DDR; 1GbE OOB management; HHHL
        - BlueField-2 P-Series DPU 25GbE Dual-Port SFP56; integrated BMC; PCIe Gen4 x8; Secure Boot Enabled; Crypto Enabled; 16GB on-board DDR; 1GbE OOB management; FHHL
        - BlueField-2 P-Series DPU 25GbE Dual-Port SFP56; integrated BMC; PCIe Gen4 x8; Secure Boot Enabled; Crypto Enabled; 32GB on-board DDR; 1GbE OOB management; FHHL
        - BlueField-2 P-Series DPU 100GbE Dual-Port QSFP56; integrated BMC; PCIe Gen4 x16; Secure Boot Enabled; Crypto Enabled; 32GB on-board DDR; 1GbE OOB management; FHHL
    - BlueField-3 DPUs:
        - BlueField-3 B3210E E-Series FHHL DPU; 100GbE (default mode) / HDR100 IB; Dual-port QSFP112; PCIe Gen5.0 x16 with x16 PCIe extension option; 16 Arm cores; 32GB on-board DDR; integrated BMC; Crypto Enabled
        - BlueField-3 B3220 P-Series FHHL DPU; 200GbE (default mode)/NDR200 IB; Dual-port QSFP112; PCIe Gen5.0 x16 with x16 PCIe extension option; 16 Arm cores; 32GB on-board DDR; integrated BMC; Crypto Enabled
        - BlueField-3 B3240 P-Series Dual-slot FHHL DPU; 400GbE/NDR IB (default mode); Dual-port QSFP112; PCIe Gen5.0 x16 with x16 PCIe extension option; 16 Arm cores; 32GB on-board DDR; integrated BMC; Crypto Enabled
    - BlueField-3 SuperNICs:
        - BlueField-3 B3210L E-series FHHL SuperNIC, 100GbE (default mode)/HDR100 IB, Dual port QSFP112, PCIe Gen4.0 x16, 8 Arm cores, 16GB on-board DDR, integrated BMC, Crypto Enabled
        - BlueField-3 B3220L E-Series FHHL SuperNIC, 200GbE (default mode)/NDR200 IB, Dual-port QSFP112, PCIe Gen5.0 x16, 8 Arm cores, 16GB on-board DDR, integrated BMC, Crypto Enabled
        - BlueField-3 B3140L E-Series FHHL SuperNIC, 400GbE/NDR IB (default mode), Single-port QSFP112, PCIe Gen5.0 x16, 8 Arm cores, 16GB on-board DDR, integrated BMC, Crypto Enabled
        - BlueField-3 B3140H E-series HHHL SuperNIC, 400GbE (default mode)/NDR IB, Single-port QSFP112, PCIe Gen5.0 x16, 8 Arm cores, 16GB on board DDR, integrated BMC, Crypto Enabled

HBN does not support BlueField platforms with 8GB on-board DDR memory.

8.9.3.2.2 Supported BlueField OS

This version supports DOCA 3.5.0 (BSP 4.16.0) on Ubuntu 22.04 OS.

8.9.3.2.3 Verified Scalability Limits

This version has been validated by NVIDIA to sustain the following scalability limits:

These scalability limits are not necessarily architectural maximums. Individual limits are not guaranteed to be simultaneously achievable in a single deployment, and combined scale scenarios require validation against the intended topology.

Limit	Blue Field -2	Blue Field -3	Comments
VTEP peers (BlueFields per control plane) in the fabric	8K	16K	Number of BlueFields (VTEPs) within a single overlay fabric (reachable in the underlay).
L2 VNIs/Overlay networks per BlueField	20	20	Total number of L2 VNIs in the fabric for L2 VXLAN use-case assuming every interface is associated with its own VLAN + L2 VNI.
L3 VNIs/Overlay networks per BlueField	20 for up to 4K VTEPs 10 for up to 8K VTEPs	20 for up to 4K VTEPs 10 for up to 8K VTEPs 8 for up to 16K VTEPs	Total number of L3 VNIs in the fabric for L3 VXLAN use-case assuming every interface is associated with its own VLAN + L2 VNI + L3 VNI + VRF.
BlueFields per a single L2 VNI network	8K	16K	Total number of DPUs, configured with the same L2 VNI (3 real DPUs, 2000 emulated VTEPs).
BlueFields per a single L3 VNI network	8K	16K	Total number of DPUs, configured with the same L3 VNI (3 real DPUs, 2000 emulated VTEPs).
Maximum number of local MAC/ARP entries per BlueField	20	20	Max total number of MAC/ARP entries learned from the host on the DPU.
Maximum number of local BGP routes per BlueField	200	200	Max total number of BGP routes advertised by the host to the BlueField (BGP peering with the host): 100 IPv4 + 100 IPv6.
Maximum number of remote L3 LPM routes (underlay)	8K	18K	IPv4 or IPv6 underlay LPM routes per BlueField (default + host routes + LPM).
Maximum number of EVPN type-2 entries	16K	16K	Remote overlay MAC/IP entries for compute peers stored on a single BlueField (L2 EVPN).
Maximum number of EVPN type-5 entries	32K	128k[1]	Remote overlay L3 LPM entries for compute peers stored on a single BlueField (L3 EVPN).
Maximum next-hops per overlay ECMP next-hop group	64	64	Maximum number of next-hops in a single overlay ECMP group. This value is unrelated to host-side VF scale.
Maximum number of PFs on the host side	2	2	Total number of PFs visible to the host.
Maximum validated host-side VFs per BlueField (across host-side PFs; not per port)	16	16	Aggregate number of host-side VFs. Higher VF counts require further validation.
Maximum number of SFs on the BlueField side	2	2	Total number of SF devices created on BlueField Arm.

Limit	Blue Field -2	Blue Field -3	Comments
Maximum number of ACL capacity per BlueField	100	100	Total number of rules at `/32` granularity.
Maximum number of VRFs per BlueField	20	20	Total number of VRFs validated per DPU as part of L3 VNI scale testing. This does not imply validation of a 1:1 VF-to-VRF deployment.

[1] *ECMP convergence time may exceed 1 second for deployments with more than 96K type-5 routes.*

8.9.3.2.4 Performance Limitations

- ECMP convergence time may exceed 1 second for deployments configured with more than 96,000 type-5 routes. This observation applies to both IPv4 and IPv6 flows in dual-stack topologies.
- Deployments utilizing 96,000 routes or fewer will maintain standard convergence SLA times.

8.9.3.3 Known Issues

The following table lists known issues and limitations for this release of HBN.

Reference	Description
4369025	Description: When operating in DAL2OVS environments, IPv6 traffic may be blocked during VXLAN L3 native tenant encapsulated testing. Specifically, UDP IPv6 traffic sent over a VXLAN tunnel fails to reach the remote destination, resulting in packet reception timeouts and execution failures.
	Workaround: N/A
	Keyword: IPv6; VXLAN; DAL2OVS
	Reported in version: 3.4.0
4369016	Description: When operating in DAL2OVS environments, datapath flow rules may be missing for ingress IPv4 traffic during VXLAN L3 native tenant encapsulated testing. Additionally, Open vSwitch (`ovs-vswitchd`) logs may display DOCA hardware steering warnings related to failures in destroying groups (`err -16`).
	Workaround: N/A
	Keyword: IPv4; VXLAN; DAL2OVS; Datapath Flows
	Reported in version: 3.4.0
4414582	Description: In L3 EVPN scenarios with 16k overlay and 4k underlay routes, OVS may get stuck or abnormally terminate.

Reference	Description
	Workaround: Set: `ovs-vsctl set Open_vSwitch . other_config:doca-congestion-threshold=60` Keywords: L3 EVPN; 16k overlay; 4k underlay; OVS Reported in HBN Version: 2.5.0
4255661	Description: ARP packets between DPU and Outside Global which are forwarded using HBN are not hardware-offloaded. Workaround: N/A Keywords: ARP; outside world/global, HW offload Reported in HBN Version: 2.5.0
4255708	Description: When several ports are configured as part of a bridge and later reconfigured as L3 interfaces, only one port (the 1st port that was enslaved previously to bridge) is correctly reprogrammed as an L3 interface in nl2doca. The remaining ports continue to appear as bridged ports in nl2doca. Workaround: Restart the HBN container after unconfiguring the bridge ports and before reconfiguring them as L3 interfaces Keywords: Bridge port, L3 port Reported in HBN Version: 2.5.0
4193046	Description: When LLDP is enabled on BlueField, it might not work on uplink ports when the HBN service is running. This might happen if LLDP is running without an interface filter configuration. Workaround: Configure LLDP to run only on interfaces where LLDP is required, using a configuration file, **/etc/lldpd.d/ports.conf**, for the **lldpd** daemon. The interfaces can be specified using a regular expression, if required. For example: • To run LLDP only on the uplinks (**p0** and **p1**), configure it as follows: `$ cat /etc/lldpd.d/ports.conf` Configure system interface pattern **p[01]**. • To run LLDP on the uplinks plus some host-facing PFs or VFs, configure it as follows: `$ cat /etc/lldpd.d/ports.conf` Configure system interface pattern **p[0-1],pf[0-1]hpf,pf[0-1]vf[0-12]**. If this configuration file is changed while the LLDP service is running, it must be restarted using **systemctl restart lldpd**. Keywords: LLDP Reported in HBN version: 2.4.1

Reference	Description
4011688	Description: The following critical error message is generated during HBN POD reboot. It can be safely ignored. `Error message: "CRIT Server 'unix_http_server' running without any HTTP authentication checking"`
	Workaround: N/A
	Keywords: Log
	Reported in HBN version: 2.4.0
4098158	Description: When using default BGP timers, an OVS restart may lead to extended traffic loss due to BGP peering reset.
	Workaround: N/A
	Keywords: BGP; OVS
	Reported in HBN version: 2.4.0
3961387	Description: Changing the port number for the NVUE REST API using the NVUE CLI or API is not supported. Do not use the following command to change the port number: `nv set system api port <port-no>`
	Workaround: On HBN, NVUE is accessible through 8765 (the default port number).
	Keywords: NVUE API; port number
	Reported in HBN version: 2.3.0
3967748	Description: The command **nv show system api connections** does not return any data.
	Workaround: N/A
	Keywords: REST API; nginx
	Reported in HBN version: 2.3.0
3865633	Description: Host-generated UDP packets directed to VXLAN destination ports 4789 or 8472 are dropped if they lack a valid VXLAN encapsulation header. Properly encapsulated host-generated VXLAN traffic remains fully supported for L2 VNIs (across all HBN versions) and L3 VNIs (starting with HBN 3.0.0).
	Workaround: N/A
	Keywords: 4789, 8472
	Reported in HBN version: 2.2.0
3769309	Description: A ping (or other IP connectivity check) from a locally connected host in VRF-X to an interface IP address on the DPU/HBN itself in VRF-Y will not work, even if VRF route-leaking is enabled between these two VRFs.
	Workaround: N/A

Reference	Description
	Keyword: IP
	Reported in HBN version: 2.2.0
3835295	Description: Traffic entering the HBN service on a host PF/VF main-interface and exiting on a sub-interface of the same PF/VF (and vice-versa) is not hardware offloaded. Similarly, traffic entering HBN service on one sub-interface and exiting through another sub-interface of the same host PF/VF is also not hardware offloaded.
	Workaround: N/A
	Keyword: Hardware offload; interfaces
	Reported in HBN version: 2.2.0
3772552	Description: The DHCP relay gateway-interface IP address does not automatically pick up the IP address assigned to the associated VRF.
	Workaround: The gateway-interface IP address must be explicitly configured.
	Keyword: DHCP relay gateway; IP
	Reported in HBN version: 2.2.0

3757686	Description: When the HBN container is coming up and applying a large configuration through the NVUE startup service which includes entities used by DHCP relay (for example, interfaces, SVIs, and VRFs), the DHCP relay service may go into a **FATAL** state. It can be observed using the following command:
	`supervisorctl status | grep isc-dhcp-relay`
	`isc-dhcp-relay-vrf11 RUNNING pid 2069, uptime 0:11:31`
	`isc-dhcp-relay-vrf12 RUNNING pid 2071, uptime 0:11:31`
	`isc-dhcp-relay-vrf13 FATAL Exited too quickly (process log may have details)`
	`isc-dhcp-relay-vrf14 FATAL Exited too quickly (process log may have details)`

	Workaround: Restart the DHCP relay service that is in the **FATAL** state using the command: `supervisorctl restart <relay-service-name>`
	Keyword: DHCP relay; fatal; container; restart
	Reported in HBN version: 2.1.0
3605486	Description: When the DPU boots up after issuing a "reboot" command from the DPU itself, some host-side interfaces may remain down.

Reference	Description

	Workaround:
	1. Restart **openibd**:
	`systemctl restart openibd`
	2. Recreate SR-IOV interfaces, if required.
	3. Replay interface config. For example:
	• If using ifupdown2:
	`ifreload -a`
	• If using Netplan:
	`netplan apply`

	Keyword: Reboot
	Reported in HBN version: 1.5.0
3547103	Description: IPv6 stateless ACLs are not supported.
	Workaround: N/A
	Keyword: IPv6 ACL
	Reported in HBN version: 1.5.0
3339304	Description: Statistics for hardware-offloaded traffic are not reflected on SFs within an HBN container.
	Workaround: Look up the statistics using **ip -s link show** on PFs outside of the HBN container. PFs display Tx/Rx data for traffic that is hardware-accelerated in the HBN container.
	Keyword: Statistics; container
	Reported in HBN version: 1.4.0
3352003	Description: NVUE **show**, **config**, and **apply** commands malfunction if the **nvued** and **nvued-startup** services are not in the **RUNNING** and **EXITED** states, respectively.
	Workaround: N/A
	Keyword: NVUE commands
	Reported in HBN version: 1.3.0
3184745	Description: The command **nv show interface <intf> acl** does not display correct information if there are multiple ACLs bound to the interface.
	Workaround: Use the command **nv show interface <intf>** to view the ACLs bound to an interface.
	Keyword: ACLs
	Reported in HBN version: 1.2.0

Reference	Description
3158934	Description: Deleting an NVUE user by removing their password file and restarting the **decrypt-user-add** service on the HBN container does not work.
	Workaround: Either respawn the container after deleting the file or delete the password file corresponding to the user by running **userdel -r username**.
	Keyword: User deletion
	Reported in HBN version: 1.2.0
3185003	Description: When a packet is encapsulated with a VXLAN header, it adds extra bytes which may cause the packet to exceed the MTU of link. Typically, the packet would be fragmented but it is silently dropped without fragmentation.
	Workaround: Make sure that the MTU on the uplink port is always 50 bytes more than the host ports so that even after adding VXLAN headers, ingress packets do not exceed the MTU.
	Keyword: MTU; VXLAN
	Reported in HBN version: 1.2.0
3184905	Description: During VXLAN encapsulation, the DF flag is not propagated to the outer header. Such a packet may be truncated when forwarded in the kernel, and it may be dropped when the hardware is offloaded.
	Workaround: Make sure that the MTU on the uplink port is always 50 bytes more than host ports so that even after adding VXLAN headers, ingress packets do not exceed the MTU.
	Keyword: VXLAN
	Reported in HBN version: 1.2.0
3188688	Description: When stopping a container using the **crictl stop** command, you may receive an error. The command uses a timeout of 0 which is insufficient to stop all the processes in the HBN container.
	Workaround: Pass a greater timeout value when stopping the HBN container by running: `crictl stop --timeout 60 <hbn-container>`
	Keyword: Timeout
	Reported in HBN version: 1.2.0
3129749	Description: The same ACL rule cannot be applied in both the inbound and outbound direction on a port.
	Workaround: N/A
	Keyword: ACLs
	Reported in HBN version: 1.2.0
3126560	Description: The system's time zone cannot be modified using NVUE in the HBN container.

Reference	Description
	Workaround: The time zone can be manually changed by symlinking the **/etc/localtime** file to a binary time zone's identifier in the **/usr/share/zoneinfo** directory. For example: `sudo ln -sf /usr/share/zoneinfo/GMT /etc/localtime`
	Keyword: Time zone; NVUE
	Reported in HBN version: 1.2.0
3118204	Description: Auto-BGP functionality (where the ASN does not need to be configured but is dynamically inferred by the system based on the system's role as a leaf or spine device) is not supported on HBN.
	Workaround: If BGP is configured and used on HBN, the BGP ASN must be manually configured.
	Keyword: BGP
	Reported in HBN version: 1.2.0
3233088	Description: Since checksum calculation is offloaded to the hardware (and not by the kernel), you might see an incorrect checksum in the tcpdump for locally generated, outgoing packets. BGP keepalives and updates are examples of the packets that display incorrect checksums in tcpdump.
	Workaround: N/A
	Keyword: BGP
	Reported in HBN version: 1.2.0
2821785	Description: MAC addresses are not learned in the hardware but only in software. This may affect performance in pure L2 unicast traffic.
	Workaround: N/A
	Keyword: MAC; L2
	Reported in HBN version: 1.3.0
3017202	Description: Due to disabled backend foundation units, some NVUE commands return **500 INTERNAL SERVER ERROR** / **404 NOT FOUND**. These commands are related to features or subsystems which are not supported on HBN.
	Workaround: N/A
	Keyword: Unsupported NVUE commands
	Reported in HBN version: 1.3.0
2828838	Description: NetworkManager and other services not directly related to HBN may display the following message in syslog: `"netlink: read: too many netlink events. Need to resynchronize platform cache"` The message has no functional impact and can be safely ignored.
	Workaround: N/A
	Keyword: Error

Reference	Description
	Reported in HBN version: 1.3.0
4127609	Description: While the **bf.cfg** configuration file allows users to select the **flexible_profile** for HBN Resource Profiles, it does not support defining the specific resource parameters (such as CPU, Memory, and Huge Pages) directly within the configuration file. Workaround: To use a flexible profile, set the profile name to **flexible_profile** in **bf.cfg** or **hbn.conf**. Then, manually configure the custom resource values by editing the **flexible_profile.yaml** file (located at **/etc/mellanox/hbn_profiles/flexible_profile.yaml**)

8.9.3.4 Bug Fixes

The following table lists the known issues which have been fixed for this release of HBN.

Reference	Description
5237005	Description: The HBN Service Configuration guide incorrectly instructed users to bind stateful ACLs to a switched virtual interface (SVI), which is unsupported with OVS connection tracking.
	Fixed in HBN Version: 3.5.0
5190786	Description: With connection-tracking hardware offload and OVN-Kubernetes, TCP traffic between host physical functions (PFs) could stop after one or two days even though ICMP traffic continued to work.
	Fixed in HBN Version: 3.5.0
5184909	Description: On DPUs managed by NVIDIA DOCA™ Platform Framework, AppArmor could prevent **rsyslog** from rotating HBN container logs, causing logging to stop or the service state to appear as **UNKNOWN**.
	Fixed in HBN Version: 3.5.0
5181627	Description: PXE HTTP file downloads on BlueField-2 devices could be significantly slower than on BlueField-3 devices.
	Fixed in HBN Version: 3.5.0
5149884	Description: When creating a subinterface with NVUE, the configured base interface could be replaced with a name derived from the subinterface, leaving the subinterface down.
	Fixed in HBN Version: 3.5.0
5107681	Description: After a DPU reboot or state change, the HBN static pod could remain in **init-sfs** when **p0_if** was missing, preventing HBN startup and host HTTP boot.
	Fixed in HBN Version: 3.5.0

Reference	Description
5091576	Description: When the same neighbor IP address was reachable through multiple interfaces, `nl2docad` could enter a continuous neighbor resynchronization loop, causing high CPU usage and traffic loss.
	Fixed in HBN Version: 3.5.0
5075105	Description: `nl2docad` could repeatedly log `hal brmac tunnel dest ip not set` messages.
	Fixed in HBN Version: 3.5.0
5015230	Description: Memory could leak when adding or updating virtual MAC bridge rules for an SVI.
	Fixed in HBN Version: 3.5.0
4985898	Description: If a flexible resource profile specified more HBN CPU cores than the system provided, the HBN container could fail to start instead of using the default profile.
	Fixed in HBN Version: 3.5.0
4982836	Description: Disabling the IPv4 unicast address family for a dynamic BGP peer-group member through NVUE had no effect, and the peer continued to exchange IPv4 unicast routes.
	Fixed in HBN Version: 3.5.0
4867417	Description: During high flow churn, hardware-offload counter resources might not be released for reuse, causing OVS split-flow creation failures.
	Fixed in HBN Version: 3.5.0
4562928	Description: Disabling a meter or policer could cause `nl2docad` to repeatedly log meter statistics query errors.
	Fixed in HBN Version: 3.5.0
5140312	Description: On BlueField-3 E-Series JBOF systems, HBN may not initialize after a cold reboot even when enabled, leaving HBN networking unavailable.
	Fixed in HBN Version: 3.5.0
4667419	Description: On systems running the Ubuntu 24.04 BFB image, restarting the HBN container may fail. This can result in `containerd` service errors or internal assertion failures (e.g., in the `nl2doca` service), preventing the container from functioning correctly.
	Fixed in HBN Version: 3.5.0
4924262	Description: During an SSH login, users may encounter a broadcast error message from `systemd-journald` stating, `Failed to allocate manager object: Permission denied`. This error is a superficial permission warning and has no functional impact on system stability or overall operations.
	Fixed in HBN Version: 3.5.0
4991261	Description: After a VRF L3 VNI is changed and then reverted, VXLAN traffic may continue to use the previous stale VNI. FRR/zebra may show the correct VRF VNI, but `nl2doca` may continue programming VXLAN encapsulation with the old VNI. In az51, this can cause traffic to use the VPC-specific VNI instead of VNI `3109`.
	Fixed in HBN Version: 3.5.0

Reference	Description
4835913	Description: When the HBN container undergoes a forced restart (e.g., during a sudden host reboot), the standard graceful cleanup procedure may fail to execute. This failure leaves stale rules in the host's `iptables`. When the container subsequently restarts, these unexpected legacy rules conflict with the new setup, breaking the container's network configuration and routing.
	Fixed in HBN Version: 3.4.0
4539931	Description: Uni-directional UDP traffic matching a Dynamic DNAT rule fails to egress the DPU. Although connection tracking entries are created, traffic is not forwarded until reverse-direction traffic is detected.
	Fixed in HBN Version: 3.4.0
4774192	Description: Router MACs are deleted continuously when L3 EVPN is configured.
	Fixed in HBN Version: 3.3.0
3743942	Description: The HBN container may hang in `init-sfs` during a container restart when the HBN YAML file (`/etc/kubelet.d/doca_hbn.yaml`) is modified while the container is running.
	Fixed in HBN Version: 3.1.0
4403853	Description: After getting DHCP lease we are not restarting ntpsec@mgmt service therefore updated configuration received from DHCP lease is not getting applied
	Fixed in HBN Version: 3.1.0
4443601	Description: Fixed a firmware issue where PXE failed to boot when both LAG ports were up.
	Fixed in HBN Version: 3.1.0
4418454	Description: The `bf.cfg` file contains configurations for two uplinks, but the DPU has only one uplink. Due to this misconfiguration, SFC initialization will stall, and the HBN container will not be able to start.
	Fixed in HBN Version: 3.1.0
3891542	Description: If an NVUE-based routing policy (route map) configuration is used to associate route target extended communities with an EVPN route, only one route target can be specified.
	Fixed in HBN Version: 3.0.0
4214631	Description: Packets with destination port 4789 coming from the host-side will be dropped if HBN is configured as L3-EVPN, leading to Customer/Tenant encapsulated VXLAN traffic drops in an L3-EVPN scenario. This prevents running VXLAN underlay over HBN VXLAN overlay in L3 EVPN scenarios.
	Fixed in HBN Version: 3.0.0
4200335	Description: Sometimes the DNS resolution might fail if `resolv.conf` is not updated with the proper name server, leading to a loss of OOB connectivity.
	Fixed in HBN Version: 3.0.0
4196880	Description: DHCP issues may lead to incomplete resolve.conf on the HBN container. The consequences can be DNS resolution failures and/or the hostname being set to 'localhost'.
	Fixed in HBN VersionL 3.0.0

Reference	Description
4264397	Description: OVS does not punt the IPv6 neighbor advertisements with unicast destination MAC address to the CPU; therefore, the endpoint MAC may not be learnt on the VTEP as long as the endpoint is silent (resulting in traffic towards endpoint to be software-forwarded). This is applicable only for absolutely silent end hosts which do not initiate any IPv6 neighbor solicitation messages. After the silent end host initiates the traffic, traffic will be hardware forwarded. This issue will persist only if the end points never initiate any traffic but only send IPv6 neighbor advertisements as a response to IPv6 neighbor solicitation (rare).
	Fixed in HBN Version: 3.0.0
4155959	Description: With uplinks in the **br-sfc** bridge, IPv6 traffic in the uplink-to-uplink direction results in an OVS crash, resulting in complete traffic drop.
	Fixed in HBN Version: 2.5.0
4197067	Description: The management VRF does not have an IPv6 address configured, resulting in the absence of a default IPv6 route in the management VRF. Consequently, IPv6 connectivity on the management port is unavailable and only IPv4 connectivity is supported.
	Fixed in HBN Version: 2.5.0
4093502	Description: VRF interfaces have a loopback address, but these loopback addresses have scope global, not scope host which can break source IP address lookup for packets originating from the VRF.
	Fixed in HBN version: 2.4.0
4029473	Description: Rarely, after deleting then re-creating an interface, BGP peering over that interface may announce IPv6 routes with an IPv4-mapped IPv6 address as the next hop, which the BGP peer device at the other end can reject.
	Fixed in HBN version: 2.4.0
4125363	Description: On newer BlueField-2 and BlueField-3 devices, **/sys/class/dmi/id/sys_vendor** shows **Nvidia** instead of https://www.mellanox.com , causing NVUE to fail to apply configurations.
	Fixed in HBN version: 2.4.0
3965589	Description: When SR-IOV VFs are created or deleted, then recreated, some ports may stay in ethX naming format and not be properly renamed to pfXvfY format. This results in the port remaining in an error state when running the command **ovs-vsctl show** due to the SFC and HBN not recognizing it.
	Fixed in HBN version: 2.4.0
4004191	Description: Due to security fixes on BlueField-2, the number of context switches increased by 20% which may result in user applications (for example, nl2doca) running slower.
	Fixed in HBN version: 2.4.0
3880352	Description: Deleting and re-adding SR-IOV ports might result in some ports in the **br-hbn** bridge going into an error state.
	Fixed in HBN version: 2.4.0
3960825	Description: When either **ENABLE_SFC_HBN** or **ENABLE_BR_HBN** is set to **yes** in **bf.cfg** , the initial DHCP request from **oob_net0** during the adapter boot does not contain the **NVIDIA/BF/OOB** string in DHCP option 60 (vendor class identifier).

Reference	Description
	Fixed in HBN version: 2.3.0
3538167	Description: An explicit restart of the FRR service may be required if the BGP AS number is changed via NVUE.
	Fixed in HBN version: 2.3.0
3360699	Description: To decrease the default MTU on HBN interfaces, you must make the change on both the BlueField and within HBN, then reboot the BlueField for the change to take effect.
	Fixed in HBN version: 2.3.0
3864080	Description: When an interface is toggled off and on, its sub-interfaces lose their IPv6 addresses and do not get them back.
	Fixed in HBN version: 2.3.0
3632344	Description: HBN interfaces on the BlueField side (outside the HBN container) may not receive their proper MTU set from `systemd-network`.
	Fixed in HBN version: 2.2.0
3760869	Description: Datapath flow with very low PPS may be deleted before aging time (60 sec) in environments with a large number of routes (16K+).
	Fixed in HBN version: 2.2.0
3770992	Description: It is not possible to configure an IPv6 default (`::/0`) static route using NVUE.
	Fixed in HBN version: 2.2.0
3824881	Description: When the number of unique ECMP groups exceeds 6, programming prefixes with more than 6 ECMP groups fails. Uniqueness is determined by the ECMP content, so multiple routes with the same nexthop paths use only one ECMP group.
	Fixed in HBN version: 2.2.0
3705894	Description: In an EVPN Symmetric Routing scenario, IPv6 traffic is not hardware-offloaded.
	Fixed in HBN version: 2.2.0
3519324	Description: The DOCA HBN container takes one minute longer to spawn, compared to the 1.4.0 HBN release.
	Fixed in HBN version: 2.1.0
3219539	Description: TC rules are set by OVS to map uplink and host representor ports to the HBN service. These rules can expire, so packets may need to be software-forwarded periodically to refresh them.
	Fixed in HBN version: 2.1.0
3610971	Description: The output of `nv show interface` command does not display information about VRFs, VXLAN, or bridges.
	Fixed in HBN version: 2.0.0

Reference	Description
3452914	Description: IPv6 OOB connectivity from the HBN container stops working if the **br-mgmt** interface on the DPU goes down. When going down, the **br-mgmt** interface loses its IPv6 address, which is used as the gateway address for the HBN container. If the **br-mgmt** interface comes back up, its IPv6 address is not added back and IPv6 OOB connectivity from the HBN container will not work.
	Fixed in HBN version: 1.5.0
3191433	Description: ECMP selection for the underlay path uses the ingress port and identifies uplink ports via round robin. This may not result in a uniform traffic distribution.
	Fixed in HBN version: 1.4.0
3049879	Description: When reloading (**ifreload**) an empty **/etc/network/interfaces** file, the previously created interfaces are not deleted.
	Fixed in HBN version: 1.4.0
3284607	Description: When an ACL is configured for IPv4 and L4 parameters (protocol tcp/udp, source, and destination ports) match, the ACL also matches IPv6 traffic with the specified L4 parameters.
	Fixed in HBN version: 1.4.0
3282113	Description: Some DPUs experience an issue with the clock settings after installing a BlueField OS in an HBN setting in which the date reverts back to "Thu Sep 8, 2022."
	Fixed in HBN version: 1.4.0
3354029	Description: If interfaces configured for BGP unnumbered peering are not defined in the **/etc/network/interfaces** file, BGP peering will not be established on them.
	Fixed in HBN version: 1.4.0

8.9.4 HBN Service Deployment

8.9.4.1 1. HBN Service Requirements

For HBN hardware and software requirements, refer to the HBN Service Release Notes.

The following subsections describe specific prerequisites for the BlueField before deploying the DOCA HBN Service.

8.9.4.1.1 1.1. Enabling BlueField DPU Mode

HBN requires BlueField to work in either DPU mode or zero-trust mode of operation. Information about configuring BlueField modes of operation can be found under "BlueField Modes of Operation".

8.9.4.1.2 1.2. Enabling SFC

HBN requires SFC configuration to be activated on the BlueField before running the HBN service container. SFC allows for additional services/containers to be chained to HBN and provides additional data manipulation capabilities. SFC can be configured in 3 modes:

1. HBN-only mode – In this mode, one OVS bridge is created, **br-hbn**. All HBN-specific ports are added to this bridge. This is the default mode of operation. This mode is configured by setting **ENABLE_BR_HBN=yes** in **bf.cfg** and leaving **ENABLE_BR_SFC** to default.
2. Dual bridge mode – In this mode, 2 OVS bridges are created, **br-hbn** and **bf-sfc**. All HBN-specific ports are added to **bf-sfc** bridge and all these ports are patched into the **br-hbn** bridge. **bf-sfc** can be used to add various custom steering flows to direct traffic across different ports in the bridge. In this mode, both **ENABLE_BR_SFC** and **ENABLE_BR_HBN** are set as to **yes**. **BR_HBN_XXX** parameters are not set and all ports are under **BR_SFC_XXX** variables.
3. Mixed mode – this is similar to the dual bridge model, except that ports can be assigned to either of the bridges (i.e., some ports in **br-hbn** and some in **br-sfc** bridge). In this mode, ports are under **BR_SFC_XXX** and **BR_HBN_XXX**.
   The use of the bridge **br-sfc** allows defining deployment-specific rules before or after HBN pipeline. User can add OpenFlow rules directly to **bf-sfc** bridge. If **ENABLE_BR_SFC_DEFAULT_FLOWS** is set to **yes**, make sure user rules are inserted at higher priority to make it effective.

The following table describes various **bf.cfg** parameters used to configure these modes as well as other parameters which assign ports to various bridges:

Parameter	Description	Mandatory	Default Value	Example
ENABLE_BR_HBN	Setting this parameter to **yes** enables the **br-hbn** bridge ⚠ This setting is necessary to work with HBN.	Yes	**no**	`ENABLE_BR_HBN=yes`
ENABLE_BR_SFC	Setting this parameter to **yes** enables the **br-sfc** bridge ℹ This is only needed when the second OVS bridge is required for custom steering flows.	No	**no**	`ENABLE_BR_SFC=no`
BR_HBN_UPLINKS	Uplinks added to **br-hbn** directly	No	**p0,p1**	`BR_HBN_UPLINKS="p0,p1"`

Parameter	Description	Mandatory	Default Value	Example
`BR_SFC_UPLINKS`	Uplinks added to `br-sfc` directly	No	`""`	`BR_SFC_UPLINKS=""`
`BR_HBN_REPS`	PFs and VFs added to `br-hbn` directly	No	`""`	`BR_HBN_REPS="pf0hpf,pf1hpf,pf0vf0-pf0vf12,pf1vf0-pf1vf4"`
`BR_SFC_REPS`	PFs and VFs added to `br-sfc` directly	No	`""`	`BR_SFC_REPS=""`
`BR_HBN_SFS`	DPU ports added to `br-hbn` directly. These ports are mostly service ports present on the DPU which require using HBN network offload services.	No	`""`	`BR_HBN_SFS=svc1,svc2`
`BR_SFC_SFS`	DPU ports added to `br-sfc` directly	No	`""`	`BR_SFC_SFS=svc1,svc2`
`BR_HBN_SFC_PATCH_PORTS`	Patch ports added to `br-sfc` . These are general purpose ports meant for muxing or demuxing of traffic across various PF/VF ports.	No		`BR_HBN_SFC_PATCH_PORTS=patch1`
`LINK_PROPAGATION`	Mapping of how link propagation should work. If nothing is provided, each uplink/PF/VF port reflects its status in its corresponding HBN port. For example, the status of p0 is reflected in `p0_if` .	No	Uplink/PF/VF to the corresponding HBN port	`LINK_PROPAGATION=""`
`ENABLE_BR_SFC_DEFAULT_FLOWS`	Provides default connectivity in the `br-sfc` bridge so each port can send traffic to its corresponding output port	No	`no`	`ENABLE_BR_SFC_DEFAULT_FLOWS=yes`
`HUGEPAGE_SIZE`	Sets the huge page size for DOCA-OVS to use. Only 2048 is tested for HBN, which is also the default value.	No	`2048`	`HUGEPAGE_SIZE=2048`

Parameter	Description	Mandatory	Default Value	Example

HUGEPAGE_COUNT	Sets the number of huge pages needed for the HBN application. The current default limit is 2048. When HBN is running with other services, this must be modified accordingly.
	The recommended upper limit for memory allocated to huge pages for HBN depends on the total memory available on the BlueField-3 DPU (assuming the default HUGEPAGE_SIZE of 2048 kB).
	• BlueField-3 with 16GB Memory:
	  • Recommended Limit: 5GB
	  • Configuration: HUGEPAGE_COUNT=2560
	• BlueField-3 with 32GB Memory:
	  • Recommended Limit: 10GB
	  • Configuration: HUGEPAGE_COUNT=5120
		No	2048	HUGEPAGE_COUNT=2048

Refer to section "HBN Deployment Configuration" for more information about port connectivity in each mode.

The following subsections provide additional information about SFC and instructions on enabling it during BlueField DOCA image installation.

8.9.4.1.2.1 1.2.1. Deploying BlueField DOCA Image with SFC from Host

For DOCA image installation on BlueField, the user should follow the instructions under *DOCA Installation Guide for Linux* with the following extra notes to enable BlueField for HBN setup:

1. Make sure link type is set to ETH under the "Installing Software on Host" section.
2. Add the following parameters to the bf.cfg configuration file:
   a. This configuration example is relevant for "HBN-only mode". Set the appropriate variables and values depending on your deployment model.
   b. Enable HBN specific OVS bridge on BlueField Arm by setting ENABLE_BR_HBN=yes.
   c. Define the uplink ports to be used by HBN BR_HBN_UPLINKS='<port>'.

Must include both ports (i.e., p0,p1) for dual-port BlueField devices and only p0 for single-port BlueField devices.

d. Include PF and VF ports to be used by HBN. The following example sets both PFs and 8 VFs on each uplink: **BR_HBN_REPS='pf0hpf,pf1hpf,pf0vf0-pf0vf7,pf1vf0-pf1vf7'**.

e. (Optional) Include SF devices to be created and connected to HBN bridge on the BlueField Arm side by setting **BR_HBN_SFS='pf0dpu1,pf0dpu3'**.

> If nothing is provided, **pf0dpu1** and **pf0dpu3** are created by default.

> While older formats of **bf.cfg** still work in this release, they will be deprecated over the next 2 releases. So, its advisable to move to the new format to avoid any upgrade issues in future releases. The following is an example for the old **bf.cfg** format:
>
> ```
> ENABLE_SFC_HBN=yes
> NUM_VFs_PHYS_PORT0=12 # <num VFs supported by HBN on Physical Port 0> (valid range:
> 0-127) Default 14
> NUM_VFs_PHYS_PORT1=2  # <num VFs supported by HBN on Physical Port 1> (valid range:
> 0-127) Default 0
> ```

3. Then run:

```
bfb-install -c bf.cfg -r rshim0 -b <BFB-image>
```

4. Once SFC deployment is done, it creates 3 set of files:
   - **/etc/mellanox/hbn.conf** – this file can be used to redeploy SFC without the need to pass through **bf.cfg** again to modify interface mapping
   - **/etc/mellanox/sfc.conf** – this file provides a view of how various ports are connected in different bridges
   - **/etc/mellanox/mlnx-sf.conf** – this file includes all the HBN ports to be created and corresponding commands to create the port

8.9.4.1.2.2 1.2.2. Deploying BlueField DOCA Image with SFC Using PXE Boot

To enable HBN SFC using a PXE installation environment with BFB content, use the following configuration for PXE:

```
bfnet=<IFNAME>:<IPADDR>:<NETMASK> or <IFNAME>:dhcp
bfks=<URL of the kickstart script>
```

The kickstart script (bash) should include the following lines:

```
cat >> /etc/bf.cfg << EOF

ENABLE_BR_HBN=yes
BR_HBN_UPLINKS='p0,p1'
BR_HBN_REPS='pf0hpf,pf1hpf,pf0vf0-pf0vf7,pf1vf0-pf1vf7'
BR_HBN_SFS='pf0dpu1,pf0dpu3'
EOF
```

The **/etc/bf.cfg** generated above is sourced by the BFB **install.sh** script.

It is recommended to verify the accuracy of the BlueField's clock post-installation. This can be done using the following command:

```
$ date
```

Please refer to the known issues listed in the "DOCA Release Notes" for more information.

8.9.4.1.2.3 1.2.3. Redeploying SFC from BlueField

Redeploying SFC from BlueField can be done after the DPU has already been deployed using **bf.cfg** and either port mapping or bridge configuration needs to be change.

To redeploy SFC from BlueField:

1. Edit **/etc/mellanox/hbn.conf** by adding or removing entries in each segment as necessary.
2. Rerun the SFC install script:

   ```
   /opt/mellanox/sfc-hbn/install.sh -c -r
   ```

   This generates a new set of **sfc.conf** and **mlnx-sf.conf** and reloads the DPU.
   Configuration and reload can be split into 2 steps by removing the **-r** option and rebooting BlueField post configuration.

After the BlueField reloads, the command **ovs-vsctl show** should show all the new ports and bridges configured in OVS.

8.9.4.1.2.4 1.2.4. Deploying HBN with Other Services

When the HBN container is deployed by itself, BlueField Arm is configured with 3k huge pages. If it is deployed with other services, the actual number of huge-pages must be adjusted based on the requirements of those services. For example, SNAP or NVMesh may need approximately 1k to 5k huge pages. So, if HBN is running with either of these services on the same BlueField, the total number of hugepages must be set to the sum of the hugepage requirement of all the services.

For example, if NVMesh needs 3k hugepages, 6k total hugepages must be set when running with HBN. To do that, add the following parameters to the **bf.cfg** configuration file alongside other desired parameters.

```
HUGEPAGE_COUNT=6144
```

Memory Requirements

These configurations must only be applied to BlueField-3 platforms equipped with 32GB of onboard RAM. Attempting these settings on 16GB systems may result in memory exhaustion or instability for services running on the BlueField Arm cores.

Service Coexistence Limits

When running HBN alongside other DOCA services, the system is qualified to support a maximum of 16 VFs.

8.9.4.2 2. Scaling Profiles

To address the scaling needs of Network Cloud Providers (NCPs) while optimizing DPU resources (CPU and memory) and enabling coexistence with other DPU services (such as SNAP, AppShield, or OVN), HBN provides pre-defined Scaling Profiles. These profiles specify resource limits for each scale target to ensure balanced performance and efficient consumption.

The following table details the available profiles and their corresponding resource limits based on the DPU platform and memory configuration.

Profiles are strictly hardware-dependent. Attempting to apply a BlueField-3 specific profile (e.g., 32GB RAM variants) to BlueField-2 hardware is not supported. If an incompatible profile is selected, the system logs an error and automatically defaults to the `rp_4k_16k` profile to maintain system stability.

8.9.4.2.1 2.1. Scaling Profiles and Resource Limits

Profile Name	Supported Platforms	DPU VTEPs	Underlay Routes	L2 EVPN Routes	L3 EVPN Routes	VRFs / L3 VNIs	Max CPU (HBN)	Max CPU (OVS)	Max RAM (HBN)	Max RAM (OVS)
`default`	• BlueField-2 (16GB/32GB) • BlueField-3 (16GB/32GB)	4K	6K	16K	16K	20	1	1	3GB	3GB
`rp_4k_16k`	• BlueField-2 (16GB/32GB) • BlueField-3 (16GB/32GB)	4K	6K	16K	16K	20	1	1	3GB	3GB
`rp_8k_40k`	BlueField-2 (16GB/32GB)	8K	10K	16K	40K	10	2	1	4GB	4GB
`rp_8k_80k`	BlueField-3 (32GB)	8K	10K	16K	80K	10	1	1	4GB	4GB
`rp_16k_128k`	BlueField-3 (32GB)	16K	18K	16K	128K	5	2	2	6GB	6GB

8.9.4.2.2 2.2. Profile Configuration

8.9.4.2.2.1 2.2.1. Initial Setup

The profile is initially selected via the **bf.cfg** configuration file using the parameter **HBN_PROFILE**. The naming convention follows the format **rp_<vtep_num>_<l3_routes>**.

- Default behavior: If no profile is selected, the **default** profile is applied (equivalent to **rp_4K_16K**).
- Invalid profile: If a user selects a non-existent profile, the error is logged and the system defaults to **default**.
- Flexible profile: If **HBN_PROFILE=flexible_profile** is selected in **bf.cfg**, you must also provide valid values for all required flexible profile parameters in **bf.cfg**: **HBN_CPU_CORE**, **HBN_RAM_MEMORY**, and **OVS_CPU_CORE**.

8.9.4.2.2.2 2.2.2. Flexible Profile

Users requiring custom resource allocation can use the flexible profile.

1. Set the profile name to **flexible_profile**.
2. Edit the **/etc/mellanox/hbn_profiles/flexible_profile.yaml** file to define custom values for **hbn_cpu**, **memory**, and **ovs_cpu**.

   Unlike standard profiles, flexible mode supports fractional values (e.g., **HBN_CPU_CORE="0.5"**).

   When using the **flexible_profile**, you must explicitly configure all required parameters. If any parameter is missing or unused, the system will automatically fall back to the **default** profile.

8.9.4.2.3 2.3. Changing Profiles

To switch profiles on a running system, update the **hbn.conf** file.

- File location: **hbn.conf** is found under **/etc/mellanox/hbn.conf**.

  Do not edit **resources_profiles.yaml** directly as it is not editable. Instead, update the profile name in **hbn.conf** only.

- Effect: The **hbn-profile.path** unit monitors this file. When a change is detected, it triggers **hbn-profile.service**. This service retrieves values from either **resources_profiles.yaml** (for standard profiles) or **flexible_profile.yaml** (for flexible profiles) and applies them.

  Container Restart

If the HBN container does not restart automatically after the profile change, it must be restarted manually.

OVS Restart

Changes to `OVS_CPU` settings require a manual OVS restart for the configuration to take effect. Use the command `systemctl restart openvswitch-switch.service`.

8.9.4.2.4 2.4. System Behavior and Services

8.9.4.2.4.1 2.4.1. HBN Profile Service

The `hbn-profile.service` manages resource application. The `hbn-profile.path` unit monitors `hbn.conf`, `flexible_profile.yaml` and the HBN container YAML file `/etc/kubelet.d/doca_hbn.yaml` using `inotify` mechanisms.

Overwrite Protection

If a user manually updates the container YAML file, the service will overwrite those changes with the values defined in `hbn.conf`.

8.9.4.2.4.2 2.4.2. OVS Resource Limitation

This feature also limits resources for OVS to ensure accurate consumption estimates.

- CPU: The service configures CPU core limits via the OVS CLI using `other_config:pmd-cpu-mask`.
- Memory: Huge pages are allocated according to the profile.

8.9.4.3 3. Launching HBN Service

8.9.4.3.1 3.1. HBN Service Container Deployment

HBN service is available on NGC, NVIDIA's container catalog. For information about the deployment of DOCA containers on top of the BlueField, refer to NVIDIA DOCA Container Deployment Guide.

8.9.4.3.1.1 3.1.1. Downloading HBN Container Resource File

Pull the latest DOCA container resource as a `*.zip` file from NGC and extract it to the `<resource>` folder. The HBN resource from NGC is available here

8.9.4.3.1.2 3.1.2. Running HBN Preparation Script

The HBN script (`hbn-dpu-setup.sh`) performs the following steps on BlueField Arm which are required for HBN service to run:

1. Sets up interface MTU if needed.
2. Sets up mount points between BlueField Arm and HBN container for logs and configuration persistency.
3. Sets up various paths as needed by supervisord and other services inside container.
4. Enables the REST API access if needed.
5. Creates or updates credentials

The script is located in **<resource>/scripts/doca_hbn/<hbn_version>/** folder, which is downloaded as part of the DOCA Container Resource.

> **Optional**
>
> To achieve the desired configuration on HBN's first boot, before running preparation script, users can update default NVUE or flat (network interfaces and FRR) configuration files, which are located in **<resource>/scripts/doca_hbn/<hbn_version>/** .
>
> - For NVUE-based configuration:
>   - **etc/nvue.d/startup.yaml**
> - For flat-files based configuration:
>   - **etc/network/interfaces**
>   - **etc/frr/frr.conf**
>   - **etc/frr/daemons**

Run the following commands to execute the **hbn-dpu-setup.sh** script:

```
cd <resource>/scripts/doca_hbn/2.4.0/
chmod +x hbn-dpu-setup.sh
sudo ./hbn-dpu-setup.sh
```

The following is the help menu for the **hbn-dpu-setup.sh** script:

```
./hbn-dpu-setup.sh -h
usage: hbn-dpu-setup.sh
hbn-dpu-setup.sh -m|--mtu <MTU> Use <MTU> bytes for all HBN interfaces (default 9216)
hbn-dpu-setup.sh -u|--username <username> User creation
hbn-dpu-setup.sh -p|--password <password> Password for --username <username>
hbn-dpu-setup.sh -H|--hashed_password <hashed password> Password for --username <username>
hbn-dpu-setup.sh -e|--enable-rest-api-access Enable REST API from external IPs
hbn-dpu-setup.sh -h|--help
```

3.1.2.1. Enabling REST API Access

To enable the REST API access:

1. Change the default password for the **nvidia** username:
   a. You can either use this method to provide an already hashed password (recommended):

   ```
   ./hbn-dpu-setup.sh -u nvidia -H <new-hahsed-password>
   ```

   b. Or provide a plain text password using this method (to be deprecated soon):

   ```
   ./hbn-dpu-setup.sh -u nvidia -p <new-password>
   ```

2. Enable REST API:

```
./hbn-dpu-setup.sh --enable-rest-api-access
```

3. Perform BlueField system-level reset.

8.9.4.3.1.3 3.1.3. Spawning HBN Container

HBN container **.yaml** configuration is called **doca_hbn.yaml** and it is located in **<resource>/configs/<doca_version>/** directory. To spawn the HBN container, simply copy the **doca_hbn.yaml** file to the **/etc/kubelet.d** directory:

```
cd <resource>/configs/2.9.0/
sudo cp doca_hbn.yaml /etc/kubelet.d/
```

Kubelet automatically pulls the container image from NGC and spawns a pod executing the container. The DOCA HBN Service starts executing right away.

8.9.4.3.1.4 3.1.4. Verifying HBN Container is Running

To inspect the HBN container and verify if it is running correctly:

1. Check HBN pod and container status and logs:
   a. Examine the currently active pods and their IDs (it may take up to 20 seconds for the pod to start):

   ```
   sudo crictl pods
   ```

   b. View currently active containers and their IDs:

   ```
   sudo crictl ps
   ```

   c. Examine logs of a given container:

   ```
   sudo crictl logs
   ```

   d. Examine kubelet logs if something did not work as expected:

   ```
   sudo journalctl -u kubelet@mgmt
   ```

2. Log into the HBN container:

```
sudo crictl exec -it $(crictl ps | grep hbn | awk '{print $1;}') bash
```

3. While logged into HBN container, verify that the **frr**, **nl2doca**, and **neighmgr** services are running:

```
(hbn-container)$ supervisorctl status frr
(hbn-container)$ supervisorctl status nl2doca
(hbn-container)$ supervisorctl status neighmgr
```

4. Users may also examine various logs under **/var/log** inside the HBN container.

8.9.4.3.2 3.2. HBN Deployment Configuration

The HBN service comes with four types of configurable interfaces:

- Two uplinks (**p0_if** , **p1_if**)
- Two PF port representors (**pf0hpf_if** , **pf1hpf_if**)
- User-defined number of VFs (i.e., **pf0vf0_if** , **pf0vf1_if** , ..., **pf1vf0_if** , **pf1vf1_if** , ...)
- DPU interfaces to connect to services running on BlueField, outside of the HBN container (**pf0dpu1_if** and **pf0dpu3_if**)

The ***_if** suffix indicates that these are sub-functions and are different from the physical uplinks (i.e., PFs, VFs). They can be viewed as virtual interfaces from a virtualized BlueField.

Each of these interfaces is connected outside the HBN container to the corresponding physical interface, see section "Service Function Chaining" (SFC) for more details.

The HBN container runs as an isolated namespace and does not see any interfaces outside the container (**oob_net0** , real uplinks and PFs, ***_if_r** representors).

8.9.4.3.2.1 3.2.1. HBN-only Deployment Configuration

This is the default deployment model of HBN. In this model, only one OVS bridge is created.

The following is a sample **bf.cfg** and the resulting OVS and port configurations:

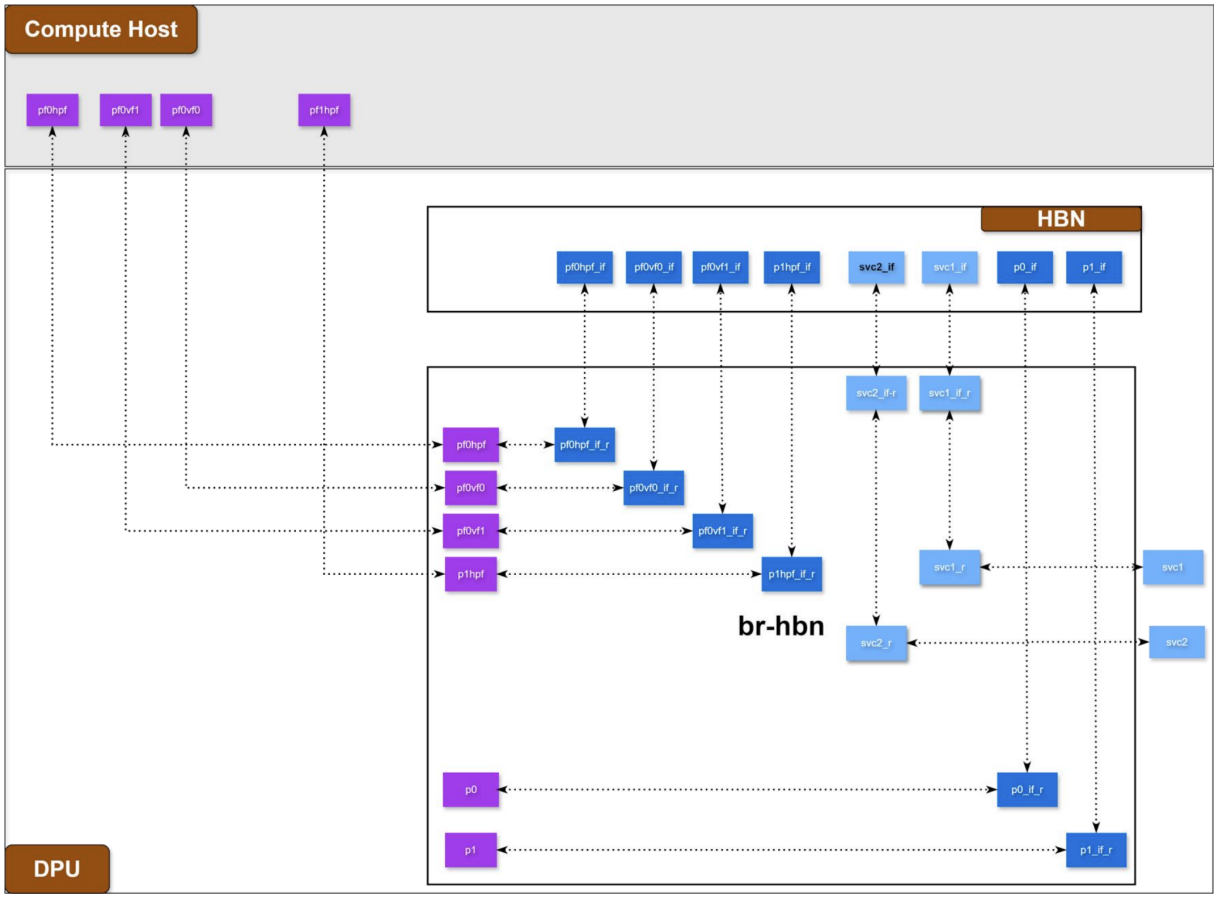
- Sample **bf.cfg** :
  bf.cfg

```
BR_HBN_UPLINKS="p0,p1"
BR_HBN_REPS="pf0hpf,pf1hpf,pf0vf0-pf0vf12,pf1vf0-pf1vf1"
BR_HBN_SFS="svc1,svc2"
```

- Generated **hbn.conf** :
  Generated hbn.conf

```
[BR_HBN_UPLINKS]
p0
p1
[BR_HBN_REPS]
pf0hpf
pf0vf0
pf0vf1
pf0vf2
pf0vf3
pf0vf4
pf0vf5
pf0vf6
pf0vf7
pf0vf8
pf0vf9
pf0vf10
pf0vf11
pf0vf12
pf1hpf
pf1vf0
pf1vf1

[BR_HBN_SFS]
svc1
svc2

[BR_SFC_UPLINKS]


[BR_SFC_REPS]

[BR_SFC_SFS]

[BR_HBN_SFC_PATCH_PORTS]
```

```
[LINK_PROPAGATION]
p0:p0_if_r
p1:p1_if_r
pf0hpf:pf0hpf_if_r
pf0vf0:pf0vf0_if_r
pf0vf1:pf0vf1_if_r
pf0vf2:pf0vf2_if_r
pf0vf3:pf0vf3_if_r
pf0vf4:pf0vf4_if_r
pf0vf5:pf0vf5_if_r
pf0vf6:pf0vf6_if_r
pf0vf7:pf0vf7_if_r
pf0vf8:pf0vf8_if_r
pf0vf9:pf0vf9_if_r
pf0vf10:pf0vf10_if_r
pf0vf11:pf0vf11_if_r
pf0vf12:pf0vf12_if_r
pf1hpf:pf1hpf_if_r
pf1vf0:pf1vf0_if_r
pf1vf1:pf1vf1_if_r
svc1_r:svc1_if_r
svc2_r:svc2_if_r

[ENABLE_BR_SFC]

[ENABLE_BR_SFC_DEFAULT_FLOWS]
```



8.9.4.3.2.2 3.2.2. Dual Bridge HBN Deployment Configuration

The following is a sample **bf.cfg** and the resulting OVS and port configurations:

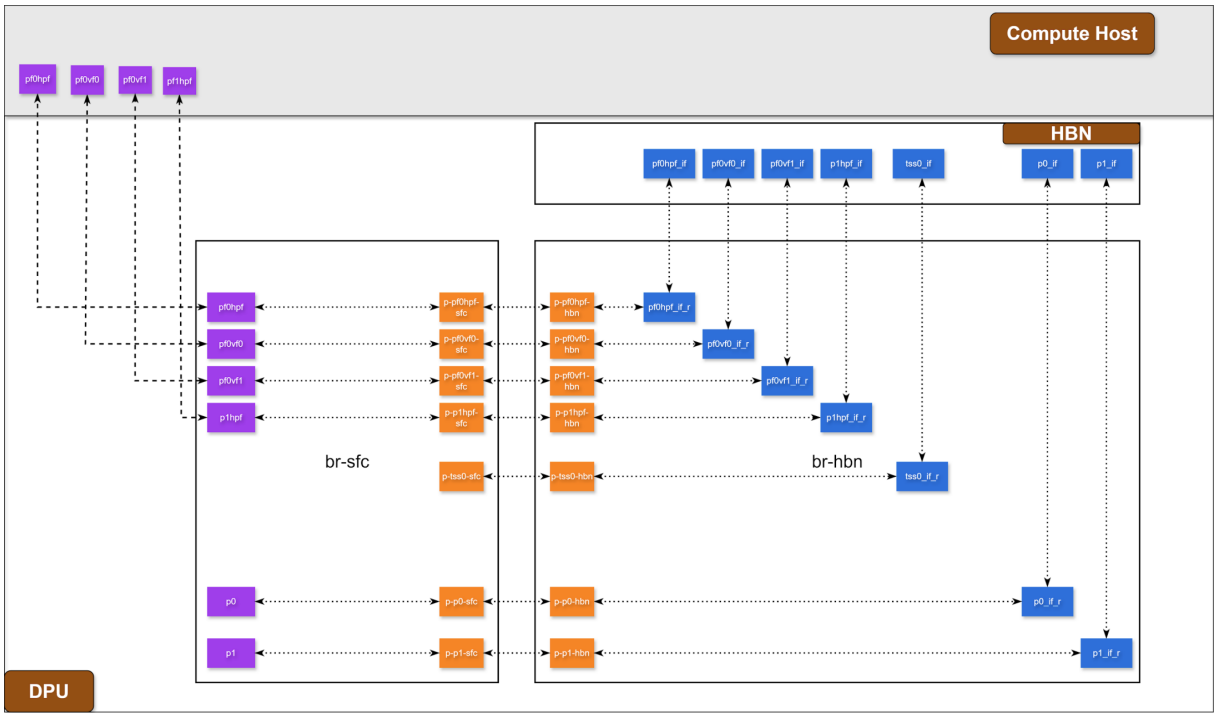
- Sample **bf.cfg**:
  bf.cfg

```
BR_HBN_UPLINKS=""
```

```
BR_SFC_UPLINKS="p0,p1"
BR_HBN_REPS=""
BR_SFC_REPS="pf0hpf,pf1hpf,pf0vf0-pf0vf1,pf1vf0-pf1vf1"
BR_HBN_SFS=""
BR_SFC_SFS=""
BR_HBN_SFC_PATCH_PORTS="tss0"
LINK_PROPAGATION="pf0hpf:tss0"
ENABLE_BR_SFC=yes
ENABLE_BR_SFC_DEFAULT_FLOWS=yes
```

- Generated **hbn.conf**:
  Generated hbn.conf

```
[BR_HBN_UPLINKS]


[BR_HBN_REPS]


[BR_HBN_SFS]


[BR_SFC_UPLINKS]
p0
p1

[BR_SFC_REPS]
pf0hpf
pf0vf0
pf0vf1
pf1hpf
pf1vf0
pf1vf1

[BR_SFC_SFS]


[BR_HBN_SFC_PATCH_PORTS]
tss0

[LINK_PROPAGATION]
pf0hpf:tss0
p0:p0_if_r
p1:p1_if_r
pf0vf0:pf0vf0_if_r
pf0vf1:pf0vf1_if_r
pf1hpf:pf1hpf_if_r
pf1vf0:pf1vf0_if_r
pf1vf1:pf1vf1_if_r

[ENABLE_BR_SFC]
yes

[ENABLE_BR_SFC_DEFAULT_FLOWS]
yes
```



8.9.4.3.2.3 3.2.3. Mixed Mode HBN Deployment Configuration

The following is a sample **bf.cfg** and the resulting OVS and port configurations:

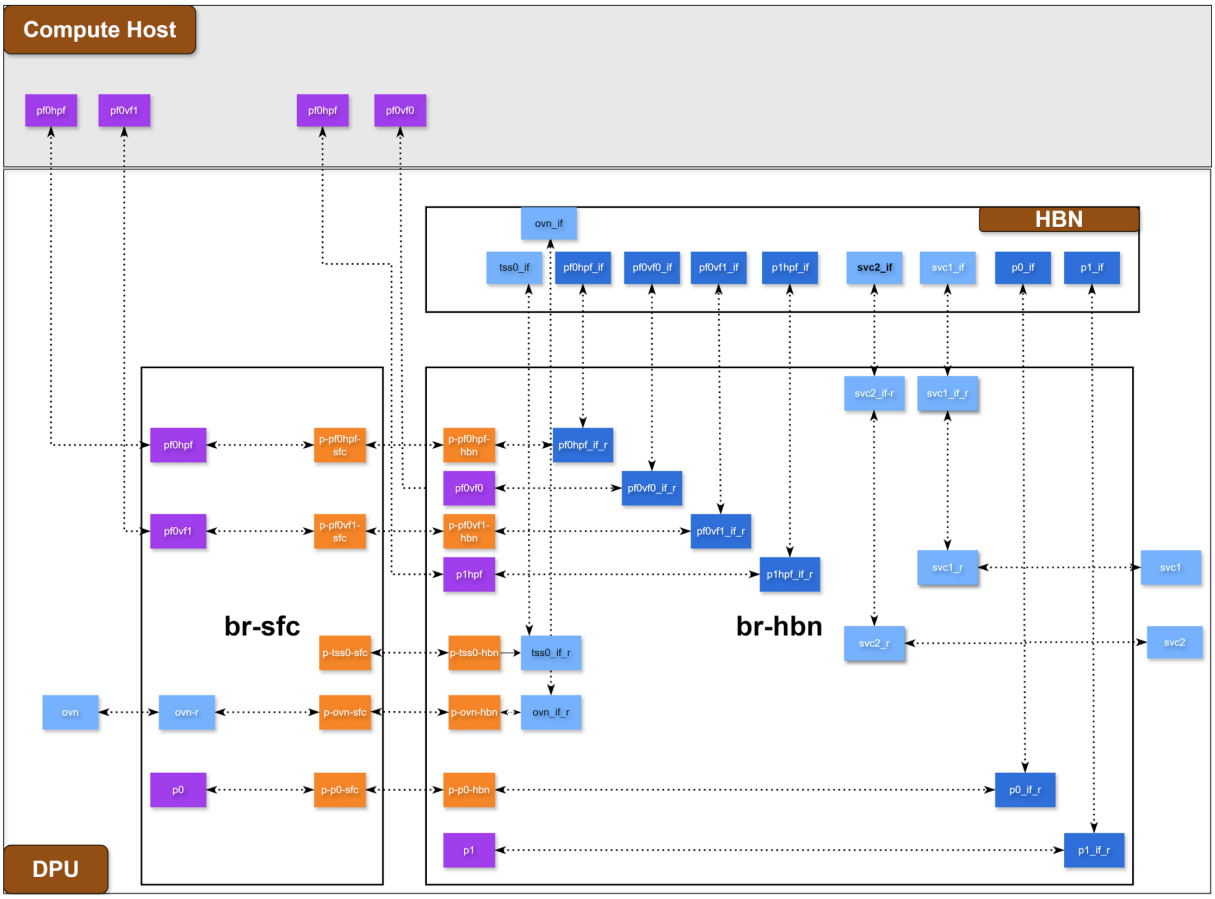
- Sample **bf.cfg**:
  bf.cfg

```
BR_HBN_UPLINKS="p1"
BR_SFC_UPLINKS="p0"
BR_HBN_REPS="pf1hpf,pf0vf0"
BR_SFC_REPS="pf0hpf,pf0vf1"
BR_HBN_SFS="svc1,svc2"
BR_SFC_SFS="ovn"
BR_HBN_SFC_PATCH_PORTS="tss0"
LINK_PROPAGATION="pf0hpf:tss0"
ENABLE_BR_SFC=yes
ENABLE_BR_SFC_DEFAULT_FLOWS=yes
```

- Generated **hbn.conf**:
  Generated hbn.conf

```
[BR_HBN_UPLINKS]
p1

[BR_HBN_REPS]
pf0vf0
pf1hpf

[BR_HBN_SFS]
svc1
svc2

[BR_SFC_UPLINKS]
p0

[BR_SFC_REPS]
pf0hpf
pf0vf1

[BR_SFC_SFS]
ovn

[BR_HBN_SFC_PATCH_PORTS]
tss0

[LINK_PROPAGATION]
```

```
pf0hpf:tss0
p1:p1_if_r
p0:p0_if_r
pf0vf0:pf0vf0_if_r
pf0hpf:pf0hpf_if_r
pf0vf1:pf0vf1_if_r
pf1hpf:pf1hpf_if_r
svc1_r:svc1_if_r
svc2_r:svc2_if_r
ovn_r:ovn_if_r

[ENABLE_BR_SFC]
yes

[ENABLE_BR_SFC_DEFAULT_FLOWS]
yes
```



8.9.4.3.3 3.3. HBN Deployment Considerations

8.9.4.3.3.1 3.3.1. SF Interface State Tracking

When HBN is deployed with SFC, the interface state of the following network devices is propagated to their corresponding SFs:

- Uplinks – `p0` , `p1`
- PFs – `pf0hpf` , `pf1hpf`
- VFs – `pf0vfX` , `pf1vfX` where `X` is the VF number

For example, if the p0 uplink cable gets disconnected:

- **p0** transitions to DOWN state with **NO-CARRIER** (default behavior on Linux); and
- **p0** state is propagated to **p0_if** whose state also becomes DOWN with NO-CARRIER

After **p0** connection is reestablished:

- **p0** transitions to UP state; and
- **p0** state is propagated to **p0_if** whose state becomes UP

Interface state propagation only happens in the uplink/PF/VF-to-SF direction.

A daemon called **sfc-state-propagation** runs on BlueField, outside of the HBN container, to sync the state. The daemon listens to netlink notifications for interfaces and transfers the state to SFs.

8.9.4.3.3.2 3.3.2. SF Interface MTU

In the HBN container, all the interfaces MTU are set to 9216 by default. MTU of specific interfaces can be overwritten using flat-files configuration or NVUE.

On BlueField side (i.e., outside of the HBN container), the MTU of the uplinks, PFs and VFs interfaces are also set to 9216. This can be changed by modifying **/etc/systemd/network/30-hbn-mtu.network** or by adding a new configuration file in the **/etc/systemd/network** for specific directories.

To reload this configuration, run:

```
systemctl restart systemd-networkd
```

8.9.4.3.3.3 3.3.3. Connecting to DOCA Services to HBN on BlueField Arm

There are various SF ports (named **pf0dpuX_if**, where X is [0..n]) on BlueField Arm, which can be used to run any services on BlueField and use HBN to provide network connectivity. These ports can have a flexible naming convention based on the service name. For example, to support OVN service, it can create an interface named **ovn** which can be used by the OVN service running on the BlueField Arm, and it will get a corresponding HBN port named **ovn_if**. These interfaces are created using either **BR_SFC_SFS** or **BR_HBN_SFS** based on which the bridge needs the service interface and mode of service deployment.

Traffic between BlueField and the outside world is hardware-accelerated when the HBN side port is an L3 interface or access-port using switch virtual interface (SVI). So, it is treated the same way as PF or VF ports from a traffic handling standpoint.

There are 2 SF port pairs created by default on BlueField Arm side so there can be 2 separate DOCA services running at same time.

8.9.4.3.3.4 3.3.4. Disabling BlueField Uplinks

The uplink ports must be always kept administratively up for proper operation of HBN. Otherwise, the NVIDIA® ConnectX® firmware would bring down the corresponding representor port which would cause data forwarding to stop.

Change in operational status of uplink (e.g., carrier down) would result in traffic being switched to the other uplink.

When using ECMP failover on the two uplink SFs, locally disabling one uplink does not result in traffic switching to the second uplink. Disabling local link in this case means to set one uplink admin DOWN directly on BlueField.

To test ECMP failover scenarios correctly, the uplink must be disabled from its remote counterpart (i.e., execute admin DOWN on the remote system's link which is connected to the uplink).

8.9.4.3.3.5 3.3.5. HBN NVUE User Credentials

The preconfigured default user credentials are as follows:

Username	**nvidia**
Password	**nvidia**

NVUE user credentials can be added post installation:

1. This can be done by specifying additional parameters to the HBN startup script (refer to "Running HBN Preparation Script"). The user can provide either an already hashed password or a plain text password. It is recommended to use a salted hash value for the password to avoid using a plaintext password. Use of plaintext passwords will be deprecated on the next release. For example:
   a. Using the **--username** and **--hashed_password** parameters:

```
openssl passwd -6
Password:
Verifying - Password:
$6$gPG6srw9hgvVCVdo$hto3fyl975Nb7UUNamdwqzz0tNCLn1AgYpuCt2DQTLnv4Hh2FHFbXmztQR.6Vb7XZZ1JK.exYRzaq8Q
nRmzwG0

sudo ./hbn-dpu-setup.sh -u newuser -H
$6$gPG6srw9hgvVCVdo$hto3fyl975Nb7UUNamdwqzz0tNCLn1AgYpuCt2DQTLnv4Hh2FHFbXmztQR.6Vb7XZZ1JK.exYRzaq8Q
nRmzwG0
```

   b. Using the **--username** and **--password** parameters:

```
sudo ./hbn-dpu-setup.sh -u newuser -p newpassword
```

2. After executing this script, respawn the container or start the **add-users** script inside running HBN container:

```
supervisorctl start add-users
add-users: started
```

   The script creates a new user in the HBN container:

```
cat /etc/passwd | grep newuser
newuser:x:1001:1001::/home/newuser:/bin/bash
```

8.9.4.3.3.6 3.3.6. HBN NVUE Interface Classification

Interface	Interface Type	NVUE Type
p0_if	Uplink representor	swp
p1_if	Uplink representor	swp
lo	Loopback	loopback
pf0hpf_if	Host representor	swp
pf1hpf_if	Host representor	swp
pf0vfx_if (where **x** is 0-255)	VF representor	swp
pf1vfx_if (where **x** is 0-255)	VF representor	swp

8.9.4.3.3.7 3.3.7. HBN Files Persistence

The following directories are mounted from BlueField Arm to the HBN container namespace and are persistent across HBN service restarts and BlueField reboots:

	BlueField Arm Mount Point	HBN Container Mount Point
Configuration file mount points	**/var/lib/hbn/etc/network/**	**/etc/network/**
	/var/lib/hbn/etc/frr/	**/etc/frr/**
	/var/lib/hbn/etc/nvue.d/	**/etc/nvue.d/**
	/var/lib/hbn/etc/supervisor/conf.d/	**/etc/supervisor/conf.d/**
	/var/lib/hbn/var/lib/nvue/	**/var/lib/nvue/**
Support and log file mount points	**/var/lib/hbn/var/support/**	**/var/support/**
	/var/log/doca/hbn/	**/var/log/hbn/**

8.9.4.3.3.8 3.3.8. SR-IOV Support in HBN

3.3.8.1. Creating SR-IOV VFs on Host

The first step to use SR-IOV is to create Virtual Functions (VFs) on the host server.

VFs can be created using the following command:

```
sudo echo N > /sys/class/net/<host-rep>/device/sriov_numvfs
```

Where:

- **<host-rep>** is one of the two host representors (e.g., **ens1f0** or **ens1f1**)
- 0≤ **N** ≤16 is the desired total number of VFs
  - Set **N** =0 to delete all the VFs on 0≤N≤16

- **N** =16 is the maximum number of VFs supported on HBN across all representors

3.3.8.2. Automatic Creation of VF Representors and SF Devices on BlueField

VFs created on the host must have corresponding VF representor devices and SF devices for HBN on BlueField side. For example:

- **ens1f0vf0** is the first SR-IOV VF device from the first host representor; this interface is created on the host server
- **pf0vf0** is the corresponding VF representor device to **ens1f0vf0**; this device is present on the BlueField Arm side and automatically created at the same time as **ens1f0vf0** is created by the user on the host side
- **pf0vf0_if** is the corresponding SF device for **pf0vf0** which is used to connect the VF to HBN pipeline

The creation of the SF device for VFs is done ahead of time when provisioning the BlueField and installing the DOCA image on it, see section "Enabling SFC" to see how to select how many SFs to create ahead of time.

The SF devices for VFs (i.e., **pfXvfY**) are pre-mapped to work with the corresponding VF representors when these are created with the command from the previous step.

8.9.4.3.3.9 3.3.9. Management VRF

Two management VRFs are automatically configured for HBN when BlueField is deployed with SFC:

- The first management VRF is outside the HBN container on BlueField. This VRF provides separation between out-of-band (OOB) traffic (via **oob_net0** or **tmfifo_net0**) and data-plane traffic via uplinks and PFs.
- The second management VRF is inside the HBN container and provides similar separation. The OOB traffic (via **eth0**) is isolated from the traffic via the ***_if** interfaces.

3.3.9.1. MGMT VRF on BlueField Arm

The management (mgmt) VRF is enabled by default when the BlueField is deployed with SFC (see section "Enabling SFC"). The mgmt VRF provides separation between the OOB management network and the in-band data plane network.

The uplinks and PFs/VFs use the default routing table while the **oob_net0** (OOB Ethernet port) and the **tmifo_net0** netdevices use the mgmt VRF to route their packets.

When logging in either via SSH or the console, the shell is by default in mgmt VRF context. This is indicated by a mgmt added to the shell prompt:

```
root@bf2:mgmt:/home/ubuntu#
root@bf2:mgmt:/home/ubuntu# ip vrf identify
mgmt.
```

When logging into the HBN container with **crictl**, the HBN shell will be in the default VRF. Users must switch to MGMT VRF manually if OOB access is required. Use **ip vrf exec** to do so.

```
root@bf2:mgmt:/home/ubuntu# ip vrf exec mgmt bash
```

The user must run **ip vrf exec mgmt** to perform operations requiring OOB access (e.g., apt-get update).

Network devices belonging to the mgmt VRF can be listed with the **vrf** utility:

```
root@bf2:mgmt:/home/ubuntu# vrf link list

VRF: mgmt
--------------------
tmfifo_net0      UP             00:1a:ca:ff:ff:03 <BROADCAST,MULTICAST,UP,LOWER_UP>
oob_net0         UP             08:c0:eb:c0:5a:32 <BROADCAST,MULTICAST,UP,LOWER_UP>

root@bf2:mgmt:/home/ubuntu# vrf help
vrf <OPTS>

VRF domains:
    vrf list

Links associated with VRF domains:
    vrf link list [<vrf-name>]

Tasks and VRF domain asociation:
    vrf task exec <vrf-name> <command>
    vrf task list [<vrf-name>]
    vrf task identify <pid>

    NOTE: This command affects only AF_INET and AF_INET6 sockets opened by the
          command that gets exec'ed. Specifically, it has *no* impact on netlink
          sockets (e.g., ip command).
```

To show the routing table for the default VRF, run:

```
root@bf2:mgmt:/home/ubuntu# ip route show
```

To show the routing table for the mgmt VRF, run:

```
root@bf2:mgmt:/home/ubuntu# ip route show vrf mgmt
```

3.3.9.2. MGMT VRF Inside HBN Container

Inside the HBN container, a separate mgmt VRF is present. Similar commands as those listed under section "MGMT VRF on BlueField Arm" can be used to query management routes.

The ***_if** interfaces use the default routing table while the **eth0** (OOB) uses the mgmt VRF to route out-of-band packets out of the container. The OOB traffic gets NATed through the **oob_net0** interface on BlueField Arm, ultimately using the BlueField OOB's IP address.

When logging into the HBN container via **crictl**, the shell enters the default VRF context by default. Switching to the mgmt VRF can be done using the command **ip vrf exec mgmt <cmd>**.

3.3.9.3. Existing Services in MGMT VRF on BlueField Arm

On the BlueField Arm, outside the HBN container, a set of existing services run in the mgmt VRF context as they need OOB network access:

- containerd
- kubelet
- ssh
- docker

These services can be restarted and queried for their status using the command **systemctl** while adding **@mgmt** to the original service name. For example:

- To restart containerd:

```
root@bf2:mgmt:/home/ubuntu# systemctl restart containerd@mgmt
```

- To query containerd status:

```
root@bf2:mgmt:/home/ubuntu# systemctl status containerd@mgmt
```

The original version of these services (without **@mgmt**) are not used and must not be started.

3.3.9.4. Running New Service in MGMT VRF on BlueField Arm

If a service needs OOB access to run, it can be added to the set of services running in mgmt VRF context. Adding such a service is only possible on the BlueField Arm (i.e., outside the HBN container).

To add a service to the set of mgmt VRF services:

1. Add it to **/etc/vrf/systemd.conf** (if it is not present already). For example, NTP is already listed in this file.
2. Run the following:

```
root@bf2:mgmt:/home/ubuntu# systemctl daemon-reload
```

3. Stop and disable to the non-VRF version of the service to be able to start the mgmt VRF one:

```
root@bf2:mgmt:/home/ubuntu# systemctl stop ntp
root@bf2:mgmt:/home/ubuntu# systemctl disable ntp
root@bf2:mgmt:/home/ubuntu# systemctl enable ntp@mgmt
root@bf2:mgmt:/home/ubuntu# systemctl start ntp@mgmt
```

8.9.5 HBN Service Configuration

To start configuring HBN, log in to the HBN container:

```
sudo crictl exec -it $(crictl ps | grep hbn | awk '{print $1;}') bash
```

8.9.5.1 General Network Configuration

8.9.5.1.1 Flat Files Configuration

Add network interfaces and FRR configuration files to HBN. For more more information refer to the Cumulus Linux FRRouting guide.

- **/etc/network/interfaces**

- **/etc/frr/frr.conf** ; **/etc/frr/daemons**

8.9.5.2 NVUE Configuration

This section assumes familiarity with NVIDIA user experience (NVUE) Cumulus Linux. The following sections elaborate on HBN-specific aspects of NVUE.

8.9.5.2.1 NVUE Service

HBN installs NVUE by default and enables NVUE service at boot.

8.9.5.2.2 NVUE REST API

HBN enables the REST API with only localhost access by default. To enable REST API access, refer to Enable REST API Access.

You can run cURL commands from the command line. Use the default HBN username, **nvidia**, and password which you must be update when enabling the REST API using the HBN preparation script.

To change the default password for the **nvidia** user or to add additional users for NVUE access, refer to section HBN Service Configuration#HBN NVUE User Credentials.

REST API example:

```
curl -u 'nvidia:nvidia' --insecure https://<mgmt_ip>:8765/nvue_v1/vrf/default/router/bgp
{
  "configured-neighbors": 2,
  "established-neighbors": 2,
  "router-id": "10.10.10.201"
}
```

8.9.5.2.2.1 NVUE REST API Management Through CLI

- To enable the REST API service:

```
nv set system api state enabled
```

- To disable the REST API service:

```
nv set system api state disabled
```

- To bind the REST API service to a specific address:

```
nv set system api listening-address <localhost|ipv4|ipv6|0.0.0.0>
```

For information about using the NVUE REST API, refer to the NVUE API documentation.

8.9.5.2.3 NVUE REST API in DOCA Platform Framework

> For information on DOCA Platform Framework (DPF), refer to DOCA Framework.

This section explains how to configure external access to the HBN NVUE REST API in a DOCA Platform Framework (DPF) environment.

By default, the NVUE REST API is only accessible from within the cluster (i.e., via **localhost**). To expose the API to external networks, you must configure Kubernetes NodePort and set the appropriate listening address.

8.9.5.2.3.1 Enabling External Access to the NVUE REST API

1. Configure NodePort in DPUService – Add the following section under **spec.serviceConfiguration.helmChart.values** in your **DPUServiceConfiguration** YAML:

```
service:
  type: NodePort
  nodePort: 30765
```

2. Set the API listening address – To bind the API to all available HBN IP addresses (for example, for external access), configure the following:

```
configuration:
  startupYAMLJ2: |
    - set:
        system:
          api:
            listening-address:
              0.0.0.0: {}
```

3. Verify API accessibility – Use **curl** to confirm that the NVUE REST API is reachable:

```
curl -k -u 'username:password' https://<node-ip>:<nodePort>/nvue_v1/interface/
```

4. Check service status – To check the NodePort service status:

```
kubectl get svc -n <your-namespace>
```

8.9.5.2.3.2 Example: Combined DPUServiceConfiguration and DPUServiceTemplate

The following is a sample manifest to deploy HBN with the NVUE REST API enabled:

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: doca-hbn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "doca-hbn"
  serviceConfiguration:
    helmChart:
      values:
        service:
          nodePort: 30765
          type: NodePort
        configuration:
          user:
            create: true
            password:
              secretKey: password
              secretName: hbn-user-password
```

```
    username: nvidia
  perDPUValuesYAML: |
    - hostnamePattern: "*"
  startupYAMLJ2: |
    - header:
        model: bluefield
        nvue-api-version: nvue_v1
        rev-id: 1.0
        version: HBN 3.1.0
    - set:
        system:
          api:
            listening-address:
              0.0.0.0: {}
```

8.9.5.2.3.3 Security Considerations

Authentication

- Always use valid username/password authentication.
- Use HTTPS to secure all REST API communications.
- The default credentials are `nvidia:nvidia`.
- It is strongly recommended to change the default credentials using the user creation procedure described below.

User Creation/Update Procedure for NVUE REST API

> ⚠ The user should not interact directly with the DPU cluster in DPF.

1. Create a password secret on the host cluster:

```
kubectl -n dpf-operator-system create secret generic hbn-user-password \
  --from-literal=password=<new-password>
```

2. Propagate the secret to the DPU cluster (label the secret so that it is made available to the DPU):

```
kubectl -n dpf-operator-system label secret hbn-user-password \
  dpu.nvidia.com/image-pull-secret=""
```

3. Update the user configuration in the helm values your service YAML:

```
configuration:
  user:
    create: true
    username: "your-username"
    password:
      secretName: "hbn-user-password" #Must match the secret name from Step 1
      secretKey: "password"           #Must match the key in the secret
    # Optional settings
    shell: "/bin/bash"
    homeDir: "/home/your-username"
    additionalGroups: ["group1", "group2"]
    uid: "1000"
    gid: "1000"
```

User Creation Configuration Options

The following options can be used to configure user creation or update in the `spec.serviceConfiguration.helmChart.values.configuration.user` section.

Option	Required	Description
`create`	Yes	Enables user creation or update. Default is `false`.
`username`	Yes	Username for the new user, or existing user in case of an update
`password.secretName`	Yes	Name of the Kubernetes secret that stores the password
`password.secretKey`	Yes	Key within the secret that contains the password value
`shell`	No	Shell assigned to the user (e.g., `/bin/bash`)
`homeDir`	No	Path to the user's home directory
`additionalGroups`	No	List of additional groups to assign the user to (e.g., `["group1", "group2"]`)
`uid`	No	User ID
`gid`	No	Group ID

8.9.5.2.4 NVUE CLI

For information about using the NVUE CLI, refer to the NVUE CLI documentation

8.9.5.2.5 NVUE Startup Configuration File

When the network configuration is saved using NVUE, HBN writes the configuration to the `/etc/nvue.d/startup.yaml` file.

The startup configuration is applied by following the supervisor daemon at boot time. `nvued-startup` will appear in an `EXITED` state after applying the startup configuration:

```
# supervisorctl status nvued-startup
nvued-startup                    EXITED    Apr 17 10:04 AM
```

⚠ `nv config apply startup` applies the yaml configuration saved at `/etc/nvue.d/`. `nv config save` saves the running configuration to `/etc/nvue.d/startup.yaml`.

8.9.5.3 HBN Configuration Examples

8.9.5.3.1 HBN Default Configuration

After a fresh HBN installation, the default `/etc/network/interfaces` file should contain only the declaration of a loopback interface:

```
source /etc/network/interfaces.d/*.intf

auto lo
```

```
iface lo inet loopback
```

FRR configuration files are also present in **/etc/frr/** but configurations are not enabled.

8.9.5.3.2 Layer-3 Routing

8.9.5.3.2.1 Next-hop IDs and Next-hop Groups

In HBN, route programming utilizes next-hop IDs to specify a route's next-hop. For routes with multiple paths, next-hop groups are used to identify equal-cost multi-path (ECMP) next-hop sets. This grouping enables the system to efficiently process and modify route and next-hop objects, improving scalability, network convergence, and performance. You may notice changes in the display of route information in the output of kernel operational commands.

8.9.5.3.2.2 Native Routing with BGP and ECMP

HBN supports unicast routing with BGP and ECMP for IPv4 and IPv6 traffic. ECMP is achieved by distributing traffic using hash calculation based on the source IP, destination IP, and protocol type of the IP header.

For TCP and UDP packets, the source and destination ports are also included.

ECMP Example

ECMP is implemented any time routes have multiple paths over uplinks or host ports. For example, 20.20.20.0/24 has 2 paths using both uplinks, so a path is selected based on a hash of the IP headers.

```
# ip route show 20.20.20.0/24
20.20.20.0/24 nhid 106 proto bgp metric 20
# ip nexthop show group id 106
id 106 group 105/107 proto zebra
# ip nexthop show id 105
id 105 via fe80::202:ff:fe00:7 dev p0_if scope link proto zebra
# ip nexthop show id 107
id 107 via fe80::202:ff:fe00:c dev p1_if scope link proto zebra
```

HBN supports up to 64 paths for ECMP

Sample NVUE Configuration for Native Routing with Host-facing Ports as Access

```
nv set interface lo ip address 10.10.10.1/32
nv set interface lo ip address 2010:10:10::1/128
nv set interface vlan100 type svi
nv set interface vlan100 vlan 100
nv set interface vlan100 base-interface br_default
nv set interface vlan100 ip address 2030:30:30::1/64
nv set interface vlan100 ip address 30.30.30.1/24
nv set bridge domain br_default vlan 100
nv set interface pf0hpf_if,pf1hpf_if bridge domain br_default access 100
nv set vrf default router bgp router-id 10.10.10.1
nv set vrf default router bgp autonomous-system 65501
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf default router bgp neighbor p0_if remote-as external
nv set vrf default router bgp neighbor p0_if type unnumbered
nv set vrf default router bgp neighbor p0_if address-family ipv4-unicast enable on
nv set vrf default router bgp neighbor p0_if address-family ipv6-unicast enable on
nv set vrf default router bgp neighbor p1_if remote-as external
nv set vrf default router bgp neighbor p1_if type unnumbered
nv set vrf default router bgp neighbor p1_if address-family ipv4-unicast enable on
```

```
nv set vrf default router bgp neighbor p1_if address-family ipv6-unicast enable on
```

Sample NVUE Configuration for Native Routing with Host-facing Ports as Trunk

```
nv set interface lo ip address 10.10.10.1/32
nv set interface lo ip address 2010:10:10::1/128
nv set interface vlan100 type svi
nv set interface vlan100 vlan 100
nv set interface vlan100 base-interface br_default
nv set interface vlan100 ip address 2030:30:30::1/64
nv set interface vlan100 ip address 30.30.30.1/24
nv set bridge domain br_default vlan 100
nv set interface pf0hpf_if,pf1hpf_if bridge domain br_default
nv set vrf default router bgp router-id 10.10.10.1
nv set vrf default router bgp autonomous-system 65501
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf default router bgp neighbor p0_if remote-as external
nv set vrf default router bgp neighbor p0_if type unnumbered
nv set vrf default router bgp neighbor p0_if address-family ipv4-unicast enable on
nv set vrf default router bgp neighbor p0_if address-family ipv6-unicast enable on
nv set vrf default router bgp neighbor p1_if remote-as external
nv set vrf default router bgp neighbor p1_if type unnumbered
nv set vrf default router bgp neighbor p1_if address-family ipv4-unicast enable on
nv set vrf default router bgp neighbor p1_if address-family ipv6-unicast enable on
```

Sample Flat Files Configuration for Native Routing with Host-facing Ports as Access

Example **/etc/network/interfaces** configuration:

```
auto lo
iface lo inet loopback
    address 10.10.10.1/32
    address 2010:10:10::1/128

auto p0_if
iface p0_if

auto p1_if
iface p1_if

auto pf0hpf_if
iface pf0hpf_if
    bridge-access 100

auto pf1hpf_if
iface pf1hpf_if
    bridge-access 100

auto vlan100
iface vlan100
    address 2030:30:30::1/64
    address 30.30.30.1/24
    vlan-raw-device br_default
    vlan-id 100

auto br_default
iface br_default
    bridge-ports pf0hpf_if pf1hpf_if
    bridge-vlan-aware yes
    bridge-vids 100
    bridge-pvid 1
```

Sample Flat Files Configuration for Native Routing with Host-facing Ports as Trunk

Example **/etc/network/interfaces** configuration:

```
auto lo
iface lo inet loopback
    address 10.10.10.1/32
    address 2010:10:10::1/128

auto p0_if
iface p0_if

auto p1_if
iface p1_if

auto pf0hpf_if
iface pf0hpf_if

auto pf1hpf_if
iface pf1hpf_if
```

```
auto vlan100
iface vlan100
    address 2030:30:30::1/64
    address 30.30.30.1/24
    vlan-raw-device br_default
    vlan-id 100

auto br_default
iface br_default
    bridge-ports pf0hpf_if pf1hpf_if
    bridge-vlan-aware yes
    bridge-vids 100
    bridge-pvid 1
```

Example **/etc/frr/daemons** configuration:

```
bgpd=yes
vtysh_enable=yes


FRR Config file @ /etc/frr/frr.conf -
!
frr version 7.5+cl5.3.0u0
frr defaults datacenter
hostname BLUEFIELD2
log syslog informational
no zebra nexthop kernel enable
!
router bgp 65501
 bgp router-id 10.10.10.1
 bgp bestpath as-path multipath-relax
 neighbor p0_if interface remote-as external
 neighbor p0_if advertisement-interval 0
 neighbor p0_if timers 3 9
 neighbor p0_if timers connect 10
 neighbor p1_if interface remote-as external
 neighbor p1_if advertisement-interval 0
 neighbor p1_if timers 3 9
 neighbor p1_if timers connect 10
 !
 address-family ipv4 unicast
  redistribute connected
  maximum-paths 64
  maximum-paths ibgp 64
 exit-address-family
 !
 address-family ipv6 unicast
  redistribute connected
  neighbor p0_if activate
  neighbor p1_if activate
  maximum-paths 64
  maximum-paths ibgp 64
 exit-address-family
!
line vty
!
end
```

Direct Routing on Host-facing Interfaces

Host-facing interfaces (PFs and VFs) are not limited to being part of the bridge for routing. HBN supports L3-only configuration with direct routing on host-facing PFs and VFs.

Sample NVUE Configuration

```
nv set interface pf0hpf_if ip address 30.30.11.1/24
nv set interface pf0hpf_if ip address 2030:30:11::1/64
nv set interface pf0vf0_if ip address 30.30.13.1/24
nv set interface pf0vf0_if ip address 2030:30:13::1/64
```

Sample Flat File Configuration

```
auto pf0hpf_if
iface pf0hpf_if
    address 2030:30:11::1/64
    address 30.30.11.1/24

auto pf0vf0_if
iface pf0vf0_if
    address 2030:30:13::1/64
    address 30.30.13.1/24
```

8.9.5.3.2.3 BGP Peering with the Host

HBN supports the ability to establish a BGP session between the host and the HBN service running on BlueField Arm and allow the host to announce arbitrary route prefixes through the BlueField into the underlay fabric. The host can use any standard BGP protocol stack implementation to establish BGP peering with HBN.

Traffic to and from endpoints on the host gets offloaded.

> ⚠ Both IPv4 and IPv6 unicast AFI/SAFI are supported.

It is possible to apply route filtering for these prefixes to limit the potential security impact in this configuration.

Sample NVUE Configuration for Host BGP Peering

The following code block shows configuration to peer to host at **45.3.0.4** and **2001:cafe:1ead::4**. The BGP session can be established using IPv4 or IPv6 address.

> ⚠ Either of these sessions can support IPv4 unicast and IPv6 unicast AFI/SAFI.

NVUE configuration for peering with host:

```
nv set vrf default router bgp autonomous-system 63642
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor 45.3.0.4 nexthop-connected-check off
nv set vrf default router bgp neighbor 45.3.0.4 peer-group dpu_host
nv set vrf default router bgp neighbor 45.3.0.4 type numbered
nv set vrf default router bgp neighbor 2001:cafe:1ead::4 nexthop-connected-check off
nv set vrf default router bgp neighbor 2001:cafe:1ead::4 peer-group dpu_host
nv set vrf default router bgp neighbor 2001:cafe:1ead::4 type numbered
nv set vrf default router bgp peer-group dpu_host address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group dpu_host address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group dpu_host remote-as external
```

Sample Flat Files Configuration for Host BGP peering

The following block shows configuration to peer to host at **45.3.0.4** and **2001:cafe:1ead::4**. The BGP session can be established using IPv4 or IPv6 address.

frr.conf file:

```
router bgp 63642
 bgp router-id 27.0.0.4
 bgp bestpath as-path multipath-relax
 neighbor dpu_host peer-group
 neighbor dpu_host remote-as external
 neighbor dpu_host advertisement-interval 0
 neighbor dpu_host timers 3 9
 neighbor dpu_host timers connect 10
 neighbor dpu_host disable-connected-check
 neighbor fabric peer-group
 neighbor fabric remote-as external
 neighbor fabric advertisement-interval 0
 neighbor fabric timers 3 9
 neighbor fabric timers connect 10
 neighbor 45.3.0.4 peer-group dpu_host
 neighbor 2001:cafe:1ead::4 peer-group dpu_host
 neighbor p0_if interface peer-group fabric
 neighbor p1_if interface peer-group fabric
 !
 address-family ipv4 unicast
    neighbor dpu_host activate
 !
 address-family ipv6 unicast
    neighbor dpu_host activate
```

Sample FRR configuration on the Host

Any BGP implementation can be used on the host to peer to HBN and advertise endpoints. The following is an example using FRR BGP:

Sample FRR configuration on the host:

```
bf2-s12# sh run
Building configuration...

Current configuration:
!
frr version 7.2.1
frr defaults traditional
hostname bf2-s12
no ip forwarding
no ipv6 forwarding
!
router bgp 1000008
!
router bgp 1000008 vrf v_200_2000
 neighbor 45.3.0.2 remote-as external
 neighbor 2001:cafe:1ead::2 remote-as external
 !
 address-family ipv4 unicast
  redistribute connected
 exit-address-family
 !
 address-family ipv6 unicast
  redistribute connected
  neighbor 45.3.0.2 activate
  neighbor 2001:cafe:1ead::2 activate
 exit-address-family
!
line vty
!
end
```

Sample interfaces configuration on the host:

```
root@bf2-s12:/home/cumulus# ifquery -a
auto lo
iface lo inet loopback
    address 27.0.0.7/32
    address 2001:c000:10ff:f00d::7/128

auto v_200_2000
iface v_200_2000
    address 60.1.0.1
    address 60.1.0.2
    address 60.1.0.3
    address 2001:60:1::1
    address 2001:60:1::2
    address 2001:60:1::3
    vrf-table auto
auto ens1f0np0
iface ens1f0np0
    address 45.3.0.4/24
    address 2001:cafe:1ead::4/64
    gateway 45.3.0.1
    gateway 2001:cafe:1ead::1
    vrf v_200_2000
    hwaddress 00:03:00:08:00:12
    mtu 9162
```

8.9.5.3.2.4 VRF Route Leaking

VRFs are typically used when multiple independent routing and forwarding tables are desirable. However, users may want to reach destinations in one VRF from another VRF, as in the following cases:

- To make a service, such as a firewall available to multiple VRFs
- To enable routing to external networks or the Internet for multiple VRFs, where the external network itself is reachable through a specific VRF

Route leaking can be used to reach remote destinations as well as directly connected destinations in another VRF. Multiple VRFs can import routes from a single source VRF, and a VRF can import routes from multiple source VRFs. This can be used when a single VRF provides connectivity to

external networks or a shared service for other VRFs. It is possible to control the routes leaked dynamically across VRFs with a route map.

When route leaking is used:

- The **redistribute** command (not **network** command) must be used in BGP to leak non-BGP routes (connected or static routes)
- It is not possible to leak routes between the default and non-default VRF

> **Kernel limitation**
>
> 3769309 – Ping or other IP traffic from a locally connected host in vrfX to a local interface IP address on the BlueField/HBN in vrfY does not work, even if VRF route-leaking is enabled between these two VRFs.

In the following example commands, routes in the BGP routing table of VRF **BLUE** dynamically leak into VRF **RED**:

```
nv set vrf RED router bgp address-family ipv4-unicast route-import from-vrf list BLUE
nv config apply
```

The following example commands delete leaked routes from VRF **BLUE** to VRF **RED**:

```
nv unset vrf RED router bgp address-family ipv4-unicast route-import from-vrf list BLUE
nv config apply
```

To exclude certain prefixes from the import process, configure the prefixes in a route map.

The following example configures a route map to match the source protocol BGP and imports the routes from VRF **BLUE** to VRF **RED**. For the imported routes, the community is 11:11 in VRF **RED**.

```
nv set vrf RED router bgp address-family ipv4-unicast route-import from-vrf list BLUE
nv set router policy route-map BLUEtoRED rule 10 match type ipv4
nv set router policy route-map BLUEtoRED rule 10 match source-protocol bgp
nv set router policy route-map BLUEtoRED rule 10 action permit
nv set router policy route-map BLUEtoRED rule 10 set community 11:11
nv set vrf RED router bgp address-family ipv4-unicast route-import from-vrf route-map BLUEtoRED
nv config
```

To check the status of the VRF route leaking, run:

- NVUE command:

  ```
  nv show vrf <vrf-name> router bgp address-family ipv4-unicast route-import
  ```

- Vtysh command:

  ```
  show ip bgp vrf <vrf-name> ipv4|ipv6 unicast route-leak command.
  ```

- For example:

  ```
  nv show vrf RED router bgp address-family ipv4-unicast route-import
                  operational   applied
  --------------  ------------  ---------
  from-vrf
    enable                      on
    route-map                   BLUEtoRED
    [list]        BLUE          BLUE
  ```

```
[route-target]  10.10.10.1:3
```

To show more detailed status information, the following NVUE commands are available:

- **nv show vrf <vrf-name> router bgp address-family ipv4-unicast route-import from-vrf**
- **nv show vrf <vrf-name> router bgp address-family ipv4-unicast route-import from-vrf list**
- **nv show vrf <vrf-name> router bgp address-family ipv4-unicast route-import from-vrf list <leak-vrf-id>**

To view the BGP routing table, run:

- NVUE command:

```
nv show vrf <vrf-name> router bgp address-family ipv4-unicast
```

- Vtysh command:

```
show ip bgp vrf <vrf-name> ipv4|ipv6 unicast
```

To view the FRR IP routing table, run:

- Vtysh command:

```
show ip route vrf <vrf-name>
```

- Or:

```
net show route vrf <vrf-name>
```

These commands show all routes, including routes leaked from other VRFs.

8.9.5.3.2.5 VLAN Subinterfaces

A VLAN subinterface is a VLAN device on an interface. The VLAN ID appends to the parent interface using dot (`.`) VLAN notation which is a standard way to specify a VLAN device in Linux.

For example:

- A VLAN with ID 100 which is a subinterface of **p0_if** is annotated as **p0_if.100**
- The subinterface **p0_if.100** only receives packets that have a VLAN 100 tag on port **p0_if**
- Any packets transmitted from **p0_if.100** would have VLAN tag 100

In HBN, VLAN subinterfaces can be created on uplink ports as well as on the host-facing PF and VF ports. A VLAN subinterface only receives traffic tagged for that VLAN.

VLAN subinterfaces are L3 interfaces and should not be added to a bridge.

In the following example, uplink subinterface on **p0_if** with VLAN ID 10 and a host facing subinterface on VF ports **pf1vf0_if** with VLAN ID 999 are created. The host-facing subinterface is also assigned with IPv4 and IPv6 addresses.

Subinterface configuration using NVUE commands:

```
nv set interface p0_if.10 base-interface p0_if
nv set interface p0_if.10 type sub
nv set interface p0_if.10 vlan 10

nv set interface pf1vf0_if type swp
nv set interface pf1vf0_if.999 base-interface pf1vf0_if
nv set interface pf1vf0_if.999 type sub
nv set interface pf1vf0_if.999 vlan 999
nv set interface pf1vf0_if ip address 30.30.14.1/24
nv set interface pf1vf0_if ip address 2030:30:14::1/64
```

Same configuration using sample flat file in **/etc/network/interfaces**:

subinterface configuration e/n/i file

```
auto p0_if.10
iface p0_if.10

auto pf1vf0_if.999
iface pf1vf0_if.999
    address 2030:30:40::1/64
    address 30.30.40.1/24
```

8.9.5.3.3 Ethernet Virtual Private Network – EVPN

HBN supports VXLAN with EVPN control plane for intra-subnet bridging (L2) services for IPv4 and IPv6 traffic in the overlay. For the underlay, only IPv4 or BGP unnumbered configuration is supported.

HBN supports VXLAN encapsulation only over uplink parent interfaces.

8.9.5.3.3.1 Single VXLAN Device

With a single VXLAN device, a set of VXLAN network identifiers (VNIs) represents a single device model. The single VXLAN device has a set of attributes that belong to the VXLAN construct. Individual VNIs include VLAN-to-VNI mapping which allows users to specify which VLANs are associated with which VNIs. A single VXLAN device simplifies the configuration and reduces the overhead by replacing multiple traditional VXLAN devices with a single VXLAN device.

Users may configure a single VXLAN device automatically with NVUE, or manually by editing the **/etc/network/interfaces** file. When users configure a single VXLAN device with NVUE, NVUE creates a unique name for the device in the following format using the bridge name as the hash key: **vxlan<id>**.

This example configuration performs the following steps:

1. Creates a single VXLAN device (vxlan21).
2. Maps VLAN 10 to VNI 10 and VLAN 20 to VNI 20.
3. Adds the VXLAN device to the default bridge.

```
cumulus@leaf01:~$ nv set bridge domain bridge vlan 10 vni 10
cumulus@leaf01:~$ nv set bridge domain bridge vlan 20 vni 20
```

```
cumulus@leaf01:~$ nv set nve vxlan source address 10.10.10.1
cumulus@leaf01:~$ nv config apply
```

Alternately, edit the **/etc/network/interfaces** files, then run the **ifreload -a** command to apply the SVD configuration:

```
auto lo
iface lo inet loopback
    vxlan-local-tunnelip 10.10.10.1

auto vxlan21
iface vxlan21
    bridge-vlan-vni-map 10=10 20=20
    bridge-learning off

auto bridge
iface bridge
    bridge-vlan-aware yes
    bridge-ports vxlan21 pf0hpf_if pf1hpf_if
    bridge-vids 10 20
    bridge-pvid 1
```

⚠ Do not use a combination of single and traditional VXLAN devices.

8.9.5.3.3.2 Sample Switch Configuration for EVPN

The following is a sample NVUE config for underlay switches (NVIDIA Spectrum with Cumulus Linux) to enable EVPN deployments with HBN. It assumes that the uplinks on all BlueField devices are connected to ports **swp1-4** on the switch.

```
nv set evpn enable on
nv set router bgp enable on

nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on

nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp autonomous-system 63640
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor swp1 peer-group fabric
nv set vrf default router bgp neighbor swp1 type unnumbered
nv set vrf default router bgp neighbor swp2 peer-group fabric
nv set vrf default router bgp neighbor swp2 type unnumbered
nv set vrf default router bgp neighbor swp3 peer-group fabric
nv set vrf default router bgp neighbor swp3 type unnumbered
nv set vrf default router bgp neighbor swp4 peer-group fabric
nv set vrf default router bgp neighbor swp4 type unnumbered
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp router-id 27.0.0.10

nv set interface lo ip address 2001:c000:10ff:f00d::10/128
nv set interface lo ip address 27.0.0.10/32
nv set interface lo type loopback
nv set interface swp1,swp2,swp3,swp4 type swp
```

8.9.5.3.3.3 Layer-2 EVPN

Sample NVUE Configuration for L2 EVPN with Host-facing Ports as Access

The following is a sample NVUE configuration which has L2-VNIs (**2000** , **2001**) for EVPN bridging on BlueField.

```
nv set bridge domain br_default encap 802.1Q
nv set bridge domain br_default type vlan-aware
nv set bridge domain br_default vlan 200 vni 2000 flooding enable auto
nv set bridge domain br_default vlan 200 vni 2000 mac-learning off
nv set bridge domain br_default vlan 201 vni 2001 flooding enable auto
nv set bridge domain br_default vlan 201 vni 2001 mac-learning off
```

```
nv set evpn enable on
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan mac-learning off
nv set nve vxlan source address 27.0.0.4
nv set router bgp enable on
nv set system global anycast-mac 44:38:39:42:42:07
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on

nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp autonomous-system 63642
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor p0_if peer-group fabric
nv set vrf default router bgp neighbor p0_if type unnumbered
nv set vrf default router bgp neighbor p1_if peer-group fabric
nv set vrf default router bgp neighbor p1_if type unnumbered
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast policy outbound route-map
MY_ORIGIN_ASPATH_ONLY
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast policy outbound route-map
MY_ORIGIN_ASPATH_ONLY
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp router-id 27.0.0.4

nv set interface lo ip address 2001:c000:10ff:f00d::4/128
nv set interface lo ip address 27.0.0.4/32
nv set interface lo type loopback
nv set interface p0_if,p1_if,pf0hpf_if,pf1hpf_if type swp
nv set interface pf0hpf_if bridge domain br_default access 200
nv set interface pf1hpf_if bridge domain br_default access 201

nv set interface vlan200-201 base-interface br_default
nv set interface vlan200-201 ip ipv4 forward on
nv set interface vlan200-201 ip ipv6 forward on
nv set interface vlan200-201 ip vrr enable on
nv set interface vlan200-201 ip vrr state up
nv set interface vlan200-201 link mtu 9050
nv set interface vlan200-201 type svi
nv set interface vlan200 ip address 2001:cafe:1ead::3/64
nv set interface vlan200 ip address 45.3.0.2/24
nv set interface vlan200 ip vrr address 2001:cafe:1ead::1/64
nv set interface vlan200 ip vrr address 45.3.0.1/24
nv set interface vlan200 vlan 200
nv set interface vlan201 ip address 2001:cafe:1ead:1::3/64
nv set interface vlan201 ip address 45.3.1.2/24
nv set interface vlan201 ip vrr address 2001:cafe:1ead:1::1/64
nv set interface vlan201 ip vrr address 45.3.1.1/24
nv set interface vlan201 vlan 201
```

Sample NVUE Configuration for L2 EVPN with Host-facing Ports as Trunk

The following is a sample NVUE configuration which has L2-VNIs (**2000** , **2001**) for EVPN bridging on BlueField.

```
nv set bridge domain br_default encap 802.1Q
nv set bridge domain br_default type vlan-aware
nv set bridge domain br_default vlan 200 vni 2000 flooding enable auto
nv set bridge domain br_default vlan 200 vni 2000 mac-learning off
nv set bridge domain br_default vlan 201 vni 2001 flooding enable auto
nv set bridge domain br_default vlan 201 vni 2001 mac-learning off

nv set evpn enable on
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan mac-learning off
nv set nve vxlan source address 27.0.0.4
nv set router bgp enable on
nv set system global anycast-mac 44:38:39:42:42:07
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on

nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp autonomous-system 63642
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor p0_if peer-group fabric
nv set vrf default router bgp neighbor p0_if type unnumbered
nv set vrf default router bgp neighbor p1_if peer-group fabric
nv set vrf default router bgp neighbor p1_if type unnumbered
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast policy outbound route-map
MY_ORIGIN_ASPATH_ONLY
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast policy outbound route-map
MY_ORIGIN_ASPATH_ONLY
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp router-id 27.0.0.4

nv set interface lo ip address 2001:c000:10ff:f00d::4/128
nv set interface lo ip address 27.0.0.4/32
nv set interface lo type loopback
```

```
nv set interface p0_if,p1_if,pf0hpf_if,pf1hpf_if type swp
nv set interface pf0hpf_if bridge domain br_default
nv set interface pf1hpf_if bridge domain br_default

nv set interface vlan200-201 base-interface br_default
nv set interface vlan200-201 ip ipv4 forward on
nv set interface vlan200-201 ip ipv6 forward on
nv set interface vlan200-201 ip vrr enable on
nv set interface vlan200-201 ip vrr state up
nv set interface vlan200-201 link mtu 9050
nv set interface vlan200-201 type svi
nv set interface vlan200 ip address 2001:cafe:1ead::3/64
nv set interface vlan200 ip address 45.3.0.2/24
nv set interface vlan200 ip vrr address 2001:cafe:1ead::1/64
nv set interface vlan200 ip vrr address 45.3.0.1/24
nv set interface vlan200 vlan 200
nv set interface vlan201 ip address 2001:cafe:1ead:1::3/64
nv set interface vlan201 ip address 45.3.1.2/24
nv set interface vlan201 ip vrr address 2001:cafe:1ead:1::1/64
nv set interface vlan201 ip vrr address 45.3.1.1/24
nv set interface vlan201 vlan 201
```

Sample Flat Files Configuration for L2 EVPN with Host-facing Ports as Access

The following is a sample flat files configuration which has L2-VNIs (**vx-2000** , **vx-2001**) for EVPN bridging on BlueField. This file is located at **/etc/network/interfaces** :

```
auto lo
iface lo inet loopback
    address 2001:c000:10ff:f00d::4/128
    address 27.0.0.4/32
    vxlan-local-tunnelip 27.0.0.4

auto p0_if
iface p0_if

auto p1_if
iface p1_if

auto pf0hpf_if
iface pf0hpf_if
    bridge-access 200

auto pf1hpf_if
iface pf1hpf_if
    bridge-access 201

auto vlan200
iface vlan200
    address 2001:cafe:1ead::3/64
    address 45.3.0.2/24
    mtu 9050
    address-virtual 00:00:5e:00:01:01 2001:cafe:1ead::1/64 45.3.0.1/24
    vlan-raw-device br_default
    vlan-id 200

auto vlan201
iface vlan201
    address 2001:cafe:1ead:1::3/64
    address 45.3.1.2/24
    mtu 9050
    address-virtual 00:00:5e:00:01:01 2001:cafe:1ead:1::1/64 45.3.1.1/24
    vlan-raw-device br_default
    vlan-id 201

auto vxlan48
iface vxlan48
    bridge-vlan-vni-map 200=2000 201=2001
217=2017
    bridge-learning off

auto br_default
iface br_default
    bridge-ports pf0hpf_if pf1hpf_if vxlan48
    bridge-vlan-aware yes
    bridge-vids 200 201
    bridge-pvid 1
```

Sample Flat Files Configuration for L2 EVPN with Host-facing Ports as Trunk

The following is a sample flat files configuration which has L2-VNIs (**vx-2000** , **vx-2001**) for EVPN bridging on BlueField. This file is located at **/etc/network/interfaces** :

```
auto lo
iface lo inet loopback
    address 2001:c000:10ff:f00d::4/128
    address 27.0.0.4/32
    vxlan-local-tunnelip 27.0.0.4
```

```
auto p0_if
iface p0_if

auto p1_if
iface p1_if

auto pf0hpf_if
iface pf0hpf_if

auto pf1hpf_if
iface pf1hpf_if

auto vlan200
iface vlan200
    address 2001:cafe:1ead::3/64
    address 45.3.0.2/24
    mtu 9050
    address-virtual 00:00:5e:00:01:01 2001:cafe:1ead::1/64 45.3.0.1/24
    vlan-raw-device br_default
    vlan-id 200

auto vlan201
iface vlan201
    address 2001:cafe:1ead:1::3/64
    address 45.3.1.2/24
    mtu 9050
    address-virtual 00:00:5e:00:01:01 2001:cafe:1ead:1::1/64 45.3.1.1/24
    vlan-raw-device br_default
    vlan-id 201

auto vxlan48
iface vxlan48
    bridge-vlan-vni-map 200=2000 201=2001
217=2017
    bridge-learning off

auto br_default
iface br_default
    bridge-ports pf0hpf_if pf1hpf_if vxlan48
    bridge-vlan-aware yes
    bridge-vids 200 201
    bridge-pvid 1
```

This file tells the **frr** package which daemon to start and is located at **/etc/frr/daemons**:

```
bgpd=yes
ospfd=no
ospf6d=no
isisd=no
pimd=no
ldpd=no
pbrd=no
vrrpd=no
fabricd=no
nhrpd=no
eigrpd=no
babeld=no
sharpd=no
fabricd=no
ripngd=no
ripd=no

vtysh_enable=yes
zebra_options="  -M cumulus_mlag -M snmp -A 127.0.0.1 -s 90000000"
bgpd_options="   -M snmp -A 127.0.0.1"
ospfd_options="  -M snmp -A 127.0.0.1"
ospf6d_options=" -M snmp -A ::1"
ripd_options="   -A 127.0.0.1"
ripngd_options=" -A ::1"
isisd_options="  -A 127.0.0.1"
pimd_options="   -A 127.0.0.1"
ldpd_options="   -A 127.0.0.1"
nhrpd_options="  -A 127.0.0.1"
eigrpd_options=" -A 127.0.0.1"
babeld_options=" -A 127.0.0.1"
sharpd_options=" -A 127.0.0.1"
pbrd_options="   -A 127.0.0.1"
staticd_options="-A 127.0.0.1"
fabricd_options="-A 127.0.0.1"
vrrpd_options="  -A 127.0.0.1"

frr_profile="datacenter"
```

FRR configuration file is located at **/etc/frr/frr.conf**:

```
!---- Cumulus Defaults ----
frr defaults datacenter
log syslog informational
no zebra nexthop kernel enable
vrf default
outer bgp 63642 vrf default
bgp router-id 27.0.0.4
bgp bestpath as-path multipath-relax
timers bgp 3 9
bgp deterministic-med
```

```
! Neighbors
neighbor fabric peer-group
neighbor fabric remote-as external
neighbor fabric timers 3 9
neighbor fabric timers connect 10
neighbor fabric advertisement-interval 0
neighbor p0_if interface peer-group fabric
neighbor p1_if interface peer-group fabric
address-family ipv4 unicast
maximum-paths ibgp 64
maximum-paths 64
distance bgp 20 200 200
neighbor fabric activate
exit-address-family
address-family ipv6 unicast
maximum-paths ibgp 64
maximum-paths 64
distance bgp 20 200 200
neighbor fabric activate
exit-address-family
address-family l2vpn evpn
advertise-all-vni
neighbor fabric activate
exit-address-family
```

8.9.5.3.3.4 Layer-3 EVPN with Symmetric Routing

In distributed symmetric routing, each VXLAN endpoint (VTEP) acts as a layer-3 gateway, performing routing for its attached hosts. However, both the ingress VTEP and egress VTEP route the packets (similar to traditional routing behavior of routing to a next-hop router). In a VXLAN encapsulated packet, the inner destination MAC address is the router MAC address of the egress VTEP to indicate that the egress VTEP is the next hop and that it must also perform the routing.

All routing happens in the context of a tenant (VRF). For a packet that the ingress VTEP receives from a locally attached host, the SVI interface corresponding to the VLAN determines the VRF. For a packet that the egress VTEP receives over the VXLAN tunnel, the VNI in the packet has to specify the VRF. For symmetric routing, this is a VNI corresponding to the tenant and is different from either the source VNI or the destination VNI. This VNI is a layer-3 VNI or interconnecting VNI. The regular VNI, which maps a VLAN, is the layer-2 VNI.

Refer to the Cumulus Linux Symmetric Routing.

HBN uses a one-to-one mapping between an L3 VNI and a tenant (VRF).

The VRF to L3 VNI mapping must be consistent across all VTEPs.

An L3 VNI and an L2 VNI cannot have the same ID.

In an EVPN symmetric routing configuration, when the switch announces a type-2 (MAC/IP) route, in addition to containing two VNIs (L2 and L3 VNIs), the route also contains separate route targets (RTs) for L2 and L3. The L3 RT associates the route with the tenant VRF. By default, this is auto-derived using the L3 VNI instead of the L2 VNI. However, this is configurable.

Sample NVUE Configuration for L3 EVPN with Host-facing Ports as Access

If using NVUE to configure EVPN symmetric routing, the following is a sample configuration using NVUE commands:

```
nv set bridge domain br_default vlan 111 vni 1000111
nv set bridge domain br_default vlan 112 vni 1000112
nv set bridge domain br_default vlan 213 vni 1000213
nv set bridge domain br_default vlan 214 vni 1000214
nv set evpn enable on
nv set interface lo ip address 6.0.0.19/32
nv set interface lo type loopback
```

```
nv set interface p0_if description 'alias p0_if to leaf-21 swp3'
nv set interface p0_if,p1_if,pf0hpf_if,pf0vf0_if,pf1hpf_if,pf1vf0_if type swp
nv set interface p1_if description 'alias p1_if to leaf-22 swp3'
nv set interface pf0hpf_if bridge domain br_default access 111
nv set interface pf0hpf_if description 'alias pf0hpf_if to host-211 ens2f0np0'
nv set interface pf0vf0_if bridge domain br_default access 112
nv set interface pf0vf0_if description 'alias pf0vf0_if to host-211 ens2f0np0v0'
nv set interface pf1hpf_if bridge domain br_default access 213
nv set interface pf1hpf_if description 'alias pf1hpf_if to host-211 ens2f1np1'
nv set interface pf1vf0_if bridge domain br_default access 214
nv set interface pf1vf0_if description 'alias pf1vf0_if to host-211 ens2f1np0v0'
nv set interface vlan111 ip address 60.1.1.21/24
nv set interface vlan111 ip address 2060:1:1:1::21/64
nv set interface vlan111 ip vrr address 60.1.1.250/24
nv set interface vlan111 ip vrr address 2060:1:1:1::250/64
nv set interface vlan111 vlan 111
nv set interface vlan111,213 ip vrf vrf2
nv set interface vlan111-112,213-214 ip vrr enable on
nv set interface vlan111-112,213-214 ip vrr mac-address 00:00:5e:00:01:01
nv set interface vlan111-112,213-214 ip ipv4 forward on
nv set interface vlan111-112,213-214 ip ipv6 forward on
nv set interface vlan111-112,213-214 type svi
nv set interface vlan112 ip address 50.1.1.21/24
nv set interface vlan112 ip address 2050:1:1:1::21/64
nv set interface vlan112 ip vrr address 50.1.1.250/24
nv set interface vlan112 ip vrr address 2050:1:1:1::250/64
nv set interface vlan112 vlan 112
nv set interface vlan112,214 ip vrf vrf1
nv set interface vlan213 ip address 60.1.210.21/24
nv set interface vlan213 ip address 2060:1:1:210::21/64
nv set interface vlan213 ip vrr address 60.1.210.250/24
nv set interface vlan213 ip vrr address 2060:1:1:210::250/64
nv set interface vlan213 vlan 213
nv set interface vlan214 ip address 50.1.210.21/24
nv set interface vlan214 ip address 2050:1:1:210::21/64
nv set interface vlan214 ip vrr address 50.1.210.250/24
nv set interface vlan214 ip vrr address 2050:1:1:210::250/64
nv set interface vlan214 vlan 214
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan source address 6.0.0.19
nv set platform
nv set router bgp enable on
nv set router policy route-map ALLOW_LOBR rule 10 action permit
nv set router policy route-map ALLOW_LOBR rule 10 match interface lo
nv set router policy route-map ALLOW_LOBR rule 20 action permit
nv set router policy route-map ALLOW_LOBR rule 20 match interface br_default
nv set router policy route-map ALLOW_VRF1 rule 10 action permit
nv set router policy route-map ALLOW_VRF1 rule 10 match interface vrf1
nv set router policy route-map ALLOW_VRF2 rule 10 action permit
nv set router policy route-map ALLOW_VRF2 rule 10 match interface vrf2
nv set router vrr enable on
nv set system global system-mac 00:01:00:00:1e:03
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast multipaths ebgp 16
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected route-map ALLOW_LOBR
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp autonomous-system 650019
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor p0_if address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp neighbor p0_if address-family l2vpn-evpn enable on
nv set vrf default router bgp neighbor p0_if peer-group TOR_LEAF_SPINE
nv set vrf default router bgp neighbor p0_if remote-as external
nv set vrf default router bgp neighbor p0_if type unnumbered
nv set vrf default router bgp neighbor p1_if address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp neighbor p1_if address-family l2vpn-evpn enable on
nv set vrf default router bgp neighbor p1_if peer-group TOR_LEAF_SPINE
nv set vrf default router bgp neighbor p1_if remote-as external
nv set vrf default router bgp neighbor p1_if type unnumbered
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp path-selection routerid-compare on
nv set vrf default router bgp peer-group TOR_LEAF_SPINE address-family ipv4-unicast enable on
nv set vrf default router bgp router-id 6.0.0.19
nv set vrf vrf1 evpn enable on
nv set vrf vrf1 evpn vni 104001
nv set vrf vrf1 loopback ip address 50.1.21.21/32
nv set vrf vrf1 loopback ip address 2050:50:50:21::21/128
nv set vrf vrf1 router bgp address-family ipv4-unicast enable on
nv set vrf vrf1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf vrf1 router bgp address-family ipv4-unicast redistribute connected route-map ALLOW_VRF1
nv set vrf vrf1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf vrf1 router bgp address-family ipv6-unicast enable on
nv set vrf vrf1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf vrf1 router bgp address-family ipv6-unicast redistribute connected route-map ALLOW_VRF1
nv set vrf vrf1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf vrf1 router bgp autonomous-system 650019
nv set vrf vrf1 router bgp enable on
nv set vrf vrf1 router bgp router-id 50.1.21.21
nv set vrf vrf2 evpn enable on
nv set vrf vrf2 evpn vni 104002
nv set vrf vrf2 loopback ip address 60.1.21.21/32
nv set vrf vrf2 loopback ip address 2060:60:60:21::21/128
nv set vrf vrf2 router bgp address-family ipv4-unicast enable on
nv set vrf vrf2 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf vrf2 router bgp address-family ipv4-unicast redistribute connected route-map ALLOW_VRF2
nv set vrf vrf2 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf vrf2 router bgp address-family ipv6-unicast enable on
nv set vrf vrf2 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf vrf2 router bgp address-family ipv6-unicast redistribute connected route-map ALLOW_VRF2
nv set vrf vrf2 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf vrf2 router bgp autonomous-system 650019
nv set vrf vrf2 router bgp enable on
nv set vrf vrf2 router bgp router-id 60.1.21.21
```

Sample NVUE Configuration for L3 EVPN with Host-facing Ports as Trunk

If using NVUE to configure EVPN symmetric routing, the following is a sample configuration using NVUE commands:

```
nv set bridge domain br_default vlan 111 vni 1000111
nv set bridge domain br_default vlan 112 vni 1000112
nv set bridge domain br_default vlan 213 vni 1000213
nv set bridge domain br_default vlan 214 vni 1000214
nv set evpn enable on
nv set interface lo ip address 6.0.0.19/32
nv set interface lo type loopback
nv set interface p0_if description 'alias p0_if to leaf-21 swp3'
nv set interface p0_if,p1_if,pf0hpf_if,pf0vf0_if,pf1hpf_if,pf1vf0_if type swp
nv set interface p1_if description 'alias p1_if to leaf-22 swp3'
nv set interface pf0hpf_if bridge domain br_default
nv set interface pf0hpf_if description 'alias pf0hpf_if to host-211 ens2f0np0'
nv set interface pf0vf0_if bridge domain br_default
nv set interface pf0vf0_if description 'alias pf0vf0_if to host-211 ens2f0np0v0'
nv set interface pf1hpf_if bridge domain br_default
nv set interface pf1hpf_if description 'alias pf1hpf_if to host-211 ens2f1np1'
nv set interface pf1vf0_if bridge domain br_default
nv set interface pf1vf0_if description 'alias pf1vf0_if to host-211 ens2f1np0v0'
nv set interface vlan111 ip address 60.1.1.21/24
nv set interface vlan111 ip address 2060:1:1:1::21/64
nv set interface vlan111 ip vrr address 60.1.1.250/24
nv set interface vlan111 ip vrr address 2060:1:1:1::250/64
nv set interface vlan111 vlan 111
nv set interface vlan111,213 ip vrf vrf2
nv set interface vlan111-112,213-214 ip vrr enable on
nv set interface vlan111-112,213-214 ip vrr mac-address 00:00:5e:00:01:01
nv set interface vlan111-112,213-214 ip ipv4 forward on
nv set interface vlan111-112,213-214 ip ipv6 forward on
nv set interface vlan111-112,213-214 type svi
nv set interface vlan112 ip address 50.1.1.21/24
nv set interface vlan112 ip address 2050:1:1:1::21/64
nv set interface vlan112 ip vrr address 50.1.1.250/24
nv set interface vlan112 ip vrr address 2050:1:1:1::250/64
nv set interface vlan112 vlan 112
nv set interface vlan112,214 ip vrf vrf1
nv set interface vlan213 ip address 60.1.210.21/24
nv set interface vlan213 ip address 2060:1:1:210::21/64
nv set interface vlan213 ip vrr address 60.1.210.250/24
nv set interface vlan213 ip vrr address 2060:1:1:210::250/64
nv set interface vlan213 vlan 213
nv set interface vlan214 ip address 50.1.210.21/24
nv set interface vlan214 ip address 2050:1:1:210::21/64
nv set interface vlan214 ip vrr address 50.1.210.250/24
nv set interface vlan214 ip vrr address 2050:1:1:210::250/64
nv set interface vlan214 vlan 214
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan source address 6.0.0.19
nv set platform
nv set router bgp enable on
nv set router policy route-map ALLOW_LOBR rule 10 action permit
nv set router policy route-map ALLOW_LOBR rule 10 match interface lo
nv set router policy route-map ALLOW_LOBR rule 20 action permit
nv set router policy route-map ALLOW_LOBR rule 20 match interface br_default
nv set router policy route-map ALLOW_VRF1 rule 10 action permit
nv set router policy route-map ALLOW_VRF1 rule 10 match interface vrf1
nv set router policy route-map ALLOW_VRF2 rule 10 action permit
nv set router policy route-map ALLOW_VRF2 rule 10 match interface vrf2
nv set router vrr enable on
nv set system global system-mac 00:01:00:00:1e:03
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast multipaths ebgp 16
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected route-map ALLOW_LOBR
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp autonomous-system 650019
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor p0_if address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp neighbor p0_if address-family l2vpn-evpn enable on
nv set vrf default router bgp neighbor p0_if peer-group TOR_LEAF_SPINE
nv set vrf default router bgp neighbor p0_if remote-as external
nv set vrf default router bgp neighbor p0_if type unnumbered
nv set vrf default router bgp neighbor p1_if address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp neighbor p1_if address-family l2vpn-evpn enable on
nv set vrf default router bgp neighbor p1_if peer-group TOR_LEAF_SPINE
nv set vrf default router bgp neighbor p1_if remote-as external
nv set vrf default router bgp neighbor p1_if type unnumbered
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp path-selection routerid-compare on
nv set vrf default router bgp peer-group TOR_LEAF_SPINE address-family ipv4-unicast enable on
nv set vrf default router bgp router-id 6.0.0.19
nv set vrf vrf1 evpn enable on
nv set vrf vrf1 evpn vni 104001
nv set vrf vrf1 loopback ip address 50.1.21.21/32
nv set vrf vrf1 loopback ip address 2050:50:50:21::21/128
nv set vrf vrf1 router bgp address-family ipv4-unicast enable on
nv set vrf vrf1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf vrf1 router bgp address-family ipv4-unicast redistribute connected route-map ALLOW_VRF1
nv set vrf vrf1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf vrf1 router bgp address-family ipv6-unicast enable on
nv set vrf vrf1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf vrf1 router bgp address-family ipv6-unicast redistribute connected route-map ALLOW_VRF1
nv set vrf vrf1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf vrf1 router bgp autonomous-system 650019
nv set vrf vrf1 router bgp enable on
```

```
nv set vrf vrf1 router bgp router-id 50.1.21.21
nv set vrf vrf2 evpn enable on
nv set vrf vrf2 evpn vni 104002
nv set vrf vrf2 loopback ip address 60.1.21.21/32
nv set vrf vrf2 loopback ip address 2060:60:60:21::21/128
nv set vrf vrf2 router bgp address-family ipv4-unicast enable on
nv set vrf vrf2 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf vrf2 router bgp address-family ipv4-unicast redistribute connected route-map ALLOW_VRF2
nv set vrf vrf2 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf vrf2 router bgp address-family ipv6-unicast enable on
nv set vrf vrf2 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf vrf2 router bgp address-family ipv6-unicast redistribute connected route-map ALLOW_VRF2
nv set vrf vrf2 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf vrf2 router bgp autonomous-system 650019
nv set vrf vrf2 router bgp enable on
nv set vrf vrf2 router bgp router-id 60.1.21.21
```

Sample Flat Files Configuration for L3 EVPN for Host-facing Interfaces as Access

The following is a sample flat file configuration which has L2 VNIs and L3 VNIs for EVPN bridging and symmetric routing on BlueField.

This file is located in **/etc/network/interfaces**:

```
auto lo
iface lo inet loopback
    address 6.0.0.19/32
    vxlan-local-tunnelip 6.0.0.19

auto vrf1
iface vrf1
    address 2050:50:50:21::21/128
    address 50.1.21.21/32
    vrf-table auto

auto vrf2
iface vrf2
    address 2060:60:60:21::21/128
    address 60.1.21.21/32
    vrf-table auto

auto p0_if
iface p0_if
    alias alias p0_if to leaf-21 swp3

auto p1_if
iface p1_if
    alias alias p1_if to leaf-22 swp3

auto pf0hpf_if
iface pf0hpf_if
    alias alias pf0hpf_if to host-211 ens2f0np0
    bridge-access 111

auto pf0vf0_if
iface pf0vf0_if
    alias alias pf0vf0_if to host-211 ens2f0np0v0
    bridge-access 112

auto pf1hpf_if
iface pf1hpf_if
    alias alias pf1hpf_if to host-211 ens2f1np1
    bridge-access 213

auto pf1vf0_if
iface pf1vf0_if
    alias alias pf1vf0_if to host-211 ens2f1np0v0
    bridge-access 214

auto vlan111
iface vlan111
    address 2060:1:1:1::21/64
    address 60.1.1.21/24
    address-virtual 00:00:5e:00:01:01 2060:1:1:1::250/64 60.1.1.250/24
    hwaddress 00:01:00:00:1e:03
    vrf vrf2
    vlan-raw-device br_default
    vlan-id 111

auto vlan112
iface vlan112
    address 2050:1:1:1::21/64
    address 50.1.1.21/24
    address-virtual 00:00:5e:00:01:01 2050:1:1:1::250/64 50.1.1.250/24
    hwaddress 00:01:00:00:1e:03
    vrf vrf1
    vlan-raw-device br_default
    vlan-id 112

auto vlan213
iface vlan213
    address 2060:1:1:210::21/64
    address 60.1.210.21/24
    address-virtual 00:00:5e:00:01:01 2060:1:1:210::250/64 60.1.210.250/24
    hwaddress 00:01:00:00:1e:03
    vrf vrf2
    vlan-raw-device br_default
```

```
    vlan-id 213

auto vlan214
iface vlan214
    address 2050:1:1:210::21/64
    address 50.1.210.21/24
    address-virtual 00:00:5e:00:01:01 2050:1:1:210::250/64 50.1.210.250/24
    hwaddress 00:01:00:00:1e:03
    vrf vrf1
    vlan-raw-device br_default
    vlan-id 214

auto vlan4058_l3
iface vlan4058_l3
    vrf vrf1
    vlan-raw-device br_default
    address-virtual none
    vlan-id 4058

auto vlan4059_l3
iface vlan4059_l3
    vrf vrf2
    vlan-raw-device br_default
    address-virtual none
    vlan-id 4059

auto vxlan48
iface vxlan48
    bridge-vlan-vni-map 111=1000111 112=1000112 213=1000213 214=1000214 4058=104001 4059=104002
    bridge-learning off

auto br_default
iface br_default
    bridge-ports pf0hpf_if pf0vf0_if pf1hpf_if pf1vf0_if vxlan48
    hwaddress 00:01:00:00:1e:03
    bridge-vlan-aware yes
    bridge-vids 111 112 213 214
    bridge-pvid 1
```

Sample Flat Files Configuration for L3 EVPN for Host-facing Interfaces as Trunk

The following is a sample flat file configuration which has L2 VNIs and L3 VNIs for EVPN bridging and symmetric routing on BlueField.

This file is located in **/etc/network/interfaces**:

```
auto lo
iface lo inet loopback
    address 6.0.0.19/32
    vxlan-local-tunnelip 6.0.0.19

auto vrf1
iface vrf1
    address 2050:50:50:21::21/128
    address 50.1.21.21/32
    vrf-table auto

auto vrf2
iface vrf2
    address 2060:60:60:21::21/128
    address 60.1.21.21/32
    vrf-table auto

auto p0_if
iface p0_if
    alias alias p0_if to leaf-21 swp3

auto p1_if
iface p1_if
    alias alias p1_if to leaf-22 swp3

auto pf0hpf_if
iface pf0hpf_if
    alias alias pf0hpf_if to host-211 ens2f0np0

auto pf0vf0_if
iface pf0vf0_if
    alias alias pf0vf0_if to host-211 ens2f0np0v0

auto pf1hpf_if
iface pf1hpf_if
    alias alias pf1hpf_if to host-211 ens2f1np1

auto pf1vf0_if
iface pf1vf0_if
    alias alias pf1vf0_if to host-211 ens2f1np0v0

auto vlan111
iface vlan111
    address 2060:1:1:1::21/64
    address 60.1.1.21/24
    address-virtual 00:00:5e:00:01:01 2060:1:1:1::250/64 60.1.1.250/24
    hwaddress 00:01:00:00:1e:03
    vrf vrf2
    vlan-raw-device br_default
    vlan-id 111
```

```
auto vlan112
iface vlan112
    address 2050:1:1:1::21/64
    address 50.1.1.21/24
    address-virtual 00:00:5e:00:01:01 2050:1:1:1::250/64 50.1.1.250/24
    hwaddress 00:01:00:00:1e:03
    vrf vrf1
    vlan-raw-device br_default
    vlan-id 112

auto vlan213
iface vlan213
    address 2060:1:1:210::21/64
    address 60.1.210.21/24
    address-virtual 00:00:5e:00:01:01 2060:1:1:210::250/64 60.1.210.250/24
    hwaddress 00:01:00:00:1e:03
    vrf vrf2
    vlan-raw-device br_default
    vlan-id 213

auto vlan214
iface vlan214
    address 2050:1:1:210::21/64
    address 50.1.210.21/24
    address-virtual 00:00:5e:00:01:01 2050:1:1:210::250/64 50.1.210.250/24
    hwaddress 00:01:00:00:1e:03
    vrf vrf1
    vlan-raw-device br_default
    vlan-id 214

auto vlan4058_l3
iface vlan4058_l3
    vrf vrf1
    vlan-raw-device br_default
    address-virtual none
    vlan-id 4058

auto vlan4059_l3
iface vlan4059_l3
    vrf vrf2
    vlan-raw-device br_default
    address-virtual none
    vlan-id 4059

auto vxlan48
iface vxlan48
    bridge-vlan-vni-map 111=1000111 112=1000112 213=1000213 214=1000214 4058=104001 4059=104002
    bridge-learning off

auto br_default
iface br_default
    bridge-ports pf0hpf_if pf0vf0_if pf1hpf_if pf1vf0_if vxlan48
    hwaddress 00:01:00:00:1e:03
    bridge-vlan-aware yes
    bridge-vids 111 112 213 214
    bridge-pvid 1
```

The FRR configuration is located in **/etc/frr/frr.conf**:

```
frr version 8.4.3
frr defaults datacenter
hostname doca-hbn-service-bf3-s05-1-ipmi
log syslog informational
no zebra nexthop kernel enable
service integrated-vtysh-config
!
vrf vrf1
 vni 104001
exit-vrf
!
vrf vrf2
 vni 104002
exit-vrf
!
router bgp 650019
 bgp router-id 6.0.0.19
 bgp bestpath as-path multipath-relax
 bgp bestpath compare-routerid
 neighbor TOR_LEAF_SPINE peer-group
 neighbor TOR_LEAF_SPINE advertisement-interval 0
 neighbor TOR_LEAF_SPINE timers 3 9
 neighbor TOR_LEAF_SPINE timers connect 10
 neighbor p0_if interface peer-group TOR_LEAF_SPINE
 neighbor p0_if remote-as external
 neighbor p0_if advertisement-interval 0
 neighbor p0_if timers 3 9
 neighbor p0_if timers connect 10
 neighbor p1_if interface peer-group TOR_LEAF_SPINE
 neighbor p1_if remote-as external
 neighbor p1_if advertisement-interval 0
 neighbor p1_if timers 3 9
 neighbor p1_if timers connect 10
 !
 address-family ipv4 unicast
  redistribute connected route-map ALLOW_LOBR
  maximum-paths 16
  maximum-paths ibgp 64
 exit-address-family
 !
 address-family l2vpn evpn
```

```
  neighbor p0_if activate
  neighbor p1_if activate
  advertise-all-vni
 exit-address-family
exit
!
router bgp 650019 vrf vrf1
 bgp router-id 50.1.21.21
 !
 address-family ipv4 unicast
  redistribute connected route-map ALLOW_VRF1
  maximum-paths 64
  maximum-paths ibgp 64
 exit-address-family
 !
 address-family ipv6 unicast
  redistribute connected route-map ALLOW_VRF1
  maximum-paths 64
  maximum-paths ibgp 64
 exit-address-family
 !
 address-family l2vpn evpn
  advertise ipv4 unicast
  advertise ipv6 unicast
 exit-address-family
exit
!
router bgp 650019 vrf vrf2
 bgp router-id 60.1.21.21
 !
 address-family ipv4 unicast
  redistribute connected route-map ALLOW_VRF2
  maximum-paths 64
  maximum-paths ibgp 64
 exit-address-family
 !
 address-family ipv6 unicast
  redistribute connected route-map ALLOW_VRF2
  maximum-paths 64
  maximum-paths ibgp 64
 exit-address-family
 !
 address-family l2vpn evpn
  advertise ipv4 unicast
  advertise ipv6 unicast
 exit-address-family
exit
!
route-map ALLOW_LOBR permit 10
 match interface lo
exit
!
route-map ALLOW_LOBR permit 20
 match interface br_default
exit
!
route-map ALLOW_VRF1 permit 10
 match interface vrf1
exit
!
route-map ALLOW_VRF2 permit 10
 match interface vrf2
exit
```

8.9.5.3.3.5 Multi-hop eBGP Peering for EVPN (Route Server in Symmetric EVPN Routing)

eBGP multi-hop peering for EVPN support in a route server-like role in EVPN topology, allows the deployment of EVPN on any cloud that supports IP transport.

Route servers and BF/HBN VTEPs are connected via the IP cloud. That is:

- Switches in the cloud provider need not be EVPN-aware
- Switches in the provider fabric provide IPv4 and IPv6 transport and do not have to support EVPN

Sample Route Server Configuration for EVPN

The following is a sample configuration of an Ubuntu server running FRR 9.0 stable, configured as EVPN route server and an HBN VTEP that is peering to two spine switches for IP connectivity and 3 Route servers for EVPN overlay control.

```
root@sn1:/home/cumulus# uname -a
Linux sn1 5.15.0-88-generic #98-Ubuntu SMP Mon Oct 2 15:18:56 UTC 2023 x86_64 x86_64 x86_64 GNU/Linux
root@sn1:/home/cumulus# dpkg -l frr
Desired=Unknown/Install/Remove/Purge/Hold
```

```
| Status=Not/Inst/Conf-files/Unpacked/halF-conf/Half-inst/trig-aWait/Trig-pend
|/ Err?=(none)/Reinst-required (Status,Err: uppercase=bad)
||/ Name           Version              Architecture Description
+++-==============-====================-============-===============================================================
ii  frr            9.0.1-0~ubuntu22.04.1 amd64        FRRouting suite of internet protocols (BGP, OSPF, IS-IS, ...)
root@sn1:/home/cumulus#
```

FRR configuration (**frr.conf**):

```
sn1# sh run
Building configuration...

Current configuration:
!
frr version 9.0.1
frr defaults datacenter
hostname sn1
no ip forwarding
no ipv6 forwarding
service integrated-vtysh-config
!
router bgp 4200065507
 bgp router-id 6.0.0.7
 timers bgp 60 180
 neighbor rclients peer-group
 neighbor rclients remote-as external
 neighbor rclients ebgp-multihop 10
 neighbor rclients update-source lo
 neighbor rclients advertisement-interval 0
 neighbor rclients timers 3 9
 neighbor rclients timers connect 10
 neighbor rcsuper peer-group
 neighbor rcsuper remote-as external
 neighbor rcsuper advertisement-interval 0
 neighbor rcsuper timers 3 9
 neighbor rcsuper timers connect 10
 neighbor swp1 interface peer-group rcsuper
 bgp listen range 6.0.0.0/24 peer-group rclients
 !
 address-family ipv4 unicast
  redistribute connected
  neighbor fabric route-map pass in
  neighbor fabric route-map pass out
  no neighbor rclients activate
  maximum-paths 64
  maximum-paths ibgp 64
 exit-address-family
 !
 address-family l2vpn evpn
  neighbor rclients activate
  neighbor rcsuper activate
 exit-address-family
exit
!
route-map pass permit 10
 set community 11:11 additive
exit
!
end
sn1#
```

Interfaces configuration (**/etc/network/interfaces**):

```
root@sn1:/home/cumulus# ifquery -a
auto lo
iface lo inet loopback
    address 6.0.0.7/32

auto lo
iface lo inet loopback

auto swp1
iface swp1

auto eth0
iface eth0
    address 192.168.0.15/24
    gateway 192.168.0.2

root@sn1:/home/cumulus#
```

Sample HBN Configuration for Deployments with EVPN Route Server

```
root@doca-hbn-service-bf2-s12-1-ipmi:/tmp# nv config show -o commands
nv set bridge domain br_default vlan 101 vni 10101
nv set bridge domain br_default vlan 102 vni 10102
nv set bridge domain br_default vlan 201 vni 10201
nv set bridge domain br_default vlan 202 vni 10202
nv set evpn enable on
nv set evpn route-advertise svi-ip off
```

```
nv set interface ilan3200 ip vrf internet1
nv set interface ilan3200 vlan 3200
nv set interface ilan3200,slan3201,vlan101-102,201-202,3001-3002 base-interface br_default
nv set interface ilan3200,slan3201,vlan101-102,201-202,3001-3002 type svi
nv set interface lo ip address 6.0.0.13/32
nv set interface lo ip address 2001::13/128
nv set interface lo type loopback
nv set interface p0_if,p1_if,pf0hpf_if,pf0vf0_if,pf0vf1_if,pf0vf2_if,pf0vf3_if,pf1hpf_if type swp
nv set interface pf0vf0_if bridge domain br_default access 101
nv set interface pf0vf1_if bridge domain br_default access 102
nv set interface pf0vf2_if bridge domain br_default access 201
nv set interface pf0vf3_if bridge domain br_default access 202
nv set interface slan3201 ip vrf special1
nv set interface slan3201 vlan 3201
nv set interface vlan101 ip address 21.1.0.13/16
nv set interface vlan101 ip address 2020:0:1:1::13/64
nv set interface vlan101 ip vrr address 21.1.0.250/16
nv set interface vlan101 ip vrr address 2020:0:1:1::250/64
nv set interface vlan101 ip vrr mac-address 00:00:01:00:00:65
nv set interface vlan101 vlan 101
nv set interface vlan101-102,201-202 ip vrr enable on
nv set interface vlan101-102,3001 ip vrf tenant1
nv set interface vlan102 ip address 21.2.0.13/16
nv set interface vlan102 ip address 2020:0:1:2::13/64
nv set interface vlan102 ip vrr address 21.2.0.250/16
nv set interface vlan102 ip vrr address 2020:0:1:2::250/64
nv set interface vlan102 ip vrr mac-address 00:00:01:00:00:66
nv set interface vlan102 vlan 102
nv set interface vlan201 ip address 22.1.0.13/16
nv set interface vlan201 ip address 2020:0:2:1::13/64
nv set interface vlan201 ip vrr address 22.1.0.250/16
nv set interface vlan201 ip vrr address 2020:0:2:1::250/64
nv set interface vlan201 ip vrr mac-address 00:00:02:00:00:c9
nv set interface vlan201 vlan 201
nv set interface vlan201-202,3002 ip vrf tenant2
nv set interface vlan202 ip address 22.2.0.13/16
nv set interface vlan202 ip address 2020:0:2:2::13/64
nv set interface vlan202 ip vrr address 22.2.0.250/16
nv set interface vlan202 ip vrr address 2020:0:2:2::250/64
nv set interface vlan202 ip vrr mac-address 00:00:02:00:00:ca
nv set interface vlan202 vlan 202
nv set interface vlan3001 vlan 3001
nv set interface vlan3002 vlan 3002
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan source address 6.0.0.13
nv set platform
nv set router bgp autonomous-system 4200065011
nv set router bgp enable on
nv set router bgp router-id 6.0.0.13
nv set router vrr enable on
nv set system config snippet
nv set system global
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor 6.0.0.7 peer-group rservers
nv set vrf default router bgp neighbor 6.0.0.7 type numbered
nv set vrf default router bgp neighbor 6.0.0.8 peer-group rservers
nv set vrf default router bgp neighbor 6.0.0.8 type numbered
nv set vrf default router bgp neighbor 6.0.0.9 peer-group rservers
nv set vrf default router bgp neighbor 6.0.0.9 type numbered
nv set vrf default router bgp neighbor p0_if peer-group fabric
nv set vrf default router bgp neighbor p0_if type unnumbered
nv set vrf default router bgp neighbor p1_if peer-group fabric
nv set vrf default router bgp neighbor p1_if type unnumbered
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on

nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp peer-group rservers address-family ipv4-unicast enable off
nv set vrf default router bgp peer-group rservers address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group rservers address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group rservers multihop-ttl 3
nv set vrf default router bgp peer-group rservers remote-as external
nv set vrf default router bgp peer-group rservers update-source lo
nv set vrf internet1 evpn enable on
nv set vrf internet1 evpn vni 42000
nv set vrf internet1 loopback ip address 8.1.0.13/32
nv set vrf internet1 loopback ip address 2008:0:1::13/64
nv set vrf internet1 router bgp address-family ipv4-unicast enable on
nv set vrf internet1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf internet1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf internet1 router bgp enable on
nv set vrf special1 evpn enable on
nv set vrf special1 evpn vni 42001
nv set vrf special1 loopback ip address 9.1.0.13/32
nv set vrf special1 loopback ip address 2009:0:1::13/64
nv set vrf special1 router bgp address-family ipv4-unicast enable on
nv set vrf special1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf special1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf special1 router bgp enable on
nv set vrf tenant1 evpn enable on
nv set vrf tenant1 evpn vni 30001
nv set vrf tenant1 router bgp address-family ipv4-unicast enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant1 router bgp enable on
nv set vrf tenant1 router bgp router-id 6.0.0.13
nv set vrf tenant2 evpn enable on
nv set vrf tenant2 evpn vni 30002
nv set vrf tenant2 router bgp address-family ipv4-unicast enable on
nv set vrf tenant2 router bgp address-family ipv4-unicast redistribute connected enable on
```

```
nv set vrf tenant2 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant2 router bgp enable on
nv set vrf tenant2 router bgp router-id 6.0.0.13
root@doca-hbn-service-bf2-s12-1-ipmi:/tmp#
```

Verifying BGP sessions in HBN:

```
doca-hbn-service-bf2-s12-1-ipmi# sh bgp sum

IPv4 Unicast Summary (VRF default):
BGP router identifier 6.0.0.13, local AS number 4200065011 vrf-id 0
BGP table version 20
RIB entries 21, using 4032 bytes of memory
Peers 2, using 40 KiB of memory
Peer groups 2, using 128 bytes of memory

Neighbor        V         AS   MsgRcvd   MsgSent   TblVer  InQ OutQ  Up/Down State/PfxRcd   PfxSnt Desc
spine11(p0_if)  4      65201     30617     30620        0    0    0 1d01h30m            9       11 N/A
spine12(p1_if)  4      65201     30620     30623        0    0    0 1d01h30m            9       11 N/A

Total number of neighbors 2

IPv6 Unicast Summary (VRF default):
BGP router identifier 6.0.0.13, local AS number 4200065011 vrf-id 0
BGP table version 0
RIB entries 0, using 0 bytes of memory
Peers 2, using 40 KiB of memory
Peer groups 2, using 128 bytes of memory

Neighbor        V         AS   MsgRcvd   MsgSent   TblVer  InQ OutQ  Up/Down State/PfxRcd   PfxSnt Desc
spine11(p0_if)  4      65201     30617     30620        0    0    0 1d01h30m            0        0 N/A
spine12(p1_if)  4      65201     30620     30623        0    0    0 1d01h30m            0        0 N/A

Total number of neighbors 2

L2VPN EVPN Summary (VRF default):
BGP router identifier 6.0.0.13, local AS number 4200065011 vrf-id 0
BGP table version 0
RIB entries 79, using 15 KiB of memory
Peers 3, using 60 KiB of memory
Peer groups 2, using 128 bytes of memory

Neighbor        V         AS   MsgRcvd   MsgSent   TblVer  InQ OutQ  Up/Down State/PfxRcd   PfxSnt Desc
sn1(6.0.0.7)    4 4200065507     31410     31231        0    0    0 00:27:51           69       95 N/A
sn2(6.0.0.8)    4 4200065508     31169     31062        0    0    0 02:34:47           69       95 N/A
sn3(6.0.0.9)    4 4200065509     31285     31059        0    0    0 02:34:47           69       95 N/A

Total number of neighbors 3
doca-hbn-service-bf2-s12-1-ipmi#
```

The command output shows that the HBN has BGP sessions with spine switches exchanging IPv4/IPv6 unicast. BGP sessions with route servers **sn1**, **sn2**, and **sn3** only exchanging L2VPN EVPN AFI/SAFI.

8.9.5.3.3.6 Downstream VNI (DVNI)

Downstream VNI (symmetric EVPN route leaking) allows users to leak remote EVPN routes without having the source tenant VRF locally configured. A common use case is where upstream switches learn the L3VNI from downstream leaf switches and impose the learned L3VNI to the traffic VXLAN routed to the associated VRF. This eliminates the need to configure L3VNI-SVI interfaces on all leaf switches and enables shared service and hub-and-spoke scenarios.

To configure access to a shared service in a specific VRF, users must:

1. Configure route-target import statements, effectively leaking routes from remote tenants to the shared VRF.
2. Import shared VRF's route-target at the remote nodes.

The route target import or export statement takes the following format:

```
route-target import|export <asn>:<vni>
```

For example:

```
route-target import 65101:6000
```

For route target import statements, users can use **route-target import ANY:<vni>** for NVUE commands or **route-target import *:<vni>** in the **/etc/frr/frr.conf** file. **ANY** in NVUE commands or the asterisk (*****) in the **/etc/frr/frr.conf** file use any ASN (autonomous system number) as a wildcard.

The NVUE commands are as follows:

1. To configure a route import statement:

```
nv set vrf <vrf> router bgp route-import from-evpn route-target <asn>:<vni>
```

2. To configure a route export statement:

```
nv set vrf <vrf> router bgp route-export from-evpn route-target <asn>:<vni>
```

Important considerations when implementing DVNI configuration:

- EVPN symmetric mode supports downstream VNI with L3 VNIs and single VXLAN devices only
- You can configure multiple import and export route targets in a VRF
- You cannot leak (import) overlapping tenant prefixes into the same destination VRF

If symmetric EVPN configuration is using automatic import/export (which is often the case), when DVNI is configured, automatic import of a tenant's VNI is disabled which isolates the VRF from the tenant. To avoid this issue, add **route-import from-evpn route-target auto** to the command line.

DVNI Configurations for Shared Internet Service

Configuration example here considers a scenario where External/Internet connectivity is available via a firewall (FW), which is connected to a shared VRF (**vrf external** in this example).

The routes on super spine switches have **external** VRF configured in which the route-targets from remote tenants are imported.

On BlueField devices with HBN, a local tenant VRF imports route-target corresponding to the shared **external** VRF.

L3VNI:

Tenant	L3VNI	
tenant1	30001	On HBN VTEPs
tenant2	30002	On HBN VTEPs
tenant3	30003	On HBN VTEPs
tenant4	30004	On HBN VTEPs
tenant5	30005	On HBN VTEPs
tenant6	30006	On HBN VTEPs

Tenant	L3VNI	
external	60000	Configured on superspines and connects to external world

On BlueField devices with HBN, every tenant VRF on HBN one must import VNI of shared **external** VRF:

```
nv set vrf tenant1 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant1 router bgp route-import from-evpn route-target auto
nv set vrf tenant2 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant2 router bgp route-import from-evpn route-target auto
nv set vrf tenant3 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant3 router bgp route-import from-evpn route-target auto
nv set vrf tenant4 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant4 router bgp route-import from-evpn route-target auto
nv set vrf tenant5 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant5 router bgp route-import from-evpn route-target auto
nv set vrf tenant6 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant6 router bgp route-import from-evpn route-target auto
root@doca-hbn-service-bf3-s06-1-ipmi:/tmp#
```

On super spine switches (SS1 in this example), every remote tenant VRF that needs access to shared services has to be leaked to the shared **external** VRF.

```
nv set vrf external router bgp route-import from-evpn route-target ANY:30001
nv set vrf external router bgp route-import from-evpn route-target ANY:30002
nv set vrf external router bgp route-import from-evpn route-target ANY:30003
nv set vrf external router bgp route-import from-evpn route-target ANY:30004
nv set vrf external router bgp route-import from-evpn route-target ANY:30005
nv set vrf external router bgp route-import from-evpn route-target ANY:30006
nv set vrf external router bgp route-import from-evpn route-target auto
root@superspine1:mgmt:/home/cumulus#
```

All super spines in this case need this configuration.

DVNI Leaked Routes in VRF Table of HBN

> Each super spine here is advertising reachability providing 4-way overlay ECMP.

Kernel table for all tenant VRFs, showing the imported shared service:

```
root@doca-hbn-service-bf3-s06-1-ipmi:/tmp# ip -4 route show table all  6.0.0.4/32
6.0.0.4 table tenant1 proto bgp metric 20
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.12 ttl 0 tos 0 via 6.0.0.12 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.13 ttl 0 tos 0 via 6.0.0.13 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.14 ttl 0 tos 0 via 6.0.0.14 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.15 ttl 0 tos 0 via 6.0.0.15 dev vxlan48 weight 1 onlink
6.0.0.4 table tenant2 proto bgp metric 20
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.12 ttl 0 tos 0 via 6.0.0.12 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.13 ttl 0 tos 0 via 6.0.0.13 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.14 ttl 0 tos 0 via 6.0.0.14 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.15 ttl 0 tos 0 via 6.0.0.15 dev vxlan48 weight 1 onlink
6.0.0.4 table tenant3 proto bgp metric 20
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.12 ttl 0 tos 0 via 6.0.0.12 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.13 ttl 0 tos 0 via 6.0.0.13 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.14 ttl 0 tos 0 via 6.0.0.14 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.15 ttl 0 tos 0 via 6.0.0.15 dev vxlan48 weight 1 onlink
6.0.0.4 table tenant4 proto bgp metric 20
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.12 ttl 0 tos 0 via 6.0.0.12 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.13 ttl 0 tos 0 via 6.0.0.13 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.14 ttl 0 tos 0 via 6.0.0.14 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.15 ttl 0 tos 0 via 6.0.0.15 dev vxlan48 weight 1 onlink
6.0.0.4 table tenant5 proto bgp metric 20
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.12 ttl 0 tos 0 via 6.0.0.12 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.13 ttl 0 tos 0 via 6.0.0.13 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.14 ttl 0 tos 0 via 6.0.0.14 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.15 ttl 0 tos 0 via 6.0.0.15 dev vxlan48 weight 1 onlink
6.0.0.4 table tenant6 proto bgp metric 20
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.12 ttl 0 tos 0 via 6.0.0.12 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.13 ttl 0 tos 0 via 6.0.0.13 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.14 ttl 0 tos 0 via 6.0.0.14 dev vxlan48 weight 1 onlink
    nexthop  encap ip id 60000 src 0.0.0.0 dst 6.0.0.15 ttl 0 tos 0 via 6.0.0.15 dev vxlan48 weight 1 onlink
root@doca-hbn-service-bf3-s06-1-ipmi:/tmp#
```

FRR RIB table:

```
root@doca-hbn-service-bf3-s06-1-ipmi:/tmp# vtysh

Hello, this is FRRouting (version 8.4.3).
Copyright 1996-2005 Kunihiro Ishiguro, et al.

doca-hbn-service-bf3-s06-1-ipmi# sh ip route vrf tenant1
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, E - EIGRP, N - NHRP,
       T - Table, A - Babel, D - SHARP, F - PBR, f - OpenFabric,
       Z - FRR,
       > - selected route, * - FIB route, q - queued, r - rejected, b - backup
       t - trapped, o - offload failure

VRF tenant1:
K>* 0.0.0.0/0 [255/8192] unreachable (ICMP unreachable), 00:10:36
B>* 6.0.0.4/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
  *                   via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
  *                   via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
  *                   via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
B>* 6.6.0.12/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
B>* 6.6.0.13/32 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
B>* 6.6.0.14/32 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
B>* 6.6.0.15/32 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
B>* 7.1.0.6/32 [20/0] via 6.0.0.6, vlan4052_l3 onlink, weight 1, 00:05:37
C>* 7.1.0.16/32 is directly connected, tenant1, 00:10:36
B>* 7.1.0.18/32 [20/0] via 6.0.0.18, vlan4052_l3 onlink, weight 1, 00:05:37
B>* 7.1.0.20/32 [20/0] via 6.0.0.20, vlan4052_l3 onlink, weight 1, 00:05:37
C>* 21.1.0.0/16 is directly connected, vlan101, 00:10:36
C * 21.1.0.0/16 [0/1024] is directly connected, vlan101-v0, 00:10:36
C * 21.2.0.0/16 [0/1024] is directly connected, vlan102-v0, 00:10:36
C>* 21.2.0.0/16 is directly connected, vlan102, 00:10:36
B>* 101.12.4.0/24 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
B>* 101.13.4.0/24 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
B>* 101.14.4.0/24 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
B>* 101.15.4.0/24 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:38
doca-hbn-service-bf3-s06-1-ipmi# sh ip route vrf all
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, E - EIGRP, N - NHRP,
       T - Table, A - Babel, D - SHARP, F - PBR, f - OpenFabric,
       Z - FRR,
       > - selected route, * - FIB route, q - queued, r - rejected, b - backup
       t - trapped, o - offload failure

VRF default:
B>* 6.0.0.6/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:06:47
  *                   via fe80::202:ff:fe00:27, p1_if, weight 1, 00:06:47
B>* 6.0.0.7/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:05:48
  *                   via fe80::202:ff:fe00:27, p1_if, weight 1, 00:05:48
B>* 6.0.0.8/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:05:38
  *                   via fe80::202:ff:fe00:27, p1_if, weight 1, 00:05:38
B>* 6.0.0.9/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:05:28
  *                   via fe80::202:ff:fe00:27, p1_if, weight 1, 00:05:28
B>* 6.0.0.10/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:06:49
B>* 6.0.0.11/32 [20/0] via fe80::202:ff:fe00:27, p1_if, weight 1, 00:06:47
B>* 6.0.0.12/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:06:47
  *                   via fe80::202:ff:fe00:27, p1_if, weight 1, 00:06:47
B>* 6.0.0.13/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:06:47
  *                   via fe80::202:ff:fe00:27, p1_if, weight 1, 00:06:47
B>* 6.0.0.14/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:06:47
  *                   via fe80::202:ff:fe00:27, p1_if, weight 1, 00:06:47
B>* 6.0.0.15/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:06:47
  *                   via fe80::202:ff:fe00:27, p1_if, weight 1, 00:06:47
C>* 6.0.0.16/32 is directly connected, lo, 00:10:42
B>* 6.0.0.18/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:06:47
  *                   via fe80::202:ff:fe00:27, p1_if, weight 1, 00:06:47
B>* 6.0.0.20/32 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:06:47
  *                   via fe80::202:ff:fe00:27, p1_if, weight 1, 00:06:47
B>* 192.168.0.0/24 [20/0] via fe80::202:ff:fe00:1f, p0_if, weight 1, 00:05:48
  *                       via fe80::202:ff:fe00:27, p1_if, weight 1, 00:05:48

VRF internet1:
K>* 0.0.0.0/0 [255/8192] unreachable (ICMP unreachable), 00:10:42
B>* 8.1.0.6/32 [20/0] via 6.0.0.6, vlan4004_l3 onlink, weight 1, 00:05:43
C>* 8.1.0.16/32 is directly connected, internet1, 00:10:42
B>* 8.1.0.18/32 [20/0] via 6.0.0.18, vlan4004_l3 onlink, weight 1, 00:05:43
B>* 8.1.0.20/32 [20/0] via 6.0.0.20, vlan4004_l3 onlink, weight 1, 00:05:43

VRF mgmt:
K>* 0.0.0.0/0 [255/8192] unreachable (ICMP unreachable), 00:10:42
C>* 10.88.0.0/16 is directly connected, eth0, 00:10:42

VRF special1:
K>* 0.0.0.0/0 [255/8192] unreachable (ICMP unreachable), 00:10:42
B>* 9.1.0.6/32 [20/0] via 6.0.0.6, vlan4033_l3 onlink, weight 1, 00:05:43
C>* 9.1.0.16/32 is directly connected, special1, 00:10:42
B>* 9.1.0.18/32 [20/0] via 6.0.0.18, vlan4033_l3 onlink, weight 1, 00:05:43
B>* 9.1.0.20/32 [20/0] via 6.0.0.20, vlan4033_l3 onlink, weight 1, 00:05:43

VRF tenant1:
K>* 0.0.0.0/0 [255/8192] unreachable (ICMP unreachable), 00:10:42
B>* 6.0.0.4/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.12/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.13/32 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.14/32 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.15/32 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 7.1.0.6/32 [20/0] via 6.0.0.6, vlan4052_l3 onlink, weight 1, 00:05:43
C>* 7.1.0.16/32 is directly connected, tenant1, 00:10:42
B>* 7.1.0.18/32 [20/0] via 6.0.0.18, vlan4052_l3 onlink, weight 1, 00:05:43
B>* 7.1.0.20/32 [20/0] via 6.0.0.20, vlan4052_l3 onlink, weight 1, 00:05:43
C>* 21.1.0.0/16 is directly connected, vlan101, 00:10:42
C * 21.1.0.0/16 [0/1024] is directly connected, vlan101-v0, 00:10:42
```

```
C * 21.2.0.0/16 [0/1024] is directly connected, vlan102-v0, 00:10:42
C>* 21.2.0.0/16 is directly connected, vlan102, 00:10:42
B>* 101.12.4.0/24 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.13.4.0/24 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.14.4.0/24 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.15.4.0/24 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44

VRF tenant2:
K>* 0.0.0.0/0 [255/8192] unreachable (ICMP unreachable), 00:10:42
B>* 6.0.0.4/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.12/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.13/32 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.14/32 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.15/32 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 7.2.0.6/32 [20/0] via 6.0.0.6, vlan4037_l3 onlink, weight 1, 00:05:43
C>* 7.2.0.16/32 is directly connected, tenant2, 00:10:42
B>* 7.2.0.18/32 [20/0] via 6.0.0.18, vlan4037_l3 onlink, weight 1, 00:05:43
B>* 7.2.0.20/32 [20/0] via 6.0.0.20, vlan4037_l3 onlink, weight 1, 00:05:43
C * 22.1.0.0/16 [0/1024] is directly connected, vlan201-v0, 00:10:42
C>* 22.1.0.0/16 is directly connected, vlan201, 00:10:42
C * 22.2.0.0/16 [0/1024] is directly connected, vlan202-v0, 00:10:42
C>* 22.2.0.0/16 is directly connected, vlan202, 00:10:42
B>* 101.12.4.0/24 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.13.4.0/24 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.14.4.0/24 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.15.4.0/24 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44

VRF tenant3:
K>* 0.0.0.0/0 [255/8192] unreachable (ICMP unreachable), 00:10:42
B>* 6.0.0.4/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.12/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.13/32 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.14/32 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.15/32 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 7.3.0.6/32 [20/0] via 6.0.0.6, vlan4022_l3 onlink, weight 1, 00:05:43
C>* 7.3.0.16/32 is directly connected, tenant3, 00:10:42
B>* 7.3.0.18/32 [20/0] via 6.0.0.18, vlan4022_l3 onlink, weight 1, 00:05:43
B>* 7.3.0.20/32 [20/0] via 6.0.0.20, vlan4022_l3 onlink, weight 1, 00:05:43
C>* 23.17.0.0/16 is directly connected, pf0vf4_if.3, 00:10:42
B>* 23.19.0.0/16 [20/0] via 6.0.0.18, vlan4022_l3 onlink, weight 1, 00:05:43
B>* 23.21.0.0/16 [20/0] via 6.0.0.20, vlan4022_l3 onlink, weight 1, 00:05:43
B>* 101.12.4.0/24 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.13.4.0/24 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.14.4.0/24 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.15.4.0/24 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44

VRF tenant4:
K>* 0.0.0.0/0 [255/8192] unreachable (ICMP unreachable), 00:10:42
B>* 6.0.0.4/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.12/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.13/32 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.14/32 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.15/32 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 7.4.0.6/32 [20/0] via 6.0.0.6, vlan4017_l3 onlink, weight 1, 00:05:43
C>* 7.4.0.16/32 is directly connected, tenant4, 00:10:42
B>* 7.4.0.18/32 [20/0] via 6.0.0.18, vlan4017_l3 onlink, weight 1, 00:05:43
B>* 7.4.0.20/32 [20/0] via 6.0.0.20, vlan4017_l3 onlink, weight 1, 00:05:43
C>* 24.17.0.0/16 is directly connected, pf0vf4_if.4, 00:10:42
B>* 24.19.0.0/16 [20/0] via 6.0.0.18, vlan4017_l3 onlink, weight 1, 00:05:43
B>* 24.21.0.0/16 [20/0] via 6.0.0.20, vlan4017_l3 onlink, weight 1, 00:05:43
B>* 101.12.4.0/24 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.13.4.0/24 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.14.4.0/24 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.15.4.0/24 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44

VRF tenant5:
K>* 0.0.0.0/0 [255/8192] unreachable (ICMP unreachable), 00:10:42
B>* 6.0.0.4/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.12/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.13/32 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.14/32 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.15/32 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 7.5.0.6/32 [20/0] via 6.0.0.6, vlan4046_l3 onlink, weight 1, 00:05:43
C>* 7.5.0.16/32 is directly connected, tenant5, 00:10:42
B>* 7.5.0.18/32 [20/0] via 6.0.0.18, vlan4046_l3 onlink, weight 1, 00:05:43
B>* 7.5.0.20/32 [20/0] via 6.0.0.20, vlan4046_l3 onlink, weight 1, 00:05:43
C>* 25.17.0.0/16 is directly connected, pf0vf4_if.5, 00:10:42
B>* 25.19.0.0/16 [20/0] via 6.0.0.18, vlan4046_l3 onlink, weight 1, 00:05:43
B>* 25.21.0.0/16 [20/0] via 6.0.0.20, vlan4046_l3 onlink, weight 1, 00:05:43
B>* 101.12.4.0/24 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.13.4.0/24 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.14.4.0/24 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.15.4.0/24 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44

VRF tenant6:
K>* 0.0.0.0/0 [255/8192] unreachable (ICMP unreachable), 00:10:42
B>* 6.0.0.4/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
  *                   via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.12/32 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.13/32 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 6.6.0.14/32 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
```

```
B>* 6.6.0.15/32 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 7.6.0.6/32 [20/0] via 6.0.0.6, vlan4041_l3 onlink, weight 1, 00:05:43
C>* 7.6.0.16/32 is directly connected, tenant6, 00:10:42
B>* 7.6.0.18/32 [20/0] via 6.0.0.18, vlan4041_l3 onlink, weight 1, 00:05:43
B>* 7.6.0.20/32 [20/0] via 6.0.0.20, vlan4041_l3 onlink, weight 1, 00:05:43
C>* 26.17.0.0/16 is directly connected, pf0vf4_if.6, 00:10:42
B>* 26.19.0.0/16 [20/0] via 6.0.0.18, vlan4041_l3 onlink, weight 1, 00:05:43
B>* 26.21.0.0/16 [20/0] via 6.0.0.20, vlan4041_l3 onlink, weight 1, 00:05:43
B>* 101.12.4.0/24 [20/0] via 6.0.0.12, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.13.4.0/24 [20/0] via 6.0.0.13, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.14.4.0/24 [20/0] via 6.0.0.14, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
B>* 101.15.4.0/24 [20/0] via 6.0.0.15, vxlan48 (vrf default) onlink, label 60000, weight 1, 00:05:44
doca-hbn-service-bf3-s06-1-ipmi#
```

DVNI Debugging

BGP/Zebra debug:

```
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [GKC5Y-XBAX9] vrf tenant1: import evpn prefix [5]:
[0]:[32]:[6.0.0.4] parent 0xaaaafda63a90 flags 0x410
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [KZNVF-SX7KT] ... new pi dest 0xaaaafe524650 (l 2)
pi 0xaaaafe5ae400 (l 1, f 0x4010)
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [GKC5Y-XBAX9] vrf tenant2: import evpn prefix [5]:
[0]:[32]:[6.0.0.4] parent 0xaaaafda63a90 flags 0x410
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [KZNVF-SX7KT] ... new pi dest 0xaaaafe51c420 (l 2)
pi 0xaaaafe55d230 (l 1, f 0x4010)
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [GKC5Y-XBAX9] vrf tenant3: import evpn prefix [5]:
[0]:[32]:[6.0.0.4] parent 0xaaaafda63a90 flags 0x410
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [KZNVF-SX7KT] ... new pi dest 0xaaaafe51a670 (l 2)
pi 0xaaaafe674820 (l 1, f 0x4010)
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [GKC5Y-XBAX9] vrf tenant4: import evpn prefix [5]:
[0]:[32]:[6.0.0.4] parent 0xaaaafda63a90 flags 0x410
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [KZNVF-SX7KT] ... new pi dest 0xaaaafe519fb0 (l 2)
pi 0xaaaafe675e40 (l 1, f 0x4010)
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [GKC5Y-XBAX9] vrf tenant5: import evpn prefix [5]:
[0]:[32]:[6.0.0.4] parent 0xaaaafda63a90 flags 0x410
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [KZNVF-SX7KT] ... new pi dest 0xaaaafe55ae50 (l 2)
pi 0xaaaafe5482f0 (l 1, f 0x4010)
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [GKC5Y-XBAX9] vrf tenant6: import evpn prefix [5]:
[0]:[32]:[6.0.0.4] parent 0xaaaafda63a90 flags 0x410
May  7 20:59:49 doca-hbn-service-bf3-s06-1-ipmi bgpd[1775018]: [KZNVF-SX7KT] ... new pi dest 0xaaaafdaf3590 (l 2)
pi 0xaaaafe48fbf0 (l 1, f 0x4010)
```

DVNI table:

```
root@doca-hbn-service-bf3-s06-1-ipmi:/tmp# cat /cumulus/nl2docad/run/software-tables/15
{
  "table": {
    "id": 15,
    "name": "HAL Downstream-VNI Table ",
    "count": 1,
    "records": [
    {
      "vni": 60000,
      "fid": 4098,
      "mark-for-del": 0,
      "vtep-users":
      {
        "count": 4,
        "vtep-user-list": [
          {
            "dest-vtep": "6.0.0.12",
            "dest-mac": "44:38:39:f0:00:12",
            "is-dmac-null": 0,
            "ref-cnt": 36
          },
          {
            "dest-vtep": "6.0.0.14",
            "dest-mac": "44:38:39:f0:00:14",
            "is-dmac-null": 0,
            "ref-cnt": 36
          },
          {
            "dest-vtep": "6.0.0.13",
            "dest-mac": "44:38:39:f0:00:13",
            "is-dmac-null": 0,
            "ref-cnt": 36
          },
          {
            "dest-vtep": "6.0.0.15",
            "dest-mac": "44:38:39:f0:00:15",
            "is-dmac-null": 0,
            "ref-cnt": 36
          }
        ]
      }
    }
   ]
  }
}root@doca-hbn-service-bf3-s06-1-ipmi:/tmp#
```

Sample DVNI Configuration

HBN configuration example for BlueField devices:

```
root@doca-hbn-service-bf3-s06-1-ipmi:/tmp# nv config show -o commands
nv set bridge domain br_default vlan 101 vni 10101
nv set bridge domain br_default vlan 102 vni 10102
nv set bridge domain br_default vlan 201 vni 10201
nv set bridge domain br_default vlan 202 vni 10202
nv set evpn enable on
nv set evpn route-advertise svi-ip off
nv set interface ilan3200 ip vrf internet1
nv set interface ilan3200 vlan 3200
nv set interface ilan3200,slan3201,vlan101-102,201-202,3001-3006 base-interface br_default
nv set interface ilan3200,slan3201,vlan101-102,201-202,3001-3006 type svi
nv set interface lo ip address 6.0.0.16/32
nv set interface lo ip address 2001::16/128
nv set interface lo type loopback
nv set interface p0_if,p1_if,pf0hpf_if,pf0vf0_if,pf0vf1_if,pf0vf2_if,pf0vf3_if,pf0vf4_if,pf1hpf_if type swp
nv set interface pf0vf0_if bridge domain br_default access 101
nv set interface pf0vf1_if bridge domain br_default access 102
nv set interface pf0vf2_if bridge domain br_default access 201
nv set interface pf0vf3_if bridge domain br_default access 202
nv set interface pf0vf4_if.3 ip address 23.17.0.16/16
nv set interface pf0vf4_if.3 ip address 2020:0:3:17::16/64
nv set interface pf0vf4_if.3 vlan 3
nv set interface pf0vf4_if.3,vlan3003 ip vrf tenant3
nv set interface pf0vf4_if.3-6 base-interface pf0vf4_if
nv set interface pf0vf4_if.3-6 type sub
nv set interface pf0vf4_if.4 ip address 24.17.0.16/16
nv set interface pf0vf4_if.4 ip address 2020:0:4:17::16/64
nv set interface pf0vf4_if.4 vlan 4
nv set interface pf0vf4_if.4,vlan3004 ip vrf tenant4
nv set interface pf0vf4_if.5 ip address 25.17.0.16/16
nv set interface pf0vf4_if.5 ip address 2020:0:5:17::16/64
nv set interface pf0vf4_if.5 vlan 5
nv set interface pf0vf4_if.5,vlan3005 ip vrf tenant5
nv set interface pf0vf4_if.6 ip address 26.17.0.16/16
nv set interface pf0vf4_if.6 ip address 2020:0:6:17::16/64
nv set interface pf0vf4_if.6 vlan 6
nv set interface pf0vf4_if.6,vlan3006 ip vrf tenant6
nv set interface slan3201 ip vrf special1
nv set interface slan3201 vlan 3201
nv set interface vlan101 ip address 21.1.0.16/16
nv set interface vlan101 ip address 2020:0:1:1::16/64
nv set interface vlan101 ip vrr address 21.1.0.250/16
nv set interface vlan101 ip vrr address 2020:0:1:1::250/64
nv set interface vlan101 ip vrr mac-address 00:00:01:00:00:65
nv set interface vlan101 vlan 101
nv set interface vlan101-102,201-202 ip vrr enable on
nv set interface vlan101-102,3001 ip vrf tenant1
nv set interface vlan102 ip address 21.2.0.16/16
nv set interface vlan102 ip address 2020:0:1:2::16/64
nv set interface vlan102 ip vrr address 21.2.0.250/16
nv set interface vlan102 ip vrr address 2020:0:1:2::250/64
nv set interface vlan102 ip vrr mac-address 00:00:01:00:00:66
nv set interface vlan102 vlan 102
nv set interface vlan201 ip address 22.1.0.16/16
nv set interface vlan201 ip address 2020:0:2:1::16/64
nv set interface vlan201 ip vrr address 22.1.0.250/16
nv set interface vlan201 ip vrr address 2020:0:2:1::250/64
nv set interface vlan201 ip vrr mac-address 00:00:02:00:00:c9
nv set interface vlan201 vlan 201
nv set interface vlan201-202,3002 ip vrf tenant2
nv set interface vlan202 ip address 22.2.0.16/16
nv set interface vlan202 ip address 2020:0:2:2::16/64
nv set interface vlan202 ip vrr address 22.2.0.250/16
nv set interface vlan202 ip vrr address 2020:0:2:2::250/64
nv set interface vlan202 ip vrr mac-address 00:00:02:00:00:ca
nv set interface vlan202 vlan 202
nv set interface vlan3001 vlan 3001
nv set interface vlan3002 vlan 3002
nv set interface vlan3003 vlan 3003
nv set interface vlan3004 vlan 3004
nv set interface vlan3005 vlan 3005
nv set interface vlan3006 vlan 3006
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan source address 6.0.0.16
nv set platform
nv set router bgp autonomous-system 65011
nv set router bgp enable on
nv set router bgp router-id 6.0.0.16
nv set router vrr enable on
nv set system config snippet
nv set system global
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor 6.0.0.7 peer-group rservers
nv set vrf default router bgp neighbor 6.0.0.7 type numbered
nv set vrf default router bgp neighbor 6.0.0.8 peer-group rservers
nv set vrf default router bgp neighbor 6.0.0.8 type numbered
nv set vrf default router bgp neighbor 6.0.0.9 peer-group rservers
nv set vrf default router bgp neighbor 6.0.0.9 type numbered
nv set vrf default router bgp neighbor p0_if peer-group fabric
nv set vrf default router bgp neighbor p0_if type unnumbered
nv set vrf default router bgp neighbor p1_if peer-group fabric
nv set vrf default router bgp neighbor p1_if type unnumbered
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group fabric bfd detect-multiplier 3
```

```
nv set vrf default router bgp peer-group fabric bfd enable on
nv set vrf default router bgp peer-group fabric bfd min-rx-interval 1000
nv set vrf default router bgp peer-group fabric bfd min-tx-interval 1000
nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp peer-group rservers address-family ipv4-unicast enable off
nv set vrf default router bgp peer-group rservers address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group rservers address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group rservers multihop-ttl 10
nv set vrf default router bgp peer-group rservers remote-as external
nv set vrf default router bgp peer-group rservers update-source lo
nv set vrf internet1 evpn enable on
nv set vrf internet1 evpn vni 42000
nv set vrf internet1 loopback ip address 8.1.0.16/32
nv set vrf internet1 loopback ip address 2008:0:1::16/64
nv set vrf internet1 router bgp address-family ipv4-unicast enable on
nv set vrf internet1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf internet1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf internet1 router bgp address-family ipv6-unicast enable on
nv set vrf internet1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf internet1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf internet1 router bgp enable on
nv set vrf special1 evpn enable on
nv set vrf special1 evpn vni 42001
nv set vrf special1 loopback ip address 9.1.0.16/32
nv set vrf special1 loopback ip address 2009:0:1::16/64
nv set vrf special1 router bgp address-family ipv4-unicast enable on
nv set vrf special1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf special1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf special1 router bgp address-family ipv6-unicast enable on
nv set vrf special1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf special1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf special1 router bgp enable on
nv set vrf tenant1 evpn enable on
nv set vrf tenant1 evpn vni 30001
nv set vrf tenant1 loopback ip address 7.1.0.16/32
nv set vrf tenant1 loopback ip address 2007:0:1::16/64
nv set vrf tenant1 router bgp address-family ipv4-unicast enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf tenant1 router bgp enable on
nv set vrf tenant1 router bgp neighbor 21.1.0.17 peer-group hostgroup
nv set vrf tenant1 router bgp neighbor 21.1.0.17 type numbered
nv set vrf tenant1 router bgp peer-group hostgroup address-family ipv4-unicast enable on
nv set vrf tenant1 router bgp peer-group hostgroup address-family ipv6-unicast enable on
nv set vrf tenant1 router bgp peer-group hostgroup remote-as external
nv set vrf tenant1 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant1 router bgp route-import from-evpn route-target auto
nv set vrf tenant1 router bgp router-id 6.0.0.16
nv set vrf tenant2 evpn enable on
nv set vrf tenant2 evpn vni 30002
nv set vrf tenant2 loopback ip address 7.2.0.16/32
nv set vrf tenant2 loopback ip address 2007:0:2::16/64
nv set vrf tenant2 router bgp address-family ipv4-unicast enable on
nv set vrf tenant2 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant2 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant2 router bgp address-family ipv6-unicast enable on
nv set vrf tenant2 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf tenant2 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf tenant2 router bgp enable on
nv set vrf tenant2 router bgp neighbor 22.1.0.17 peer-group hostgroup
nv set vrf tenant2 router bgp neighbor 22.1.0.17 type numbered
nv set vrf tenant2 router bgp peer-group hostgroup address-family ipv4-unicast enable on
nv set vrf tenant2 router bgp peer-group hostgroup address-family ipv6-unicast enable on
nv set vrf tenant2 router bgp peer-group hostgroup remote-as external
nv set vrf tenant2 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant2 router bgp route-import from-evpn route-target auto
nv set vrf tenant2 router bgp router-id 6.0.0.16
nv set vrf tenant3 evpn enable on
nv set vrf tenant3 evpn vni 30003
nv set vrf tenant3 loopback ip address 7.3.0.16/32
nv set vrf tenant3 loopback ip address 2007:0:3::16/64
nv set vrf tenant3 router bgp address-family ipv4-unicast enable on
nv set vrf tenant3 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant3 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant3 router bgp address-family ipv6-unicast enable on
nv set vrf tenant3 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf tenant3 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf tenant3 router bgp enable on
nv set vrf tenant3 router bgp neighbor 23.17.0.17 peer-group hostgroup
nv set vrf tenant3 router bgp neighbor 23.17.0.17 type numbered
nv set vrf tenant3 router bgp peer-group hostgroup address-family ipv4-unicast enable on
nv set vrf tenant3 router bgp peer-group hostgroup address-family ipv6-unicast enable on
nv set vrf tenant3 router bgp peer-group hostgroup remote-as external
nv set vrf tenant3 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant3 router bgp route-import from-evpn route-target auto
nv set vrf tenant3 router bgp router-id 6.0.0.16
nv set vrf tenant3 table auto
nv set vrf tenant4 evpn enable on
nv set vrf tenant4 evpn vni 30004
nv set vrf tenant4 loopback ip address 7.4.0.16/32
nv set vrf tenant4 loopback ip address 2007:0:4::16/64
nv set vrf tenant4 router bgp address-family ipv4-unicast enable on
nv set vrf tenant4 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant4 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant4 router bgp address-family ipv6-unicast enable on
nv set vrf tenant4 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf tenant4 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf tenant4 router bgp enable on
nv set vrf tenant4 router bgp neighbor 24.17.0.17 peer-group hostgroup
nv set vrf tenant4 router bgp neighbor 24.17.0.17 type numbered
nv set vrf tenant4 router bgp peer-group hostgroup address-family ipv4-unicast enable on
nv set vrf tenant4 router bgp peer-group hostgroup address-family ipv6-unicast enable on
```

```
nv set vrf tenant4 router bgp peer-group hostgroup remote-as external
nv set vrf tenant4 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant4 router bgp route-import from-evpn route-target auto
nv set vrf tenant4 router bgp router-id 6.0.0.16
nv set vrf tenant4 table auto
nv set vrf tenant5 evpn enable on
nv set vrf tenant5 evpn vni 30005
nv set vrf tenant5 loopback ip address 7.5.0.16/32
nv set vrf tenant5 loopback ip address 2007:0:5::16/64
nv set vrf tenant5 router bgp address-family ipv4-unicast enable on
nv set vrf tenant5 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant5 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant5 router bgp address-family ipv6-unicast enable on
nv set vrf tenant5 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf tenant5 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf tenant5 router bgp enable on
nv set vrf tenant5 router bgp neighbor 25.17.0.17 peer-group hostgroup
nv set vrf tenant5 router bgp neighbor 25.17.0.17 type numbered
nv set vrf tenant5 router bgp peer-group hostgroup address-family ipv4-unicast enable on
nv set vrf tenant5 router bgp peer-group hostgroup address-family ipv6-unicast enable on
nv set vrf tenant5 router bgp peer-group hostgroup remote-as external
nv set vrf tenant5 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant5 router bgp route-import from-evpn route-target auto
nv set vrf tenant5 router bgp router-id 6.0.0.16
nv set vrf tenant5 table auto
nv set vrf tenant6 evpn enable on
nv set vrf tenant6 evpn vni 30006
nv set vrf tenant6 loopback ip address 7.6.0.16/32
nv set vrf tenant6 loopback ip address 2007:0:6::16/64
nv set vrf tenant6 router bgp address-family ipv4-unicast enable on
nv set vrf tenant6 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant6 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant6 router bgp address-family ipv6-unicast enable on
nv set vrf tenant6 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf tenant6 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf tenant6 router bgp enable on
nv set vrf tenant6 router bgp neighbor 26.17.0.17 peer-group hostgroup
nv set vrf tenant6 router bgp neighbor 26.17.0.17 type numbered
nv set vrf tenant6 router bgp peer-group hostgroup address-family ipv4-unicast enable on
nv set vrf tenant6 router bgp peer-group hostgroup address-family ipv6-unicast enable on
nv set vrf tenant6 router bgp peer-group hostgroup remote-as external
nv set vrf tenant6 router bgp route-import from-evpn route-target ANY:60000
nv set vrf tenant6 router bgp route-import from-evpn route-target auto
nv set vrf tenant6 router bgp router-id 6.0.0.16
nv set vrf tenant6 table auto
root@doca-hbn-service-bf3-s06-1-ipmi:/tmp#
```

SS1 switch configuration example:

```
root@superspine1:mgmt:/home/cumulus# nv config show -o commands
nv set bridge domain br_default vlan 101 vni 10101
nv set bridge domain br_default vlan 102 vni 10102
nv set bridge domain br_default vlan 201 vni 10201
nv set bridge domain br_default vlan 202 vni 10202
nv set evpn enable on
nv set interface eth0 ip address 192.168.0.15/24
nv set interface eth0 ip gateway 192.168.0.2
nv set interface eth0 type eth
nv set interface lo ip address 6.0.0.12/32
nv set interface lo ip address 2001::12/128
nv set interface lo type loopback
nv set interface swp1-6 type swp
nv set interface swp6 ip address 101.12.4.12/24
nv set interface swp6 ip address 2101:12::4:12/112
nv set interface swp6 ip vrf external
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan source address 6.0.0.12
nv set platform
nv set router bgp autonomous-system 65300
nv set router bgp enable on
nv set router bgp router-id 6.0.0.12
nv set system config snippet
nv set system global system-mac 44:38:39:f0:00:12
nv set system hostname superspine1
nv set system ssh-server permit-root-login enabled
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor swp1 peer-group fabric
nv set vrf default router bgp neighbor swp1 type unnumbered
nv set vrf default router bgp neighbor swp2 peer-group fabric
nv set vrf default router bgp neighbor swp2 type unnumbered
nv set vrf default router bgp neighbor swp3 peer-group rservers
nv set vrf default router bgp neighbor swp3 type unnumbered
nv set vrf default router bgp neighbor swp4 peer-group rservers
nv set vrf default router bgp neighbor swp4 type unnumbered
nv set vrf default router bgp neighbor swp5 peer-group rservers
nv set vrf default router bgp neighbor swp5 type unnumbered
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group fabric bfd detect-multiplier 3
nv set vrf default router bgp peer-group fabric bfd enable on
nv set vrf default router bgp peer-group fabric bfd min-rx-interval 1000
nv set vrf default router bgp peer-group fabric bfd min-tx-interval 1000
nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp peer-group rservers address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group rservers address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group rservers address-family l2vpn-evpn enable on
```

```
nv set vrf default router bgp peer-group rservers remote-as external
nv set vrf external evpn enable on
nv set vrf external evpn vni 60000
nv set vrf external loopback ip address 6.6.0.12/32
nv set vrf external loopback ip address 2006:0:6::12/64
nv set vrf external router bgp address-family ipv4-unicast enable on
nv set vrf external router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf external router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf external router bgp address-family ipv6-unicast enable on
nv set vrf external router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf external router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf external router bgp address-family l2vpn-evpn enable on
nv set vrf external router bgp enable on
nv set vrf external router bgp neighbor swp6 peer-group peer-group-fw
nv set vrf external router bgp neighbor swp6 type unnumbered
nv set vrf external router bgp peer-group peer-group-fw address-family ipv4-unicast enable on
nv set vrf external router bgp peer-group peer-group-fw address-family ipv6-unicast enable on
nv set vrf external router bgp peer-group peer-group-fw remote-as external
nv set vrf external router bgp route-import from-evpn route-target ANY:30001
nv set vrf external router bgp route-import from-evpn route-target ANY:30002
nv set vrf external router bgp route-import from-evpn route-target ANY:30003
nv set vrf external router bgp route-import from-evpn route-target ANY:30004
nv set vrf external router bgp route-import from-evpn route-target ANY:30005
nv set vrf external router bgp route-import from-evpn route-target ANY:30006
nv set vrf external router bgp route-import from-evpn route-target auto
root@superspine1:mgmt:/home/cumulus#
```

8.9.5.3.3.7 Gateway Application Using Downstream VNI and Subinterface

A DPU running the HBN service can be deployed in the role of a border gateway using a combination of HBN features, specifically, EVPN symmetric routing, downstream VNI, VRF route-leaking, and VLAN sub-interfaces. Such a border gateway can do the northbound traffic handoff (to external networks or the Internet) for one or more tenants. In this gateway configuration, the BlueField's uplinks must carry both the tenant traffic which would be in the "overlay" and VXLAN-encapsulated, as well as traffic to and from the external network or Internet, which would be direct-routed in the "underlay". This is accomplished by configuring and running VXLAN-EVPN on the uplink interfaces while configuring and using additional VLAN sub-interfaces on those same uplinks for the traffic to and from external networks. These VLAN sub-interfaces would be configured into an Internet or external VRF for separation from the VXLAN-encapsulated traffic which is carried over the default VRF.

With a BlueField running HBN able to act as a border gateway, there is no longer a dependence on physical switches and routers to terminate VXLAN traffic and perform this role, hence the requirements on the underlying network is simply to provide end-to-end IP/UDP connectivity and facilitate the setup of overlay networks on top. Additionally, multiple border gateways can be easily deployed in the network, including dedicated gateways per tenant or shared gateways for groups of tenants.

> ⚠ Since HBN currently does not support network address translation (NAT), a dedicated border gateway must be deployed per tenant, for those tenants that have overlapping IP addresses.

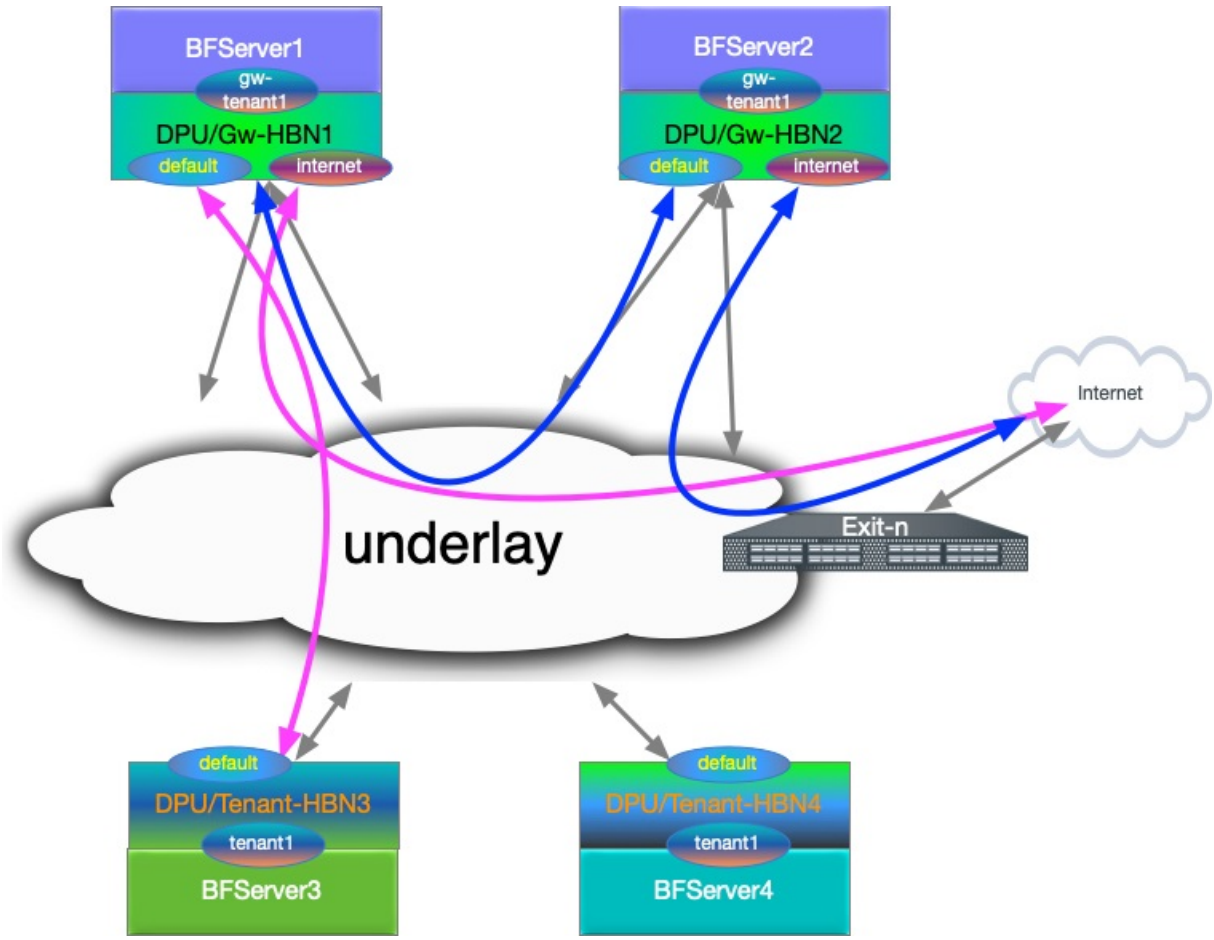
For more details and configuration of some of the key features that together enable the border gateway functionality, refer to sections on Downstream VNIs and HBN Service Configuration#VLAN Subinterfaces.

Gateway Application Example

The following topology diagram and associated configuration snippets show two different use cases of border gateway deployment:

- `tenant1` is an example of a tenant hosted on a server(s) with a non-gateway BlueField, using a dedicated border gateway on BlueField Gw-HBN1 for Internet connectivity. Traffic flow to and from the Internet for this tenant is marked in pink.

- **gw_tenant1** is an example of a tenant hosted on a server(s) with a gateway BlueField. In this case, the border gateway for this tenant is provided by BlueField Gw-HBN2. Traffic flow to and from the Internet for this tenant is depicted in blue.



L3 VNI Origin Map

HBN	VRF	L3 VNI
gw-hbn1 and **gw-hbn2**	**internet1**	**10000**
gw-hbn1 and **gw-hbn2**	**gw_tenant1**	**30000**
tenant-hbn3 and **tenant-hbn4**	**tenant1**	**20000**

Configuration Snippet for Internet VRF

- Internet VRF is established in BGP sessions using sub-interface features with underlay switches (i.e., **p0_if.60** and **p1_if.60**)
- The Internet VRF also imports all the tenant VRFs (local and remote) using the downstream VNI feature with from-EVPN syntax

```
nv set interface p0_if.60,p1_if.60,vlan10 ip vrf internet1
nv set vrf internet1 evpn enable on
```

```
nv set vrf internet1 evpn vni 10000
    nv set vrf internet1 loopback ip address 6.2.0.1/32
nv set vrf internet1 loopback ip address 2001:cafe:feed::1/128
nv set vrf internet1 router bgp address-family ipv4-unicast enable on
nv set vrf internet1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf internet1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf internet1 router bgp address-family ipv6-unicast enable on
nv set vrf internet1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf internet1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf internet1 router bgp address-family l2vpn-evpn enable on
nv set vrf internet1 router bgp autonomous-system 65552
nv set vrf internet1 router bgp enable on
nv set vrf internet1 router bgp neighbor p0_if.60 capabilities source-address internet1
nv set vrf internet1 router bgp neighbor p0_if.60 peer-group l3_pg1
nv set vrf internet1 router bgp neighbor p0_if.60 type unnumbered
nv set vrf internet1 router bgp neighbor p1_if.60 capabilities source-address internet1
nv set vrf internet1 router bgp neighbor p1_if.60 peer-group l3_pg1
nv set vrf internet1 router bgp neighbor p1_if.60 type unnumbered
nv set vrf internet1 router bgp peer-group l3_pg1 address-family ipv4-unicast enable on
nv set vrf internet1 router bgp peer-group l3_pg1 address-family ipv6-unicast enable on
nv set vrf internet1 router bgp peer-group l3_pg1 remote-as external
nv set vrf internet1 router bgp route-export to-evpn route-target 65552:10000
nv set vrf internet1 router bgp route-import from-evpn route-target ANY:20000
nv set vrf internet1 router bgp route-import from-evpn route-target ANY:30000
nv set vrf internet1 router bgp route-import from-evpn route-target auto
nv set vrf internet1 router bgp router-id 27.0.0.5
```

Configuration Snippet for Gateway Local Tenant

- **gw_tenant** is stretched across 2 gateway and connected using L3 VNI
- **gw_tenant** has multiple SVIs, which are represented as **vlan30** and **vlan31** SVIs
- Internet L3 VNI is imported using DVNI. The example also explicitly adds route targets using auto.

gw_tenant VRF:

```
nv set interface vlan30-31 ip vrf gw_tenant1
nv set vrf gw_tenant1 evpn enable on
nv set vrf gw_tenant1 evpn vni 30000
nv set vrf gw_tenant1 loopback ip address 15.3.0.1/32
nv set vrf gw_tenant1 loopback ip address 2001:bad:c0de::1/128
nv set vrf gw_tenant1 router bgp address-family ipv4-unicast enable on
nv set vrf gw_tenant1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf gw_tenant1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf gw_tenant1 router bgp address-family ipv6-unicast enable on
nv set vrf gw_tenant1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf gw_tenant1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf gw_tenant1 router bgp address-family l2vpn-evpn enable on
nv set vrf gw_tenant1 router bgp autonomous-system 65552
nv set vrf gw_tenant1 router bgp enable on
nv set vrf gw_tenant1 router bgp route-export to-evpn route-target 65552:30000
nv set vrf gw_tenant1 router bgp route-import from-evpn route-target ANY:10000
nv set vrf gw_tenant1 router bgp route-import from-evpn route-target auto
nv set vrf gw_tenant1 router bgp router-id 27.0.0.5
```

Configuration Snippet for Remote Tenant

- **tenant1** is stretched across 2 remote HBN VTEP and connected using L3 VNI
- **tenant1** is importing Internet L3 VNI routes in **tenant1** and adding its own using route-target auto

Tenant VRF:

```
nv set interface vlan20-21 ip vrf tenant1
nv set vrf tenant1 evpn enable on
nv set vrf tenant1 evpn vni 20000
nv set vrf tenant1 loopback ip address 15.1.0.1/32
nv set vrf tenant1 loopback ip address 2001:c001:c0de::1/128
nv set vrf tenant1 router bgp address-family ipv4-unicast enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf tenant1 router bgp address-family l2vpn-evpn enable on
nv set vrf tenant1 router bgp autonomous-system 6300656
nv set vrf tenant1 router bgp enable on
nv set vrf tenant1 router bgp route-export to-evpn route-target 6300656:20000
nv set vrf tenant1 router bgp route-import from-evpn route-target ANY:10000
nv set vrf tenant1 router bgp route-import from-evpn route-target auto
```

```
nv set vrf tenant1 router bgp router-id 27.0.0.17
```

HBN Accelerated Routing Plan

The following subsections pick a few IP endpoints from the code snippets above and examine their route distribution.

- The gateway devices have a remote tenant
- Internet route is injected using the default originator from the exit node

Gateway-1 Route Info

- BGP sharing the uplink via a sub-interface feature in the Internet VRF.
  gateway1 - External Routes Internet VRF

```
root@doca-hbn-service # ip -4 route show default vrf internet1
default nhid 248 proto bgp metric 20

root@doca-hbn-service-bf2-s12-1-ipmi:~# ip -6 route show default vrf internet1
default nhid 248 proto bgp metric 20 pref medium


root@doca-hbn-service # ip nexthop get id 248 ; ip nexthop get id 249 ; ip nexthop get id 250
id 248 group 249/250 proto zebra
id 249 via fe80::202:ff:fe00:1d dev p0_if.60 scope link proto zebra
id 250 via fe80::202:ff:fe00:26 dev p1_if.60 scope link proto zebra
```

- Local Tenant routing information: The Internet is reached using L3 VNI via a peer gateway.
  gateway1 - External Routes gw_tenant VRF

```
root@hbn:/# ip -4 route show vrf gw_tenant1  default
default  encap ip id 10000 src 0.0.0.0 dst 27.0.0.7 ttl 0 tos 0 via 27.0.0.7 dev vxlan48 proto bgp metric
20 onlink

root@hbn:/# ip -6 route show vrf gw_tenant1  default
default  encap ip id 10000 src 0.0.0.0 dst 27.0.0.7 ttl 0 tos 0 via ::ffff:27.0.0.7 dev vxlan48 proto bgp
metric 20 onlink pref medium
```

- Remote tenant routing reachability via **gateway1** using DVNI CFG.
- Considering an IP endpoint from the remote **tenant1** VRF on Tenant-HBN3.
  gateway1 - Routes Internet VRF

```
root@hbn:/# ip -4 route show vrf internet1  15.1.0.1/32
15.1.0.1  encap ip id 20000 src 0.0.0.0 dst 27.0.0.17 ttl 0 tos 0 via 27.0.0.17 dev vxlan48 proto bgp
metric 20 onlink

root@hbn:/# ip -6 route show vrf internet1  2001:c001:c0de::1/128
2001:c001:c0de::1  encap ip id 20000 src 0.0.0.0 dst 27.0.0.17 ttl 0 tos 0 via ::ffff:27.0.0.17 dev vxlan48
proto bgp metric 20 onlink pref medium
```

Tenant-HBN3 Route Info

- IP endpoint as **gateway1** VRF loopback and DVNI handoff for the VNI is reaching the **gateway1** node.
  tenant-hbn3 - Routes tenant VRF

```
root@hbn:/# ip -4 route show vrf tenant1 6.2.0.1/32
6.2.0.1  encap ip id 10000 src 0.0.0.0 dst 27.0.0.5 ttl 0 tos 0 via 27.0.0.5 dev vxlan48 proto bgp metric
20 onlink

root@hbn:/# ip -6 route show vrf tenant1 2001:cafe:feed::1/128
2001:cafe:feed::1  encap ip id 10000 src 0.0.0.0 dst 27.0.0.5 ttl 0 tos 0 via ::ffff:27.0.0.5 dev vxlan48
proto bgp metric 20 onlink pref medium
```

- Internet VRF default route is reaching the remote tenant VRF.
  tenant-hbn3 external - Routes tenant VRF

```
root@hbn:/# ip -4 route show vrf tenant1 default
default proto bgp metric 20
    nexthop  encap ip id 10000 src 0.0.0.0 dst 27.0.0.5 ttl 0 tos 0 via 27.0.0.5 dev vxlan48 weight 1
 onlink
    nexthop  encap ip id 10000 src 0.0.0.0 dst 27.0.0.7 ttl 0 tos 0 via 27.0.0.7 dev vxlan48 weight 1
 onlink

root@hbn:/# ip -6 route show vrf tenant1 default
default proto bgp metric 20 pref medium
    nexthop  encap ip id 10000 src 0.0.0.0 dst 27.0.0.5 ttl 0 tos 0 via ::ffff:27.0.0.5 dev vxlan48 weight
1 onlink
    nexthop  encap ip id 10000 src 0.0.0.0 dst 27.0.0.7 ttl 0 tos 0 via ::ffff:27.0.0.7 dev vxlan48 weight
1 onlink
```

Gateway and Tenant Complete Configuration Example

Gateway-1 Full Configuration

Gateway-HBN-1

```
nv set bridge domain br_default encap 802.1Q
nv set bridge domain br_default type vlan-aware
nv set bridge domain br_default untagged 1
nv set bridge domain br_default vlan 10,30-31
nv set evpn enable on
nv set interface lo ip address 27.0.0.5/32
nv set interface lo ip address 2001:c001:ff:f00d::5/128
nv set interface lo type loopback
nv set interface p0_if,p1_if,pf0hpf_if,pf0vf0_if,pf0vf1_if,pf0vf2_if,pf0vf3_if,pf0vf4_if,pf1hpf_if type swp
nv set interface p0_if.60 base-interface p0_if
nv set interface p0_if.60,p1_if.60 type sub
nv set interface p0_if.60,p1_if.60 vlan 60
nv set interface p0_if.60,p1_if.60,vlan10 ip vrf internet1
nv set interface p1_if.60 base-interface p1_if
nv set interface pf0hpf_if bridge domain br_default access 30
nv set interface pf0vf0_if bridge domain br_default access 31
nv set interface vlan10 ip address 12.2.0.1/24
nv set interface vlan10 ip address 2001:c001:d00d::1/96
nv set interface vlan10 vlan 10
nv set interface vlan10,30-31 ip ipv4 forward on
nv set interface vlan10,30-31 ip ipv6 forward on
nv set interface vlan10,30-31 type svi
nv set interface vlan30 ip address 45.3.0.1/24
nv set interface vlan30 ip address 2001:b055:b00c::1/96
nv set interface vlan30 vlan 30
nv set interface vlan30-31 ip vrf gw_tenant1
nv set interface vlan31 ip address 45.3.1.1/24
nv set interface vlan31 ip address 2001:b055:b00c::1:0:1/96
nv set interface vlan31 vlan 31
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan mac-learning off
nv set nve vxlan source address 27.0.0.5
nv set platform
nv set router bgp enable on
nv set system config snippet
nv set system global anycast-mac 44:38:39:42:42:17
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp autonomous-system 65552
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor 27.0.0.11 peer-group rs_client
nv set vrf default router bgp neighbor 27.0.0.11 type numbered
nv set vrf default router bgp neighbor 27.0.0.12 peer-group rs_client
nv set vrf default router bgp neighbor 27.0.0.12 type numbered
nv set vrf default router bgp neighbor p0_if capabilities source-address lo
nv set vrf default router bgp neighbor p0_if peer-group fabric
nv set vrf default router bgp neighbor p0_if type unnumbered
nv set vrf default router bgp neighbor p1_if capabilities source-address lo
nv set vrf default router bgp neighbor p1_if peer-group fabric
nv set vrf default router bgp neighbor p1_if type unnumbered
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn enable off
nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp peer-group fabric timers connection-retry 5
nv set vrf default router bgp peer-group fabric timers hold 30
nv set vrf default router bgp peer-group fabric timers keepalive 10
nv set vrf default router bgp peer-group rs_client address-family ipv4-unicast enable off
nv set vrf default router bgp peer-group rs_client address-family ipv6-unicast enable off
nv set vrf default router bgp peer-group rs_client address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group rs_client address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group rs_client multihop-ttl 5
nv set vrf default router bgp peer-group rs_client remote-as external
nv set vrf default router bgp peer-group rs_client timers connection-retry 5
nv set vrf default router bgp peer-group rs_client timers hold 30
nv set vrf default router bgp peer-group rs_client timers keepalive 10
nv set vrf default router bgp router-id 27.0.0.5
nv set vrf gw_tenant1 evpn enable on
nv set vrf gw_tenant1 evpn vni 30000
```

```
nv set vrf gw_tenant1 loopback ip address 15.3.0.1/32
nv set vrf gw_tenant1 loopback ip address 2001:bad:c0de::1/128
nv set vrf gw_tenant1 router bgp address-family ipv4-unicast enable on
nv set vrf gw_tenant1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf gw_tenant1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf gw_tenant1 router bgp address-family ipv6-unicast enable on
nv set vrf gw_tenant1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf gw_tenant1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf gw_tenant1 router bgp address-family l2vpn-evpn enable on
nv set vrf gw_tenant1 router bgp autonomous-system 65552
nv set vrf gw_tenant1 router bgp enable on
nv set vrf gw_tenant1 router bgp route-export to-evpn route-target 65552:30000
nv set vrf gw_tenant1 router bgp route-import from-evpn route-target ANY:10000
nv set vrf gw_tenant1 router bgp route-import from-evpn route-target auto
nv set vrf gw_tenant1 router bgp router-id 27.0.0.5
nv set vrf internet1 evpn enable on
nv set vrf internet1 evpn vni 10000
nv set vrf internet1 loopback ip address 6.2.0.1/32
nv set vrf internet1 loopback ip address 2001:cafe:feed::1/128
nv set vrf internet1 router bgp address-family ipv4-unicast enable on
nv set vrf internet1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf internet1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf internet1 router bgp address-family ipv6-unicast enable on
nv set vrf internet1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf internet1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf internet1 router bgp address-family l2vpn-evpn enable on
nv set vrf internet1 router bgp autonomous-system 65552
nv set vrf internet1 router bgp enable on
nv set vrf internet1 router bgp neighbor p0_if.60 capabilities source-address internet1
nv set vrf internet1 router bgp neighbor p0_if.60 peer-group l3_pg1
nv set vrf internet1 router bgp neighbor p0_if.60 type unnumbered
nv set vrf internet1 router bgp neighbor p1_if.60 capabilities source-address internet1
nv set vrf internet1 router bgp neighbor p1_if.60 peer-group l3_pg1
nv set vrf internet1 router bgp neighbor p1_if.60 type unnumbered
nv set vrf internet1 router bgp peer-group l3_pg1 address-family ipv4-unicast enable on
nv set vrf internet1 router bgp peer-group l3_pg1 address-family ipv6-unicast enable on
nv set vrf internet1 router bgp peer-group l3_pg1 remote-as external
nv set vrf internet1 router bgp route-export to-evpn route-target 65552:10000
nv set vrf internet1 router bgp route-import from-evpn route-target ANY:20000
nv set vrf internet1 router bgp route-import from-evpn route-target ANY:30000
nv set vrf internet1 router bgp route-import from-evpn route-target auto
nv set vrf internet1 router bgp router-id 27.0.0.5
```

Gateway-2 Full Configuration

Gateway-HBN-2

```
nv set bridge domain br_default encap 802.1Q
nv set bridge domain br_default type vlan-aware
nv set bridge domain br_default untagged 1
nv set bridge domain br_default vlan 10,30-31
nv set evpn enable on
nv set interface lo ip address 27.0.0.7/32
nv set interface lo ip address 2001:c001:ff:f00d::7/128
nv set interface lo type loopback
nv set interface p0_if,p1_if,pf0hpf_if,pf0vf0_if,pf0vf1_if,pf0vf2_if,pf0vf3_if,pf0vf4_if,pf1hpf_if type swp
nv set interface p0_if.60 base-interface p0_if
nv set interface p0_if.60,p1_if.60 type sub
nv set interface p0_if.60,p1_if.60 vlan 60
nv set interface p0_if.60,p1_if.60,vlan10 ip vrf internet1
nv set interface p1_if.60 base-interface p1_if
nv set interface pf0hpf_if bridge domain br_default access 30
nv set interface pf0vf0_if bridge domain br_default access 31
nv set interface vlan10 ip address 12.2.1.1/24
nv set interface vlan10 ip address 2001:c001:d00d::1:0:1/96
nv set interface vlan10 vlan 10
nv set interface vlan10,30-31 ip ipv4 forward on
nv set interface vlan10,30-31 ip ipv6 forward on
nv set interface vlan10,30-31 type svi
nv set interface vlan30 ip address 45.3.2.1/24
nv set interface vlan30 ip address 2001:b055:b00c::2:0:1/96
nv set interface vlan30 vlan 30
nv set interface vlan30-31 ip vrf gw_tenant1
nv set interface vlan31 ip address 45.3.3.1/24
nv set interface vlan31 ip address 2001:b055:b00c::3:0:1/96
nv set interface vlan31 vlan 31
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan mac-learning off
nv set nve vxlan source address 27.0.0.7
nv set platform
nv set router bgp enable on
nv set system config snippet
nv set system global anycast-mac 44:38:39:42:42:19
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp autonomous-system 65554
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor 27.0.0.11 peer-group rs_client
nv set vrf default router bgp neighbor 27.0.0.11 type numbered
nv set vrf default router bgp neighbor 27.0.0.12 peer-group rs_client
nv set vrf default router bgp neighbor 27.0.0.12 type numbered
nv set vrf default router bgp neighbor p0_if capabilities source-address lo
nv set vrf default router bgp neighbor p0_if peer-group fabric
nv set vrf default router bgp neighbor p0_if type unnumbered
nv set vrf default router bgp neighbor p1_if capabilities source-address lo
```

```
nv set vrf default router bgp neighbor p1_if peer-group fabric
nv set vrf default router bgp neighbor p1_if type unnumbered
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn enable off
nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp peer-group fabric timers connection-retry 5
nv set vrf default router bgp peer-group fabric timers hold 30
nv set vrf default router bgp peer-group fabric timers keepalive 10
nv set vrf default router bgp peer-group rs_client address-family ipv4-unicast enable off
nv set vrf default router bgp peer-group rs_client address-family ipv6-unicast enable off
nv set vrf default router bgp peer-group rs_client address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group rs_client address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group rs_client multihop-ttl 5
nv set vrf default router bgp peer-group rs_client remote-as external
nv set vrf default router bgp peer-group rs_client timers connection-retry 5
nv set vrf default router bgp peer-group rs_client timers hold 30
nv set vrf default router bgp peer-group rs_client timers keepalive 10
nv set vrf default router bgp router-id 27.0.0.7
nv set vrf gw_tenant1 evpn enable on
nv set vrf gw_tenant1 evpn vni 30000
nv set vrf gw_tenant1 loopback ip address 15.3.0.2/32
nv set vrf gw_tenant1 loopback ip address 2001:bad:c0de::2/128
nv set vrf gw_tenant1 router bgp address-family ipv4-unicast enable on
nv set vrf gw_tenant1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf gw_tenant1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf gw_tenant1 router bgp address-family ipv6-unicast enable on
nv set vrf gw_tenant1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf gw_tenant1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf gw_tenant1 router bgp address-family l2vpn-evpn enable on
nv set vrf gw_tenant1 router bgp autonomous-system 65554
nv set vrf gw_tenant1 router bgp enable on
nv set vrf gw_tenant1 router bgp route-export to-evpn route-target 65554:30000
nv set vrf gw_tenant1 router bgp route-import from-evpn route-target ANY:10000
nv set vrf gw_tenant1 router bgp route-import from-evpn route-target auto
nv set vrf gw_tenant1 router bgp router-id 27.0.0.7
nv set vrf internet1 evpn enable on
nv set vrf internet1 evpn vni 10000
nv set vrf internet1 loopback ip address 6.2.0.2/32
nv set vrf internet1 loopback ip address 2001:cafe:feed::2/128
nv set vrf internet1 router bgp address-family ipv4-unicast enable on
nv set vrf internet1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf internet1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf internet1 router bgp address-family ipv6-unicast enable on
nv set vrf internet1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf internet1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf internet1 router bgp address-family l2vpn-evpn enable on
nv set vrf internet1 router bgp autonomous-system 65554
nv set vrf internet1 router bgp enable on
nv set vrf internet1 router bgp neighbor p0_if.60 capabilities source-address internet1
nv set vrf internet1 router bgp neighbor p0_if.60 peer-group l3_pg1
nv set vrf internet1 router bgp neighbor p0_if.60 type unnumbered
nv set vrf internet1 router bgp neighbor p1_if.60 capabilities source-address internet1
nv set vrf internet1 router bgp neighbor p1_if.60 peer-group l3_pg1
nv set vrf internet1 router bgp neighbor p1_if.60 type unnumbered
nv set vrf internet1 router bgp peer-group l3_pg1 address-family ipv4-unicast enable on
nv set vrf internet1 router bgp peer-group l3_pg1 address-family ipv6-unicast enable on
nv set vrf internet1 router bgp peer-group l3_pg1 remote-as external
nv set vrf internet1 router bgp route-export to-evpn route-target 65554:10000
nv set vrf internet1 router bgp route-import from-evpn route-target ANY:20000
nv set vrf internet1 router bgp route-import from-evpn route-target ANY:30000
nv set vrf internet1 router bgp route-import from-evpn route-target auto
nv set vrf internet1 router bgp router-id 27.0.0.7
```

Tenant-HBN-3 Full Configuration

Tenant-HBN-3

```
nv set bridge domain br_default encap 802.1Q
nv set bridge domain br_default type vlan-aware
nv set bridge domain br_default untagged 1
nv set bridge domain br_default vlan 20-21
nv set evpn enable on
nv set interface lo ip address 27.0.0.17/32
nv set interface lo ip address 2001:c001:ff:f00d::11/128
nv set interface lo type loopback
nv set interface p0-1,pf0hpf,pf0vf0-12,pf1hpf,pf1vf0-4 type swp
nv set interface pf0hpf bridge domain br_default access 20
nv set interface pf0vf0 bridge domain br_default access 21
nv set interface vlan20 ip address 45.1.0.1/24
nv set interface vlan20 ip address 2001:c001:b00c::1/96
nv set interface vlan20 vlan 20
nv set interface vlan20-21 ip ipv4 forward on
nv set interface vlan20-21 ip ipv6 forward on
nv set interface vlan20-21 ip vrf tenant1
nv set interface vlan20-21 type svi
nv set interface vlan21 ip address 45.1.1.1/24
nv set interface vlan21 ip address 2001:c001:b00c::1:0:1/96
nv set interface vlan21 vlan 21
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan mac-learning off
nv set nve vxlan source address 27.0.0.17
nv set platform
nv set router bgp enable on
nv set system global anycast-mac 44:38:39:42:42:21
nv set vrf default router bgp address-family ipv4-unicast enable on
```

```
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp autonomous-system 6300656
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor 27.0.0.11 peer-group rs_client
nv set vrf default router bgp neighbor 27.0.0.11 type numbered
nv set vrf default router bgp neighbor 27.0.0.12 peer-group rs_client
nv set vrf default router bgp neighbor 27.0.0.12 type numbered
nv set vrf default router bgp neighbor p0 capabilities source-address lo
nv set vrf default router bgp neighbor p0 peer-group fabric
nv set vrf default router bgp neighbor p0 type unnumbered
nv set vrf default router bgp neighbor p1 capabilities source-address lo
nv set vrf default router bgp neighbor p1 peer-group fabric
nv set vrf default router bgp neighbor p1 type unnumbered
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn enable off
nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp peer-group fabric timers connection-retry 5
nv set vrf default router bgp peer-group fabric timers hold 30
nv set vrf default router bgp peer-group fabric timers keepalive 10
nv set vrf default router bgp peer-group rs_client address-family ipv4-unicast enable off
nv set vrf default router bgp peer-group rs_client address-family ipv6-unicast enable off
nv set vrf default router bgp peer-group rs_client address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group rs_client address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group rs_client multihop-ttl 5
nv set vrf default router bgp peer-group rs_client remote-as external
nv set vrf default router bgp peer-group rs_client timers connection-retry 5
nv set vrf default router bgp peer-group rs_client timers hold 30
nv set vrf default router bgp peer-group rs_client timers keepalive 10
nv set vrf default router bgp router-id 27.0.0.17
nv set vrf tenant1 evpn enable on
nv set vrf tenant1 evpn vni 20000
nv set vrf tenant1 loopback ip address 15.1.0.1/32
nv set vrf tenant1 loopback ip address 2001:c001:c0de::1/128
nv set vrf tenant1 router bgp address-family ipv4-unicast enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf tenant1 router bgp address-family l2vpn-evpn enable on
nv set vrf tenant1 router bgp autonomous-system 6300656
nv set vrf tenant1 router bgp enable on
nv set vrf tenant1 router bgp route-export to-evpn route-target 6300656:20000
nv set vrf tenant1 router bgp route-import from-evpn route-target ANY:10000
nv set vrf tenant1 router bgp route-import from-evpn route-target auto
nv set vrf tenant1 router bgp router-id 27.0.0.17
```

Tenant-HBN-4 Full Configuration

Tenant-HBN4

```
nv set bridge domain br_default encap 802.1Q
nv set bridge domain br_default type vlan-aware
nv set bridge domain br_default untagged 1
nv set bridge domain br_default vlan 20-21
nv set evpn enable on
nv set interface lo ip address 27.0.0.19/32
nv set interface lo ip address 2001:c001:ff:f00d::13/128
nv set interface lo type loopback
nv set interface p0-1,pf0hpf,pf0vf0-12,pf1hpf,pf1vf0-4 type swp
nv set interface pf0hpf bridge domain br_default access 20
nv set interface pf0vf0 bridge domain br_default access 21
nv set interface vlan20 ip address 45.1.2.1/24
nv set interface vlan20 ip address 2001:c001:b00c::2:0:1/96
nv set interface vlan20 vlan 20
nv set interface vlan20-21 ip ipv4 forward on
nv set interface vlan20-21 ip ipv6 forward on
nv set interface vlan20-21 ip vrf tenant1
nv set interface vlan20-21 type svi
nv set interface vlan21 ip address 45.1.3.1/24
nv set interface vlan21 ip address 2001:c001:b00c::3:0:1/96
nv set interface vlan21 vlan 21
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan mac-learning off
nv set nve vxlan source address 27.0.0.19
nv set platform
nv set router bgp enable on
nv set system global anycast-mac 44:38:39:42:42:23
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp autonomous-system 6300658
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor 27.0.0.11 peer-group rs_client
nv set vrf default router bgp neighbor 27.0.0.11 type numbered
nv set vrf default router bgp neighbor 27.0.0.12 peer-group rs_client
nv set vrf default router bgp neighbor 27.0.0.12 type numbered
nv set vrf default router bgp neighbor p0 capabilities source-address lo
nv set vrf default router bgp neighbor p0 peer-group fabric
nv set vrf default router bgp neighbor p0 type unnumbered
```

```
nv set vrf default router bgp neighbor p1 capabilities source-address lo
nv set vrf default router bgp neighbor p1 peer-group fabric
nv set vrf default router bgp neighbor p1 type unnumbered
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group fabric address-family l2vpn-evpn enable off
nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp peer-group fabric timers connection-retry 5
nv set vrf default router bgp peer-group fabric timers hold 30
nv set vrf default router bgp peer-group fabric timers keepalive 10
nv set vrf default router bgp peer-group rs_client address-family ipv4-unicast enable off
nv set vrf default router bgp peer-group rs_client address-family ipv6-unicast enable off
nv set vrf default router bgp peer-group rs_client address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group rs_client address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group rs_client multihop-ttl 5
nv set vrf default router bgp peer-group rs_client remote-as external
nv set vrf default router bgp peer-group rs_client timers connection-retry 5
nv set vrf default router bgp peer-group rs_client timers hold 30
nv set vrf default router bgp peer-group rs_client timers keepalive 10
nv set vrf default router bgp router-id 27.0.0.19
nv set vrf tenant1 evpn enable on
nv set vrf tenant1 evpn vni 20000
nv set vrf tenant1 loopback ip address 15.1.0.2/32
nv set vrf tenant1 loopback ip address 2001:c001:c0de::2/128
nv set vrf tenant1 router bgp address-family ipv4-unicast enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf tenant1 router bgp address-family l2vpn-evpn enable on
nv set vrf tenant1 router bgp autonomous-system 6300658
nv set vrf tenant1 router bgp enable on
nv set vrf tenant1 router bgp route-export to-evpn route-target 6300658:20000
nv set vrf tenant1 router bgp route-import from-evpn route-target ANY:10000
nv set vrf tenant1 router bgp route-import from-evpn route-target auto
nv set vrf tenant1 router bgp router-id 27.0.0.19
```

8.9.5.3.4 Access Control Lists

Access Control Lists (ACLs) are a set of rules that are used to filter network traffic. These rules are used to specify the traffic flows that must be permitted or blocked at networking device interfaces. There are two types of ACLs:

- Stateless ACLs – rules that are applied to individual packets. They inspect each packet individually and permit/block the packets based on the packet header information and the match criteria specified by the rule.
- Stateful ACLs – rules that are applied to traffic sessions/connections. They inspect each packet with respect to the state of the session/connection to which the packet belongs to determine whether to permit/block the packet.

8.9.5.3.4.1 Stateless ACLs

HBN supports configuration of stateless ACLs for IPv4 packets, IPv6 packets, and Ethernet (MAC) frames. The following examples depict how stateless ACLs are configured for each case, with NVUE and with flat files (**cl-acltool**).

Stateless ACLs can be bound to host representor ports (for e.g., pf0hpf_if, pf0vf0_if, etc.) and sub-interfaces of host representor ports (pf0hpf_if.500, pf0vf0_if.999, etc).

NVUE Examples for Stateless ACLs

NVUE IPv4 ACLs Example

The following is an example of an ingress IPv4 ACL that permits DHCP request packets ingressing on the **pf0vf1_if.999** interface towards the DHCP server:

```
root@hbn01-host01:~# nv set acl acl1_ingress type ipv4
root@hbn01-host01:~# nv set acl acl1_ingress rule 100 match ip protocol udp
root@hbn01-host01:~# nv set acl acl1_ingress rule 100 match ip dest-port 67
root@hbn01-host01:~# nv set acl acl1_ingress rule 100 match ip source-port 68
```

```
root@hbn01-host01:~# nv set acl acl1_ingress rule 100 action permit
```

Bind the ingress IPv4 ACL to host representor port **pf0vf1_if.999** of BlueField in the inbound direction:

```
root@hbn01-host01:~# nv set interface pf0vf1_if.999 acl acl1_ingress inbound
root@hbn01-host01:~# nv config apply
```

The following is an example of an egress IPv4 ACL that permits DHCP reply packets egressing out of the **pf0vf1_if.999** port towards the DHCP client:

```
root@hbn01-host01:~# nv set acl acl2_egress type ipv4
root@hbn01-host01:~# nv set acl acl2_egress rule 200 match ip protocol udp
root@hbn01-host01:~# nv set acl acl2_egress rule 200 match ip dest-port 68
root@hbn01-host01:~# nv set acl acl2_egress rule 200 match ip source-port 67
root@hbn01-host01:~# nv set acl acl2_egress rule 200 action permit
```

Bind the egress IPv4 ACL to host representor port **pf0vf1_if.999** of BlueField in the outbound direction:

```
root@hbn01-host01:~# nv set interface pf0vf1_if.999 acl acl2_egress outbound
root@hbn01-host01:~# nv config apply
```

NVUE IPv6 ACLs Example

The following is an example of an ingress IPv6 ACL that permits traffic with matching **dest-ip** and **protocol tcp** ingress on port **pf0hpf_if**:

```
root@hbn01-host01:~# nv set acl acl5_ingress type ipv6
root@hbn01-host01:~# nv set acl acl5_ingress rule 100 match ip protocol tcp
root@hbn01-host01:~# nv set acl acl5_ingress rule 100 match ip dest-ip 48:2034::80:9
root@hbn01-host01:~# nv set acl acl5_ingress rule 100 action permit
```

Bind the ingress IPv6 ACL to host representor port **pf0hpf_if** of BlueField in the inbound direction:

```
root@hbn01-host01:~# nv set interface pf0hpf_if acl acl5_ingress inbound
root@hbn01-host01:~# nv config apply
```

The following is an example of an egress IPv6 ACL that permits traffic with matching **source-ip** and **protocol tcp** egressing out of port **pf0hpf_if**:

```
root@hbn01-host01:~# nv set acl acl6_egress type ipv6
root@hbn01-host01:~# nv set acl acl6_egress rule 101 match ip protocol tcp
root@hbn01-host01:~# nv set acl acl6_egress rule 101 match ip source-ip 48:2034::80:9
root@hbn01-host01:~# nv set acl acl6_egress rule 101 action permit
```

Bind the egress IPv6 ACL to host representor port **pf0hpf_if** of BlueField in the outbound direction:

```
root@hbn01-host01:~# nv set interface pf0hpf_if acl acl6_egress outbound
root@hbn01-host01:~# nv config apply
```

NVUE MAC ACLs Example

The following is an example of an ingress MAC ACL that permits traffic with matching **source-mac** and **dest-mac** ingressing to port **pf0hpf_if**:

```
root@hbn01-host01:~# nv set acl acl3_ingress type mac
root@hbn01-host01:~# nv set acl acl3_ingress rule 1 match mac source-mac 00:00:00:00:00:0a
root@hbn01-host01:~# nv set acl acl3_ingress rule 1 match mac dest-mac 00:00:00:00:00:0b
root@hbn01-host01:~# nv set interface pf0hpf_if acl acl3_ingress inbound
```

Bind the ingress MAC ACL to host representor port **pf0hpf_if** of BlueField in the inbound direction:

```
root@hbn01-host01:~# nv set interface pf0hpf_if acl acl3_ingress inbound
root@hbn01-host01:~# nv config apply
```

The following is an example of an egress MAC ACL that permits traffic with matching **source-mac** and **dest-mac** egressing out of port **pf0hpf_if**:

```
root@hbn01-host01:~# nv set acl acl4_egress type mac
root@hbn01-host01:~# nv set acl acl4_egress rule 2 match mac source-mac 00:00:00:00:00:0b
root@hbn01-host01:~# nv set acl acl4_egress rule 2 match mac dest-mac 00:00:00:00:00:0a
root@hbn01-host01:~# nv set acl acl4_egress rule 2 action permit
```

Bind the egress MAC ACL to host representor port **pf0hpf_if** of BlueField in the outbound direction:

```
root@hbn01-host01:~# nv set interface pf0hpf_if acl acl4_egress outbound
root@hbn01-host01:~# nv config apply
```

Flat Files (cl-acltool) Examples for Stateless ACLs

For the same examples cited above, the following are the corresponding ACL rules which must be configured under **/etc/cumulus/acl/policy.d/<rule_name.rules>** followed by invoking **cl-acltool -i**. The rules in **/etc/cumulus/acl/policy.d/<rule_name.rules>** are configured using Linux iptables/ip6tables/ebtables.

Flat Files IPv4 ACLs Example

The following example configures an ingress IPv4 ACL rule matching with DHCP request under **/etc/cumulus/acl/policy.d/<rule_name.rules>** with the ingress interface as the host representor of BlueField followed by invoking **cl-acltool -i**:

```
[iptables]
## ACL acl1_ingress in dir inbound on interface pf1vf1_if.999 ##
-t filter -A FORWARD -m physdev --physdev-in pf1vf1_if.999 -p udp --sport 68 --dport 67 -j ACCEPT
```

The following example configures an egress IPv4 ACL rule matching with DHCP reply under **/etc/cumulus/acl/policy.d/<rule_name.rules>** with the egress interface as the host representor of BlueField followed by invoking **cl-acltool -i**:

```
[iptables]
## ACL acl2_egress in dir outbound on interface pf1vf1_if.999 ##
-t filter -A FORWARD -m physdev --physdev-out pf1vf1_if.999 -p udp --sport 67 --dport 68 -j ACCEPT
```

Flat File IPv6 ACLs Example

The following example configures an ingress IPv6 ACL rule matching with **dest-ip** and **tcp** protocol under **/etc/cumulus/acl/policy.d/<rule_name.rules>** with the ingress interface as the host representor of BlueField followed by invoking **cl-acltool -i**:

```
[ip6tables]
## ACL acl5_ingress in dir inbound on interface pf0hpf_if ##
-t filter -A FORWARD -m physdev --physdev-in pf0hpf_if -d 48:2034::80:9 -p tcp -j ACCEPT
```

The following example configures an egress IPv6 ACL rule matching with **source-ip** and **tcp** protocol under **/etc/cumulus/acl/policy.d/<rule_name.rules>** with the egress interface as the host representor of BlueField followed by invoking **cl-acltool -i**:

```
[ip6tables]
## ACL acl6_egress in dir outbound on interface pf0hpf_if ##
-t filter -A FORWARD -m physdev --physdev-out pf0hpf_if -s 48:2034::80:9 -p tcp -j ACCEPT
```

Flat Files MAC ACLs Example

The following example configures an ingress MAC ACL rule matching with **source-mac** and **dest-mac** under **/etc/cumulus/acl/policy.d/<rule_name.rules>** with the ingress interface as the host representor of BlueField followed by invoking **cl-acltool -i**:

```
[ebtables]
## ACL acl3_ingress in dir inbound on interface pf0hpf_if ##
-t filter -A FORWARD -m physdev --physdev-in pf0hpf_if -s 00:00:00:00:00:0a/ff:ff:ff:ff:ff:ff -d 00:00:00:00:00:0b/
ff:ff:ff:ff:ff:ff -j ACCEPT
```

The following example configures an egress MAC ACL rule matching with **source-mac** and **dest-mac** under **/etc/cumulus/acl/policy.d/<rule_name.rules>** with egress interface as host representor of BlueField followed by invoking **cl-acltool -i**:

```
[ebtables]
## ACL acl4_egress in dir outbound on interface pf0hpf_if ##
-t filter -A FORWARD -m physdev --physdev-out pf0hpf_if -s 00:00:00:00:00:0b/ff:ff:ff:ff:ff:ff -d
00:00:00:00:00:0a/ff:ff:ff:ff:ff:ff -j ACCEPT
```

8.9.5.3.4.2 Stateful ACLs

Stateful ACLs facilitate monitoring and tracking traffic flows to enforce per-flow traffic filtering (unlike stateless ACLs which filter traffic on a per-packet basis). HBN supports stateful ACLs using reflexive ACL mechanism. Reflexive ACL mechanism is used to allow initiation of connections from "within" the network to "outside" the network and allow only replies to the initiated connections from "outside" the network (or vice versa).

HBN supports stateful ACL configuration for IPv4 traffic. Stateful ACL configuration is supported for TCP, UDP, and ICMP protocols.

Stateful ACLs can be bound to host representor ports (for e.g., pf0hpf_if, pf0vf0_if, etc.) and sub-interfaces of host representor ports (pf0hpf_if.500, pf0vf0_if.999, etc).

Stateful ACLs can be applied for native routed traffic (north-south underlay routed traffic in EVPN deployments), EVPN bridged traffic (east-west overlay bridged/L2 traffic in EVPN deployments) and EVPN routed traffic (east-west overlay routed traffic in EVPN deployments). Stateful ACLs applied for native routed traffic are called "Native-L3 stateful ACLs". Stateful ACLs applied for

EVPN bridged traffic and EVPN routed traffic are called "EVPN-L2 stateful ACLs" and "EVPN-L3 stateful ACLs", respectively.

Stateful ACLs in HBN are enabled by default. To enable stateful ACL functionality, use the following NVUE commands:

```
root@hbn03-host00:~# nv set system reflexive-acl enable
root@hbn03-host00:~# nv config apply
```

If using flat-file configuration (and not NVUE), edit the file **/etc/cumulus/nl2docad.d/acl.conf** and set the knob **rflx.reflexive_acl_enable** to **TRUE**. To apply this change, execute:

```
root@hbn03-host00:~# supervisorctl start nl2doca-reload
```

NVUE Example for Stateful ACLs

The following is an example of allowing HTTP (TCP) connection originated by the host, where BlueField is hosted, to an HTTP server (with the IP address 11.11.11.11) on an external network. Two sets of ACLs matching with CONNTRACK state must be configured for a CONNTRACK entry to be established and offloaded to hardware:

- Configure an ACL rule matching TCP/HTTP connection/flow details with CONNTRACK state of NEW, ESTABLISHED and bind it to the host representor port or its sub-interface in the inbound direction.
- Configure an ACL rule matching TCP/HTTP connection/flow details with CONNTRACK state of ESTABLISHED and bind it to the host representor port or its sub-interface in the outbound direction.

Stateful ACLs should be bound to a a host representor port or its sub-interface. In this example, the host representor port is **pf1vf7_if**.

1. Configure the ingress ACL rule:

   ```
   root@hbn03-host00:~# nv set acl allow_tcp_conn_from_host rule 11 action permit
   root@hbn03-host00:~# nv set acl allow_tcp_conn_from_host rule 11 match conntrack new
   root@hbn03-host00:~# nv set acl allow_tcp_conn_from_host rule 11 match conntrack established
   root@hbn03-host00:~# nv set acl allow_tcp_conn_from_host rule 11 match ip dest-ip 11.11.11.11/32
   root@hbn03-host00:~# nv set acl allow_tcp_conn_from_host rule 11 match ip dest-port 80
   root@hbn03-host00:~# nv set acl allow_tcp_conn_from_host rule 11 match ip protocol tcp
   root@hbn03-host00:~# nv set acl allow_tcp_conn_from_host type ipv4
   ```

2. Bind this ACL to the physical interface in the inbound direction:

   ```
   root@hbn03-host00:~# nv set interface pf1vf7_if acl allow_tcp_conn_from_host inbound
   root@hbn03-host00:~# nv config apply
   ```

3. Configure the egress ACL rule:

   ```
   root@hbn03-host00:~# nv set acl allow_tcp_resp_from_server rule 21 action permit
   root@hbn03-host00:~# nv set acl allow_tcp_resp_from_server rule 21 match conntrack established
   root@hbn03-host00:~# nv set acl allow_tcp_resp_from_server rule 21 match ip protocol tcp
   root@hbn03-host00:~# nv set acl allow_tcp_resp_from_server type ipv4
   root@hbn03-host00:~# nv config apply
   ```

4. Bind this ACL to the host representor port in the outbound direction:

   ```
   root@hbn03-host00:~# nv set interface pf1vf7_if acl allow_tcp_resp_from_server outbound
   root@hbn03-host00:~# nv config apply
   ```

Flat Files (cl-acltool) Example for Stateful ACLs

For the same NVUE example for stateful ACLs cited above (HTTP server at IP address 11.11.11.11 on an external network), the following are the corresponding ACL rules which must be configured under **/etc/cumulus/acl/policy.d/<rule_name.rules>** followed by invoking **cl-acltool -i** to install the rules in BlueField hardware.

1. Configure an ingress ACL rule matching with TCP flow details and CONNTRACK state of NEW, ESTABLISHED under **/etc/cumulus/acl/policy.d/stateful_acl.rules** with the ingress interface as the host representor port or its sub-interface as the ingress interface, followed by invoking **cl-acltool -i**:

```
[iptables]
## ACL allow_tcp_conn_from_host in dir inbound on interface pf1vf7_if ##
-t filter -A FORWARD -m physdev --physdev-in pf1vf7_if -p tcp -d 11.11.11.11/32 --dport 80 -m conntrack --
ctstate EST,NEW -j ACCEPT -m mark --mark 0xdead
```

2. Configure an egress ACL rule matching the TCP flow and CONNTRACK state of ESTABLISHED, RELATED under **/etc/cumulus/acl/policy.d/stateful_acl.rules** file with the egress interface as the host representor port or its sub-interface as the egress interface, followed by invoking **cl-acltool -i**:

```
[iptables]
## ACL allow_tcp_resp_from_server in dir outbound on interface pf1vf7_if ##
-t filter -A FORWARD -m physdev --physdev-out  pf1vf7_if  -p tcp -s 11.11.11.11/32 --sport 80 -m conntrack
--ctstate EST -j ACCEPT -m mark --mark 0xdead
```

8.9.5.3.5 Network Address Translation

Network address translation (NAT) allows a network to use one set of IP addresses for internal traffic and another set for external traffic. Besides preventing the depletion of IPv4 addresses, NAT enables the use of private address space internally while still providing access to the Internet. NAT essentially consists of a set of rules for translating between public and private IP addresses and ports. Both static and dynamic NAT rules are configured on outbound or inbound uplink interfaces or sub-interfaces of uplink ports. These NAT rules should be applied on all public egress and ingress facing uplink interfaces and their sub-interfaces.

HBN supports Source NAT (SNAT), which allows traffic from the private network to reach public networks by changing the packet's source address. HBN also supports Destination NAT (DNAT), which allows incoming traffic from public networks to reach private networks by changing the packet's destination address. HBN SNAT and DNAT are supported for IPv4 and three protocols: ICMP, TCP, and UDP.

SNAT and DNAT can be configured in two types:

- Static NAT – Provides a permanent mapping between one private IP address and a single public address. Static NAT is stateless and does not use connection tracking. Static NAT is enabled by default and supports port address translation (PAT) with a one-to-one mapping between private and public ports. The static NAT mapping rules are not flushed or timed out from rule tables.
- Dynamic NAT – This is a stateful NAT which uses the OVS connection tracking infrastructure to monitor NAT connections. Dynamic NAT is disabled by default, and users must enable dynamic mode before setting up dynamic NAT rules (see section "HBN Service Configuration#Dynamic NAT Configuration"). Dynamic NAT maps private IP addresses to a

range of public addresses and supports port ranges for PAT. Users are responsible for managing the pool of IP addresses and ports used in the configuration of dynamic NAT and PAT. It is recommended to use a NAT IP address range that is a power of two.

8.9.5.3.5.1 Source NAT (SNAT)

Static SNAT Configuration

Static NAT is enabled by default and is stateless.

NVUE Examples for Static SNAT

NVUE ICMP Example

The following is an example of an SNAT rule translating ICMP packets matching source IP 30.30.30.2 to source IP 100.100.100.2:

```
nv set acl acl_1 rule 1 action source-nat translate-ip 100.100.100.1
nv set acl acl_1 rule 1 match ip protocol icmp
nv set acl acl_1 rule 1 match ip source-ip 30.30.30.2
nv set acl acl_1 type ipv4
```

Bind the egress ICMP NAT ACL uplink port **p0_if** and **p1_if** of BlueField in the outbound direction:

```
nv set interface p0_if acl acl_1 outbound
nv set interface p1_if acl acl_1 outbound
```

NVUE UDP with PAT Example

The following is an example of an SNAT rule translating UDP packets matching source IP 21.1.0.18 and source port 10011 to source IP 199.5.1.1 and source port 11011:

```
nv set acl stnatpat rule 1 action source-nat translate-ip 199.5.1.1
nv set acl stnatpat rule 1 action source-nat translate-port 11011
nv set acl stnatpat rule 1 match ip protocol udp
nv set acl stnatpat rule 1 match ip source-ip 21.1.0.18
nv set acl stnatpat rule 1 match ip source-port 10011
nv set acl stnatpat type ipv4
```

Bind the egress UDP NAT ACL to uplink port **p0_if.10** and **p1_if.10** of BlueField in the outbound direction:

```
nv set interface p0_if.10,p1_if.10 acl stnatpat outbound
```

Flat Files (cl-acltool) Example for Static SNAT

```
[iptables]

## ACL acl_1 in dir outbound on interface p0_if ##
-t nat -A POSTROUTING  -m physdev --physdev-out p0_if -s 30.30.30.2 -p icmp  -j SNAT --to-source 100.100.100.1 -m
mark --mark 0xdead

## ACL acl_1 in dir outbound on interface p1_if ##
-t nat -A POSTROUTING  -m physdev --physdev-out p1_if -s 30.30.30.2 -p icmp  -j SNAT --to-source 100.100.100.1 -m
mark --mark 0xdead

## ACL stnatpat in dir outbound on interface p0_if.10 ##
-t nat -A POSTROUTING -o p0_if.10 -s 21.1.0.18 -p udp --sport 10011  -j SNAT --to-source 199.5.1.1:11011 -m mark --
mark 0xdead

## ACL stnatpat in dir outbound on interface p1_if.10 ##
-t nat -A POSTROUTING -o p1_if.10 -s 21.1.0.18 -p udp --sport 10011  -j SNAT --to-source 199.5.1.1:11011 -m mark --
mark 0xdead
```

Dynamic NAT Configuration

Dynamic NAT is stateful and relies on underlying connection tracking.

Enabling NVUE Dynamic NAT

```
nv set system nat mode dynamic
```

NVUE Examples for Dynamic SNAT

NVUE Dynamic SNAT with TCP

The following is an example of an SNAT rule translating TCP packets matching source IP 21.1.0.0/16 to an IP range of 199.10.0.1 to 199.10.255.254:

```
nv set acl dnat1 rule 1 action source-nat translate-ip 199.10.0.1 to 199.10.255.254
nv set acl dnat1 rule 1 match ip protocol tcp
nv set acl dnat1 rule 1 match ip source-ip 21.1.0.0/16
nv set acl dnat1 type ipv4
```

Bind the egress TCP NAT to uplink sub-interfaces **p0_if.10** and **p1_if.10**:

```
nv set interface p0_if.10,p1_if.10 acl dnat1 outbound
```

NVUE Dynamic SNAT with TCP and PAT

The following is an example of an SNAT rule translating TCP packets matching source IP 30.30.30.0/29 and source port 5000 to an IP range 100.100.100.8 to 100.100.100.15 and source port 10000:

```
nv set acl acl_2 rule 1 action source-nat translate-ip 100.100.100.8 to 100.100.100.15
nv set acl acl_2 rule 1 action source-nat translate-port 10000
nv set acl acl_2 rule 1 match ip protocol  tcp
nv set acl acl_2 rule 1 match ip source-ip 30.30.30.0/29
nv set acl acl_2 rule 1 match ip source-port  5000
nv set acl acl_2 type ipv4
```

Bind the egress TCP NAT to uplink sub-interfaces **p0_if** and **p1_if**:

```
nv set interface p0_if,p1_if acl acl_2 outbound
```

Flat Files (cl-acltool) Example for Dynamic SNAT

```
## ACL dnat1 in dir outbound on interface p0_if.10 ##
-t nat -A POSTROUTING -o p0_if.10 -s 21.1.0.0/16 -p tcp  -j SNAT --to-source 199.10.0.1-199.10.255.254 -m mark --
mark 0xdead
## ACL dnat1 in dir outbound on interface p1_if.10 ##
-t nat -A POSTROUTING -o p1_if.10 -s 21.1.0.0/16 -p tcp -j SNAT --to-source 199.10.0.1-199.10.255.254 -m mark --
mark 0xdead

## ACL acl_2 in dir outbound on interface p0_if ##
-t nat -A POSTROUTING  -m physdev --physdev-out p0_if -s 30.30.30.0/29 -p tcp --sport 5000  -j SNAT --to-source
100.100.100.8-100.100.100.15:10000 -m mark --mark 0xdead
## ACL acl_2 in dir outbound on interface p1_if ##
-t nat -A POSTROUTING  -m physdev --physdev-out p1_if -s 30.30.30.0/29 -p tcp --sport 5000  -j SNAT --to-source
100.100.100.8-100.100.100.15:10000 -m mark --mark 0xdead
```

The following is an example where a pool of private IP addresses is translated to a public address using a unique port number from the available range:

```
nv set acl dnatpat1 rule 1 action source-nat translate-ip 199.9.1.1
nv set acl dnatpat1 rule 1 match ip protocol tcp
```

```
nv set acl dnatpat1 rule 1 match ip source-ip 21.1.0.0/16
nv set acl dnatpat1 rule 1 action source-nat translate-port 10001-24000
nv set acl dnatpat1 type ipv4
nv set interface p0_if.10,p1_if.10 acl dnatpat1 outbound
```

8.9.5.3.5.2 Destination NAT (DNAT)

Static DNAT Configuration

Static NAT is enabled by default and is stateless.

NVUE Examples for Static DNAT

NVUE ICMP Example

The following is an example of an SNAT rule translating ICMP packets matching incoming destination IP 100.100.100.1 to destination IP 30.30.30.2

```
nv set acl acl_1 rule 1 action dest-nat translate-ip 30.30.30.2
nv set acl acl_1 rule 1 match ip protocol icmp
nv set acl acl_1 rule 1 match ip dest-ip  100.100.100.1
nv set acl acl_1 type ipv4
```

Bind the ingress ICMP NAT ACL to uplink port p0_if and p1_if of BlueField in the inbound direction:

```
nv set interface p0_if acl acl_1 inbound
nv set interface p1_if acl acl_1 inbound
```

NVUE UDP with PAT example

Flat Files (cl-acltool) Example for Static DNAT

```
## ACL acl_1 in dir inbound on interface p0_if ##
-t nat -A PREROUTING  -m physdev --physdev-in p0_if -d 100.100.100.1 -p icmp  -j DNAT --to-destination 30.30.30.2
 -m mark --mark 0xdead

## ACL acl_1 in dir inbound on interface p1_if ##
-t nat -A PREROUTING  -m physdev --physdev-in p1_if -d 100.100.100.1 -p icmp  -j DNAT --to-destination 30.30.30.2
 -m mark --mark 0xdead
```

Dynamic DNAT configuration

This is common configuration for DNAT and SNAT so please see section "HBN Service Configuration#Dynamic NAT Configuration".

Enabling NVUE Dynamic DNAT

NVUE Examples for Dynamic DNAT

NVUE Dynamic DNAT with TCP and PAT

The following is an example of an DNAT rule translating TCP packets matching destination IP 100.100.100.1/29 and destination port 5000 to an IP range 30.30.30.1 to 30.30.30.6 and translated destination port 1234:

```
nv set acl acl_3 rule 1 action dest-nat translate-ip 30.30.30.1 to 30.30.30.6
nv set acl acl_3 rule 1 action dest-nat translate-port 1234
nv set acl acl_3 rule 1 match ip protocol  tcp
nv set acl acl_3 rule 1 match ip dest-ip 100.100.100.1/29
nv set acl acl_3 rule 1 match ip dest-port  5000
nv set acl acl_3 type ipv4
```

Bind the ingress TCP NAT ACL to uplink sub-interfaces **p0_if** and **p1_if**:

```
nv set interface p0_if acl acl_6 inbound
nv set interface p1_if acl acl_6 inbound
```

Flat Files (cl-acltool) Example for Dynamic DNAT

```
## ACL acl_6 in dir inbound on interface p0_if ##
-t nat -A PREROUTING  -m physdev --physdev-in p0_if -d 100.100.100.1/29 -p udp  -j DNAT --to-destination
30.30.30.1-30.30.30.6 -m mark --mark 0xdead

## ACL acl_6 in dir inbound on interface p1_if ##
-t nat -A PREROUTING  -m physdev --physdev-in p1_if -d 100.100.100.1/29 -p udp  -j DNAT --to-destination
30.30.30.1-30.30.30.6 -m mark --mark 0xdead
```

More examples of dynamic DNAT with NVUE

In practice multiple rules can be combined as well as multiple dnat acl can be applied to same uplink ports.

ICMP

```
nv set acl dyndnaticmp rule 1 action dest-nat translate-ip 21.1.0.19 to 21.1.0.20
nv set acl dyndnaticmp rule 1 match ip dest-ip 199.101.6.0/30
nv set acl dyndnaticmp rule 1 match ip protocol icmp
nv set acl dyndnaticmp rule 2 action dest-nat translate-ip 120.19.0.19 to 120.19.0.20
nv set acl dyndnaticmp rule 2 match ip dest-ip 199.101.6.4/30
nv set acl dyndnaticmp rule 2 match ip protocol icmp
nv set acl dyndnaticmp rule 3 action dest-nat translate-ip 21.1.0.15 to 21.1.0.16
nv set acl dyndnaticmp rule 3 match ip dest-ip 199.101.6.8/30
nv set acl dyndnaticmp rule 3 match ip protocol icmp
nv set acl dyndnaticmp type ipv4
```

TCP

```
nv set acl dyndnattcp rule 1 action dest-nat translate-ip 21.1.0.19 to 21.1.0.20
nv set acl dyndnattcp rule 1 match ip dest-ip 199.101.7.0/30
nv set acl dyndnattcp rule 1 match ip protocol tcp
nv set acl dyndnattcp rule 2 action dest-nat translate-ip 120.19.0.19 to 120.19.0.20
nv set acl dyndnattcp rule 2 match ip dest-ip 199.101.7.4/30
nv set acl dyndnattcp rule 2 match ip protocol tcp
nv set acl dyndnattcp rule 3 action dest-nat translate-ip 21.1.0.15 to 21.1.0.16
nv set acl dyndnattcp rule 3 match ip dest-ip 199.101.7.8/30
nv set acl dyndnattcp rule 3 match ip protocol tcp
nv set acl dyndnattcp type ipv4
```

UDP

```
nv set acl dyndnatudp rule 1 action dest-nat translate-ip 21.1.0.19 to 21.1.0.20
nv set acl dyndnatudp rule 1 match ip dest-ip 199.101.8.0/30
nv set acl dyndnatudp rule 1 match ip protocol udp
nv set acl dyndnatudp rule 2 action dest-nat translate-ip 120.19.0.19 to 120.19.0.20
nv set acl dyndnatudp rule 2 match ip dest-ip 199.101.8.4/30
nv set acl dyndnatudp rule 2 match ip protocol udp
nv set acl dyndnatudp rule 3 action dest-nat translate-ip 21.1.0.15 to 21.1.0.16
nv set acl dyndnatudp rule 3 match ip dest-ip 199.101.8.8/30
nv set acl dyndnatudp rule 3 match ip protocol udp
nv set acl dyndnatudp type ipv4
```

Applying these to uplink ports

```
nv set interface p0_if.10,p1_if.10 acl dyndnaticmp inbound
nv set interface p0_if.10,p1_if.10 acl dyndnattcp inbound
nv set interface p0_if.10,p1_if.10 acl dyndnatudp inbound
```

8.9.5.3.6 Bidirectional Forwarding Detection (BFD) for BGP

BFD functionality is derived from the upstream FRRouting (FRR) suite, which includes a dedicated daemon called bfdd.
HBN supports BFD for BGP from 3.0.0 release. Single-Hop (UDP port 3784) and Multi-Hop (UDP port 4784) BFD supports are present for both IPv4 and IPv6. Asynchronous Mode (Periodically send BFD Control Packets to one another) is the default and only mode supported. *BFD Echo Function is NOT supported.*

8.9.5.3.6.1 BFD Configuration

BFD configuration has two parts , BFD profile configuration and BFD profile attachment to client like BGP.

BFD Profile Configuration

We have follwing profile settings for BFD sessions and it is present under "nv set/unset/show router bfd profile <profile-name> " command.

Sr. No.	Command	Details	Range	Default
1.	detect-multiplier		1-255	3
2.	receive-interval	Configures the minimum interval that this system is capable of receiving control packets.	10-4294967ms	300ms
3.	transmit-interval	The minimum transmission interval (less jitter) that this system wants to use to send BFD control packets.	10-4294967ms	300 ms
4.	shutdown	Enables or disables the peer.	on/off	off
5.	passive-mode	Mark session as passive: a passive session will not attempt to start the connection and will wait for control packets from peer before it begins replying.	on/off	off
6.	minimum-ttl	For multi hop sessions only: configure the minimum expected TTL for an incoming BFD control packet.	1-254	254

```
nv set router bfd profile pf1 detect-multiplier 21
nv set router bfd profile pf1 min-rx-interval 100
nv set router bfd profile pf1 min-tx-interval 100
nv set router bfd profile pf1 minimum-ttl 253
nv set router bfd profile pf1 passive-mode on
```

```
nv set router bfd profile pf1 shutdown on
```

BGP Configuration for BFD

Apply BFD profile with BGP neighbor/peer-group. This will create/update BFD session for the respective BGP neighbor/peer-group.

```
nv set vrf default router bgp neighbor p0_if.100 bfd profile pf1
nv set vrf default router bgp neighbor 6000::23 bfd profile pf1
nv set vrf default router bgp peer-group EBGP_MHOP_V4 bfd profile pf1
```

8.9.5.3.6.2 BFD Show Commands

Following commands can be used to display BFD sessions.

```
root@doca-hbn-service-bf3-s13-1-ipmi:/etc/cumulus/sougatab/frr-bfd-upgrade-debs# nv show vrf default router bfd
peers --view=brief

MHop - Multihop, Local - Local, Peer - Peer, Interface - Interface, State -
State, Passive - Passive Mode, Time - Up/Down Time, Type - Config Type

LocalId     MHop   Local                     Peer                Interface  State  Passive  Time        Type
----------  -----  ------------------------  ------------------  ---------  -----  -------  ----------  -------
417705214   True   6.0.0.24                  6.0.0.23                       down   False    6:19:46:05  dynamic
770667106   False  fe80::a288:c2ff:fe2c:5253  fe80::202:ff:fe00:33  p1_if      down   False    0:03:23:27  dynamic
918556010   False  fe80::a288:c2ff:fe2c:5252  fe80::202:ff:fe00:2f  p0_if.100  up     False    6:19:38:53  dynamic
1668048197  True   6000::24                  6000::23                       down   False    6:19:46:05  dynamic
3190546666  False  fe80::a288:c2ff:fe2c:5252  fe80::202:ff:fe00:2f  p0_if.101  up     False    6:19:38:53  dynamic
root@doca-hbn-service-bf3-s13-1-ipmi:/etc/cumulus/sougatab/frr-bfd-upgrade-debs# nv show vrf default router bfd
peers --view=standard

MHop - Multihop, Local - Local, Peer - Peer, Interface - Interface, State -
State, Passive - Passive Mode, Time - Up/Down Time, Type - Config Type, MinTTL -
Minimum TTL, Multiplier - Detect Multiplier, MinRx - Min Rx Interval, MinTx -
Min Tx Interval, CtrlIn - Control Packet Input, CtrlOut - Control Packet Output

LocalId     MHop   Local                     Peer                Interface  State  Passive  Time        Type
MinTTL  Multiplier  MinRx  MinTx  CtrlIn   CtrlOut
----------  -----  ------------------------  ------------------  ---------  -----  -------  ----------  -------
------  ----------  -----  -----  -------  -------
417705214   True   6.0.0.24                  6.0.0.23                       down   False    6:19:46:14  dynamic
251     21          100    100    12517    698545
770667106   False  fe80::a288:c2ff:fe2c:5253  fe80::202:ff:fe00:33  p1_if      down   False    0:03:23:36  dynamic
22          102    102    0        13950
918556010   False  fe80::a288:c2ff:fe2c:5252  fe80::202:ff:fe00:2f  p0_if.100  up     False    6:19:39:02  dynamic
21          100    100    3270251  3448733
1668048197  True   6000::24                  6000::23                       down   False    6:19:46:14  dynamic
251     21          100    100    12373    698358
3190546666  False  fe80::a288:c2ff:fe2c:5252  fe80::202:ff:fe00:2f  p0_if.101  up     False    6:19:39:02  dynamic
22          102    102    3270331  3448614
root@doca-hbn-service-bf3-s13-1-ipmi:/etc/cumulus/sougatab/frr-bfd-upgrade-debs# nv show vrf default router bfd
peers --view=detail

MHop - Multihop, Local - Local, Peer - Peer, Interface - Interface, State -
State, RemoteId - Remote ID, Passive - Passive Mode, Time - Up/Down Time, Type -
Config Type, MinTTL - Minimum TTL, Diag - Diagnostic, RemoteDiag - Remote
Diagnostic, Multiplier - Detect Multiplier, MinRx - Min Rx Interval, MinTx - Min
Tx Interval, RemoteMultiplier - Remote Detect Multiplier, RemoteRx - Remote Rx
Interval, RemoteTx - Remote Tx Interval, EchoRx - Echo Rx Interval, EchoTx -
Echo Tx Interval, RemoteEchoRx - Remote Echo Rx Interval, RTTMin - RTT Min,
RTMax - RTT Max, RTTAvg - RTT Avg, CtrlIn - Control Packet Input, CtrlOut -
Control Packet Output, EchoIn - Echo Packet Input, EchoOut - Echo Packet Output,
Up - Session Up, Down - Session Down, Zebra - Zebra Notification

LocalId     MHop   Local                     Peer                Interface  State  RemoteId  Passive  Time
Type     MinTTL  Diag                    RemoteDiag  Multiplier  MinRx  MinTx  RemoteMultiplier  RemoteRx
RemoteTx  EchoRx  EchoTx  RemoteEchoRx  RTTMin  RTMax  RTTAvg  CtrlIn   CtrlOut  EchoIn  EchoOut  Up  Down  Zebra
----------  -----  ------------------------  ------------------  ---------  -----  --------  -------  ----------
-------  ------  ------------------------  ----------  ----------  -----  -----  ----------------  --------
--------  ------  ------  -------------  ------  -----  ------  -------  -------  ------  -------  --  ----  -----
417705214   True   6.0.0.24                  6.0.0.23                       down   0         False
6:19:46:19  dynamic  251     control-detection-expired  ok          21          100    100    30
300       300       50      0       50            0       0       0       12517    698550   0       0        1   1
3
770667106   False  fe80::a288:c2ff:fe2c:5253  fe80::202:ff:fe00:33  p1_if      down   0         False
0:03:23:41  dynamic          ok                        ok          22          102    102    3
1000      1000      50      0       0             0       0       0       0        13954    0       0        0   0
1
918556010   False  fe80::a288:c2ff:fe2c:5252  fe80::202:ff:fe00:2f  p0_if.100  up     6         False
6:19:39:07  dynamic          ok                        ok          21          100    100    20
200       200       50      0       0             0       0       0       3270275  3448759  0       0        2   1
5
1668048197  True   6000::24                  6000::23                       down   0         False
6:19:46:19  dynamic  251     control-detection-expired  ok          21          100    100    30
300       300       50      0       50            0       0       0       12373    698363   0       0        1   1
7
3190546666  False  fe80::a288:c2ff:fe2c:5252  fe80::202:ff:fe00:2f  p0_if.101  up     7         False
6:19:39:07  dynamic          ok                        ok          22          102    102    20
```

```
200       200       50      0       0            0       0       0       3270354  3448640  0       0        2   1
6
```

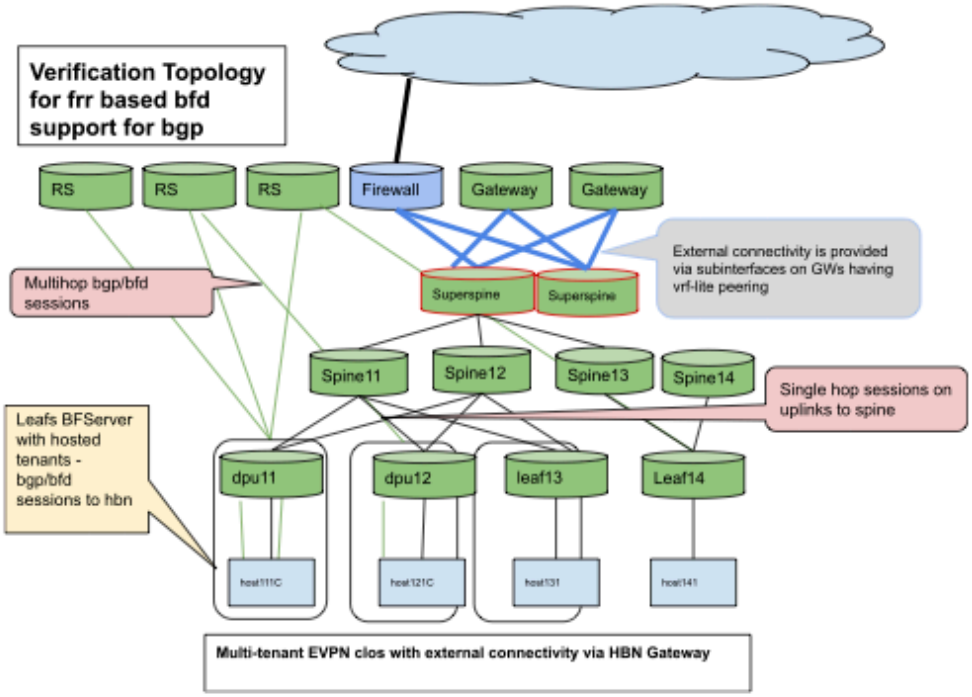
8.9.5.3.6.3 Sample BFD and BGP Configurations for a HBN Node <--> CumulusLinux Node Connection

```
HBN:
nv set bridge domain br_default vlan 111 vni 1000111
nv set bridge domain br_default vlan 112 vni 1000112
nv set bridge domain br_default vlan 213 vni 1000213
nv set bridge domain br_default vlan 214 vni 1000214
nv set evpn enable on
nv set interface lo ip address 6.0.0.24/32
nv set interface lo ip address 2006:20:20::24/128
nv set interface lo ip address 6000::24/128
nv set interface lo type loopback
nv set interface p0_if description 'alias p0_if to leaf-21 swp3'
nv set interface p0_if,p1_if,pf0hpf_if,pf0vf0_if,pf1hpf_if,pf1vf0_if type swp
nv set interface p0_if.100 vlan 100
nv set interface p0_if.100-101 base-interface p0_if
nv set interface p0_if.100-101 type sub
nv set interface p0_if.101 vlan 101
nv set interface p1_if description 'alias p1_if to leaf-22 swp3'
nv set interface pf0hpf_if bridge domain br_default access 111
nv set interface pf0hpf_if description 'alias pf0hpf_if to host-211 c1_0np0'
nv set interface pf0vf0_if bridge domain br_default access 112
nv set interface pf0vf0_if description 'alias pf0vf0_if to host-211 c1_0np0v0'
nv set interface pf1hpf_if bridge domain br_default access 213
nv set interface pf1hpf_if description 'alias pf1hpf_if to host-211 c1_1np1'
nv set interface pf1vf0_if bridge domain br_default access 214
nv set interface pf1vf0_if description 'alias pf1vf0_if to host-211 c1_1np0v0'
nv set interface vlan111 ip address 60.1.1.21/24
nv set interface vlan111 ip address 2060:1:1:1::21/64
nv set interface vlan111 ip vrr address 60.1.1.250/24
nv set interface vlan111 ip vrr address 2060:1:1:1::250/64
nv set interface vlan111 vlan 111
nv set interface vlan111,213 ip vrf vrf2
nv set interface vlan111-112,213-214 ip ipv4 forward on
nv set interface vlan111-112,213-214 ip ipv6 forward on
nv set interface vlan111-112,213-214 ip vrr enable on
nv set interface vlan111-112,213-214 ip vrr mac-address 00:00:5e:00:01:01
nv set interface vlan111-112,213-214 type svi
nv set interface vlan112 ip address 50.1.1.21/24
nv set interface vlan112 ip address 2050:1:1:1::21/64
nv set interface vlan112 ip vrr address 50.1.1.250/24
nv set interface vlan112 ip vrr address 2050:1:1:1::250/64
nv set interface vlan112 vlan 112
nv set interface vlan112,214 ip vrf vrf1
nv set interface vlan213 ip address 60.1.210.21/24
nv set interface vlan213 ip address 2060:1:1:210::21/64
nv set interface vlan213 ip vrr address 60.1.210.250/24
nv set interface vlan213 ip vrr address 2060:1:1:210::250/64
nv set interface vlan213 vlan 213
nv set interface vlan214 ip address 50.1.210.21/24
nv set interface vlan214 ip address 2050:1:1:210::21/64
nv set interface vlan214 ip vrr address 50.1.210.250/24
nv set interface vlan214 ip vrr address 2050:1:1:210::250/64
nv set interface vlan214 vlan 214
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan source address 6.0.0.24
nv set platform
nv set router bfd profile pf1 detect-multiplier 21
nv set router bfd profile pf1 min-rx-interval 100
nv set router bfd profile pf1 min-tx-interval 100
nv set router bfd profile pf2 detect-multiplier 22
nv set router bfd profile pf2 min-rx-interval 102
nv set router bfd profile pf2 min-tx-interval 102
nv set router bgp enable on
nv set router policy route-map ALLOW_LO rule 10 action permit
nv set router policy route-map ALLOW_LO rule 10 match interface lo
nv set router policy route-map ALLOW_LOBR rule 10 action permit
nv set router policy route-map ALLOW_LOBR rule 10 match interface lo
nv set router policy route-map ALLOW_LOBR rule 20 action permit
nv set router policy route-map ALLOW_LOBR rule 20 match interface br_default
nv set router vrr enable on
nv set system config snippet frr.conf 'log file /var/log/hbn/frr/bgpd.log'
nv set system global system-mac 00:01:00:00:1e:01
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast multipaths ebgp 16
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected route-map ALLOW_LOBR
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family ipv6-unicast multipaths ebgp 16
nv set vrf default router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast redistribute connected route-map ALLOW_LOBR
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp autonomous-system 650024
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor 6.0.0.23 bfd profile pf1
nv set vrf default router bgp neighbor 6.0.0.23 multihop-ttl 5
nv set vrf default router bgp neighbor 6.0.0.23 remote-as external
nv set vrf default router bgp neighbor 6.0.0.23 type numbered
nv set vrf default router bgp neighbor 6.0.0.26 peer-group EBGP_MHOP_V4
nv set vrf default router bgp neighbor 6.0.0.26 type numbered
```

```
nv set vrf default router bgp neighbor 6000::23 address-family ipv4-unicast enable off
nv set vrf default router bgp neighbor 6000::23 address-family ipv6-unicast enable on
nv set vrf default router bgp neighbor 6000::23 bfd profile pf1
nv set vrf default router bgp neighbor 6000::23 multihop-ttl 5
nv set vrf default router bgp neighbor 6000::23 remote-as external
nv set vrf default router bgp neighbor 6000::23 type numbered
nv set vrf default router bgp neighbor 6000::26 peer-group EBGP_MHOP_V6
nv set vrf default router bgp neighbor 6000::26 type numbered
nv set vrf default router bgp neighbor p0_if address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp neighbor p0_if address-family l2vpn-evpn enable on
nv set vrf default router bgp neighbor p0_if peer-group TOR_LEAF_SPINE
nv set vrf default router bgp neighbor p0_if remote-as external
nv set vrf default router bgp neighbor p0_if type unnumbered
nv set vrf default router bgp neighbor p0_if.100 address-family ipv6-unicast enable on
nv set vrf default router bgp neighbor p0_if.100 address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp neighbor p0_if.100 address-family l2vpn-evpn enable on
nv set vrf default router bgp neighbor p0_if.100 bfd profile pf1
nv set vrf default router bgp neighbor p0_if.100 peer-group TOR_LEAF_SPINE
nv set vrf default router bgp neighbor p0_if.100 remote-as external
nv set vrf default router bgp neighbor p0_if.100 type unnumbered
nv set vrf default router bgp neighbor p0_if.101 address-family ipv6-unicast enable on
nv set vrf default router bgp neighbor p0_if.101 address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp neighbor p0_if.101 address-family l2vpn-evpn enable on
nv set vrf default router bgp neighbor p0_if.101 peer-group TOR_LEAF_SPINE
nv set vrf default router bgp neighbor p0_if.101 remote-as external
nv set vrf default router bgp neighbor p0_if.101 type unnumbered
nv set vrf default router bgp neighbor p1_if address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp neighbor p1_if address-family l2vpn-evpn enable on
nv set vrf default router bgp neighbor p1_if peer-group TOR_LEAF_SPINE
nv set vrf default router bgp neighbor p1_if remote-as external
nv set vrf default router bgp neighbor p1_if type unnumbered
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp path-selection routerid-compare on
nv set vrf default router bgp peer-group EBGP_MHOP_V4 bfd profile pf2
nv set vrf default router bgp peer-group EBGP_MHOP_V4 multihop-ttl 5
nv set vrf default router bgp peer-group EBGP_MHOP_V4 remote-as external
nv set vrf default router bgp peer-group EBGP_MHOP_V6 address-family ipv4-unicast enable off
nv set vrf default router bgp peer-group EBGP_MHOP_V6 address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group EBGP_MHOP_V6 bfd profile pf2
nv set vrf default router bgp peer-group EBGP_MHOP_V6 multihop-ttl 5
nv set vrf default router bgp peer-group EBGP_MHOP_V6 remote-as external
nv set vrf default router bgp peer-group TOR_LEAF_SPINE address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group TOR_LEAF_SPINE address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group TOR_LEAF_SPINE bfd profile pf2
nv set vrf default router bgp router-id 6.0.0.24
nv set vrf default router rib ipv4 protocol static fib-filter fib2
nv set vrf vrf1 evpn enable on
nv set vrf vrf1 evpn vni 104001
nv set vrf vrf2 evpn enable on
nv set vrf vrf2 evpn vni 104002


CL:
nv set bridge domain
nv set interface eth0 ip address 192.168.0.15/24
nv set interface eth0 ip gateway 192.168.0.2
nv set interface eth0 type eth
nv set interface lo ip address 6.0.0.20/32
nv set interface lo ip address 2006:20:20::20/128
nv set interface lo ip address 6000::20/128
nv set interface lo type loopback
nv set interface swp2 description 'alias swp2 to spine-2 swp3'
nv set interface swp2-4 link auto-negotiate off
nv set interface swp2-4 link duplex full
nv set interface swp2-4 link speed 200G
nv set interface swp2-4 type swp
nv set interface swp2.100,swp3.100 vlan 100
nv set interface swp2.100-103 base-interface swp2
nv set interface swp2.100-103,swp3.100-101 type sub
nv set interface swp2.101,swp3.101 vlan 101
nv set interface swp2.102 vlan 102
nv set interface swp2.103 ip address 23.0.103.1/24
nv set interface swp2.103 router ospf enable on
nv set interface swp2.103 router ospf passive off
nv set interface swp2.103 vlan 103
nv set interface swp3 description 'alias swp3 to bfs-21 p0'
nv set interface swp3.100-101 base-interface swp3
nv set interface swp4 description 'alias swp4 to bfs-22 p0'
nv set platform
nv set router bgp enable on
nv set router ospf enable on
nv set router policy route-map ALLOW_LO rule 10 action permit
nv set router policy route-map ALLOW_LO rule 10 match interface lo
nv set router policy route-map ALLOW_LOBR rule 10 action permit
nv set router policy route-map ALLOW_LOBR rule 10 match interface lo
nv set router policy route-map ALLOW_LOBR rule 20 action permit
nv set router policy route-map ALLOW_LOBR rule 20 match interface br_default
nv set system aaa user cumulus hashed-password '*'
nv set system aaa user cumulus role system-admin
nv set system config auto-save state enabled
nv set system config snippet frr.conf 'log syslog
log file /var/log/hbn/frr/bgp_bfd.log'
nv set system control-plane acl acl-default-dos inbound
nv set system control-plane acl acl-default-whitelist inbound
nv set system hostname leaf-21
nv set system ssh-server permit-root-login enabled
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast multipaths ebgp 16
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected route-map ALLOW_LOBR
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family ipv6-unicast multipaths ebgp 16
nv set vrf default router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast redistribute connected route-map ALLOW_LOBR
nv set vrf default router bgp address-family l2vpn-evpn enable on
```

```
nv set vrf default router bgp autonomous-system 650020
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor swp2.100 address-family ipv6-unicast enable on
nv set vrf default router bgp neighbor swp2.100 address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp neighbor swp2.100 address-family l2vpn-evpn enable on
nv set vrf default router bgp neighbor swp2.100 peer-group TOR_LEAF_SPINE
nv set vrf default router bgp neighbor swp2.100 remote-as external
nv set vrf default router bgp neighbor swp2.100 type unnumbered
nv set vrf default router bgp neighbor swp3.100 address-family ipv6-unicast enable on
nv set vrf default router bgp neighbor swp3.100 address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp neighbor swp3.100 address-family l2vpn-evpn enable on
nv set vrf default router bgp neighbor swp3.100 bfd detect-multiplier 20
nv set vrf default router bgp neighbor swp3.100 bfd enable on
nv set vrf default router bgp neighbor swp3.100 bfd min-rx-interval 200
nv set vrf default router bgp neighbor swp3.100 bfd min-tx-interval 200
nv set vrf default router bgp neighbor swp3.100 peer-group TOR_LEAF_SPINE
nv set vrf default router bgp neighbor swp3.100 remote-as external
nv set vrf default router bgp neighbor swp3.100 type unnumbered
nv set vrf default router bgp neighbor swp3.101 address-family ipv6-unicast enable on
nv set vrf default router bgp neighbor swp3.101 address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp neighbor swp3.101 address-family l2vpn-evpn enable on
nv set vrf default router bgp neighbor swp3.101 bfd detect-multiplier 20
nv set vrf default router bgp neighbor swp3.101 bfd enable on
nv set vrf default router bgp neighbor swp3.101 bfd min-rx-interval 200
nv set vrf default router bgp neighbor swp3.101 bfd min-tx-interval 200
nv set vrf default router bgp neighbor swp3.101 peer-group TOR_LEAF_SPINE
nv set vrf default router bgp neighbor swp3.101 remote-as external
nv set vrf default router bgp neighbor swp3.101 type unnumbered
nv set vrf default router bgp neighbor swp4 address-family ipv6-unicast enable on
nv set vrf default router bgp neighbor swp4 address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp neighbor swp4 address-family l2vpn-evpn enable on
nv set vrf default router bgp neighbor swp4 bfd detect-multiplier 20
nv set vrf default router bgp neighbor swp4 bfd enable on
nv set vrf default router bgp neighbor swp4 bfd min-rx-interval 200
nv set vrf default router bgp neighbor swp4 bfd min-tx-interval 200
nv set vrf default router bgp neighbor swp4 peer-group TOR_LEAF_SPINE
nv set vrf default router bgp neighbor swp4 remote-as external
nv set vrf default router bgp neighbor swp4 type unnumbered
nv set vrf default router bgp path-selection multipath aspath-ignore on
nv set vrf default router bgp path-selection routerid-compare on
nv set vrf default router bgp peer-group TOR_LEAF_SPINE address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group TOR_LEAF_SPINE address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group TOR_LEAF_SPINE capabilities extended-nexthop on
nv set vrf default router bgp peer-group TOR_LEAF_SPINE timers connection-retry 10
nv set vrf default router bgp peer-group TOR_LEAF_SPINE timers hold 9
nv set vrf default router bgp peer-group TOR_LEAF_SPINE timers keepalive 3
nv set vrf default router bgp router-id 6.0.0.20
nv set vrf default router ospf area 0 network 23.0.103.0/24
nv set vrf default router ospf enable on
nv set vrf default router ospf router-id 6.0.0.20
nv set vrf vrf1 evpn enable on
nv set vrf vrf1 evpn vni 104001
nv set vrf vrf2 evpn enable on
nv set vrf vrf2 evpn vni 104002
```

8.9.5.3.6.4 Example topology with hbn, route server and BF Host using BFD



In this example HBN running on dpu11 peers with

- route servers using multihop BFD in default vrf for EVPN
- host11C using singlehop BFD in each tenant vrf
    - tenant1 uses svi for bgp/bfd session

- tenant2 uses subinterface to peer to host

8.9.5.3.6.5 Configuration

DPU1 HBN configuration

HBN (DPU1)

```
root@doca-hbn-service-bf3-s09-1-ipmi:/tmp# nv config show -o commands
nv set bridge domain br_default vlan 101 vni 10101
nv set evpn enable on
nv set evpn route-advertise svi-ip off
nv set interface lo ip address 6.0.0.9/32
nv set interface lo ip address 2001::9/128
nv set interface lo type loopback
nv set interface

p0_if,p1_if,pf0hpf_if,pf0vf0_if,pf0vf1_if,pf0vf2_if,pf0vf3_if,pf0vf4_if,pf0vf5_if,pf0vf6_if,pf0vf7_if,pf0vf8_if,pf0
vf9_if,pf1hpf_if,pf1vf0_if,pf1vf1_if type swp
nv set interface pf0vf0_if bridge domain br_default access 101
nv set interface pf0vf1_if.2 base-interface pf0vf1_if
nv set interface pf0vf1_if.2 ip address 22.10.0.9/16
nv set interface pf0vf1_if.2 ip address 2020:0:2:10::9/64
nv set interface pf0vf1_if.2 type sub
nv set interface pf0vf1_if.2 vlan 2
nv set interface pf0vf1_if.2,vlan3002 ip vrf tenant2
nv set interface vlan101 ip address 21.1.0.9/16
nv set interface vlan101 ip address 2020:0:1:1::9/64
nv set interface vlan101 ip vrr address 21.1.0.250/16
nv set interface vlan101 ip vrr address 2020:0:1:1::250/64
nv set interface vlan101 ip vrr enable on
nv set interface vlan101 ip vrr mac-address 00:00:01:00:00:65
nv set interface vlan101 vlan 101
nv set interface vlan101,3001 ip vrf tenant1
nv set interface vlan101,3001-3002 base-interface br_default
nv set interface vlan101,3001-3002 type svi
nv set interface vlan3001 vlan 3001
nv set interface vlan3002 vlan 3002
nv set nve vxlan arp-nd-suppress on
nv set nve vxlan enable on
nv set nve vxlan source address 6.0.0.9
nv set platform
nv set router bfd profile fabric-bfd-profile detect-multiplier 3
nv set router bfd profile fabric-bfd-profile min-rx-interval 300
nv set router bfd profile fabric-bfd-profile min-tx-interval 300
nv set router bfd profile fabric-bfd-profile minimum-ttl 1
nv set router bfd profile fabric-bfd-profile passive-mode off
nv set router bfd profile fabric-bfd-profile shutdown off
nv set router bfd profile hostT1-bfd-profile detect-multiplier 3
nv set router bfd profile hostT1-bfd-profile min-rx-interval 300
nv set router bfd profile hostT1-bfd-profile min-tx-interval 300
nv set router bfd profile hostT1-bfd-profile minimum-ttl 1
nv set router bfd profile hostT1-bfd-profile passive-mode off
nv set router bfd profile hostT1-bfd-profile shutdown off
nv set router bfd profile hostT2-bfd-profile detect-multiplier 3
nv set router bfd profile hostT2-bfd-profile min-rx-interval 300
nv set router bfd profile hostT2-bfd-profile min-tx-interval 300
nv set router bfd profile hostT2-bfd-profile minimum-ttl 1
nv set router bfd profile hostT2-bfd-profile passive-mode off
nv set router bfd profile hostT2-bfd-profile shutdown off
nv set router bfd profile ppp1
nv set router bfd profile rserver-bfd-profile detect-multiplier 3
nv set router bfd profile rserver-bfd-profile min-rx-interval 300
nv set router bfd profile rserver-bfd-profile min-tx-interval 300
nv set router bfd profile rserver-bfd-profile minimum-ttl 1
nv set router bfd profile rserver-bfd-profile passive-mode off
nv set router bfd profile rserver-bfd-profile shutdown off
nv set router bgp autonomous-system 65011
nv set router bgp enable on
nv set router bgp router-id 6.0.0.9
nv set router vrr enable on
nv set system config snippet
nv set system global
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor 6.0.0.5 peer-group rservers
nv set vrf default router bgp neighbor 6.0.0.5 type numbered
nv set vrf default router bgp neighbor p0_if peer-group fabric
nv set vrf default router bgp neighbor p0_if type unnumbered
nv set vrf default router bgp neighbor p1_if peer-group fabric
nv set vrf default router bgp neighbor p1_if type unnumbered
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group fabric bfd profile fabric-bfd-profile
nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp peer-group rservers address-family ipv4-unicast enable off
```

```
nv set vrf default router bgp peer-group rservers address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group rservers address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group rservers bfd profile rserver-bfd-profile
nv set vrf default router bgp peer-group rservers multihop-ttl 10
nv set vrf default router bgp peer-group rservers remote-as external
nv set vrf default router bgp peer-group rservers update-source lo
nv set vrf tenant1 evpn enable on
nv set vrf tenant1 evpn vni 30001
nv set vrf tenant1 loopback ip address 7.1.0.9/32
nv set vrf tenant1 loopback ip address 2007:0:1::9/64
nv set vrf tenant1 router bgp address-family ipv4-unicast enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant1 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf tenant1 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf tenant1 router bgp enable on
nv set vrf tenant1 router bgp neighbor 21.1.0.10 peer-group hostgroup
nv set vrf tenant1 router bgp neighbor 21.1.0.10 type numbered
nv set vrf tenant1 router bgp peer-group hostgroup address-family ipv4-unicast enable on
nv set vrf tenant1 router bgp peer-group hostgroup address-family ipv6-unicast enable on
nv set vrf tenant1 router bgp peer-group hostgroup bfd profile hostT1-bfd-profile
nv set vrf tenant1 router bgp peer-group hostgroup remote-as external
nv set vrf tenant1 router bgp router-id 6.0.0.9
nv set vrf tenant2 evpn enable on
nv set vrf tenant2 evpn vni 30002
nv set vrf tenant2 loopback ip address 7.2.0.9/32
nv set vrf tenant2 loopback ip address 2007:0:2::9/64
nv set vrf tenant2 router bgp address-family ipv4-unicast enable on
nv set vrf tenant2 router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf tenant2 router bgp address-family ipv4-unicast route-export to-evpn enable on
nv set vrf tenant2 router bgp address-family ipv6-unicast enable on
nv set vrf tenant2 router bgp address-family ipv6-unicast redistribute connected enable on
nv set vrf tenant2 router bgp address-family ipv6-unicast route-export to-evpn enable on
nv set vrf tenant2 router bgp enable on
nv set vrf tenant2 router bgp neighbor 22.10.0.10 peer-group hostgroup
nv set vrf tenant2 router bgp neighbor 22.10.0.10 type numbered
nv set vrf tenant2 router bgp peer-group hostgroup address-family ipv4-unicast enable on
nv set vrf tenant2 router bgp peer-group hostgroup address-family ipv6-unicast enable on
nv set vrf tenant2 router bgp peer-group hostgroup bfd profile hostT2-bfd-profile
nv set vrf tenant2 router bgp peer-group hostgroup remote-as external
nv set vrf tenant2 router bgp router-id 6.0.0.9
nv set vrf tenant2 table auto
root@doca-hbn-service-bf3-s09-1-ipmi:/tmp#
```

Host11C configuration

Host is running frr that has support for BFD:

Host Requirements

```
root@bf3-s09:/home/cumulus# dpkg -l | grep frr
ii  frr                                  8.4.4-1.1ubuntu6.3                      amd64        FRRouting
suite of internet protocols (BGP, OSPF, IS-IS, ...)
ii  frr-pythontools                      8.4.4-1.1ubuntu6.3                      all          FRRouting
suite - Python tools
root@bf3-s09:/home/cumulus#
root@bf3-s09:/home/cumulus# cat /etc/os-release
PRETTY_NAME="Ubuntu 24.04.2 LTS"
NAME="Ubuntu"
VERSION_ID="24.04"
VERSION="24.04.2 LTS (Noble Numbat)"
VERSION_CODENAME=noble
ID=ubuntu
ID_LIKE=debian
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
UBUNTU_CODENAME=noble
LOGO=ubuntu-logo
root@bf3-s09:/home/cumulus#
```

Host FRR configuration

```
bf3-s09# sh run
Building configuration...

Current configuration:
!
frr version 8.4.4
frr defaults datacenter
hostname bf3-s09
```

```
log syslog informational
no ip forwarding
no ipv6 forwarding
service integrated-vtysh-config
!
router bgp 1000010
exit
!
router bgp 1000010 vrf tenant1_101
 neighbor 21.1.0.9 remote-as external
 neighbor 21.1.0.9 bfd
 neighbor 21.1.0.9 bfd profile host-bfd-profile
 !
 address-family ipv4 unicast
  redistribute connected
 exit-address-family
 !
 address-family ipv6 unicast
  redistribute connected
  neighbor 21.1.0.9 activate
 exit-address-family
exit
!
router bgp 1000010 vrf tenant2
 neighbor 22.10.0.9 remote-as external
 neighbor 22.10.0.9 bfd
 neighbor 22.10.0.9 bfd profile host-bfd-profile
 !
 address-family ipv4 unicast
  redistribute connected
 exit-address-family
 !
 address-family ipv6 unicast
  redistribute connected
  neighbor 22.10.0.9 activate
 exit-address-family
exit
!
bfd
 profile host-bfd-profile
  transmit-interval 301
  receive-interval 301
  minimum-ttl 1
 exit
 !
exit
!
end
bf3-s09#
```

Host11C vrf and interface configuration

```
root@bf3-s09:/home/cumulus# ip vrf list
Name              Table
----------------------
tenant1_101       1001
tenant2           1002
root@bf3-s09:/home/cumulus# ifquery -a
auto lo
iface lo inet loopback
    address 10.0.0.10/32
    address 2000::10/128

auto tenant1_101
iface tenant1_101
    vrf-table auto

auto tenant2
iface tenant2
    vrf-table auto

auto c1_0np0
iface c1_0np0

auto c1_0v0
iface c1_0v0
    address 21.1.0.10/16
    address 2020:0:1:1::10/64
    gateway 21.1.0.9
    gateway 2020:0:1:1::9
    vrf tenant1_101
    hwaddress 00:03:00:0a:0a:00

auto c1_0v1
iface c1_0v1

auto c1_0v2
iface c1_0v2

auto c1_0v3
iface c1_0v3

auto c1_0v4
iface c1_0v4

auto c1_0v5
iface c1_0v5
```

```
auto c1_0v6
iface c1_0v6

auto c1_0v7
iface c1_0v7

auto c1_0v8
iface c1_0v8

auto c1_0v9
iface c1_0v9

auto c1_1np1
iface c1_1np1

auto c1_1v0
iface c1_1v0

auto c1_1v1
iface c1_1v1

auto c1_0v1.2
iface c1_0v1.2
    address 22.10.0.10/16
    address 2020:0:2:10::10/64
    gateway 22.10.0.9
    gateway 2020:0:2:10::9
    vrf tenant2

root@bf3-s09:/home/cumulus#
```

show commands

BFD peers

```
root@doca-hbn-service-bf3-s09-1-ipmi:/tmp# nv show  vrf tenant2 router bfd peer

MHop - Multihop, Local - Local, Peer - Peer, Interface - Interface, State -
State, Passive - Passive Mode, Time - Up/Down Time, Type - Config Type

LocalId     MHop   Local      Peer        Interface    State  Passive  Time        Type
----------  -----  ---------  ----------  -----------  -----  -------  ----------  -------
1672363941  False  22.10.0.9  22.10.0.10  pf0vf1_if.2  up     False    1:15:55:07  dynamic
root@doca-hbn-service-bf3-s09-1-ipmi:/tmp# nv show  vrf tenant1 router bfd peer

MHop - Multihop, Local - Local, Peer - Peer, Interface - Interface, State -
State, Passive - Passive Mode, Time - Up/Down Time, Type - Config Type

LocalId     MHop   Local     Peer       Interface  State  Passive  Time        Type
----------  -----  --------  ---------  ---------  -----  -------  ----------  -------
3647797911  False  21.1.0.9  21.1.0.10  vlan101    up     False    1:15:55:17  dynamic
root@doca-hbn-service-bf3-s09-1-ipmi:/tmp# nv show  vrf default router bfd peer

MHop - Multihop, Local - Local, Peer - Peer, Interface - Interface, State -
State, Passive - Passive Mode, Time - Up/Down Time, Type - Config Type

LocalId     MHop   Local                      Peer                Interface  State  Passive  Time        Type
----------  -----  -------------------------  ------------------  ---------  -----  -------  ----------  -------
1241890423  False  fe80::5aa2:e1ff:fe62:f81e  fe80::202:ff:fe00:a  p0_if      up     False    1:15:55:27  dynamic
1280361828  False  fe80::5aa2:e1ff:fe62:f81f  fe80::202:ff:fe00:d  p1_if      up     False    1:15:55:27  dynamic
1936332380  True   6.0.0.9                    6.0.0.5                         up     False    1:15:55:26  dynamic
root@doca-hbn-service-bf3-s09-1-ipmi:/tmp#
```

Route Server SN1 configuration

In this example route server is running CumulusLinux 5.12

RS SN1 configuration

```
root@sn1:mgmt:/var/home/cumulus# nv config show -o commands
nv set bridge domain
nv set interface eth0 ip address 192.168.0.15/24
nv set interface eth0 ip gateway 192.168.0.2
nv set interface eth0 type eth
```

```
nv set interface lo ip address 6.0.0.5/32
nv set interface lo ip address 2001::5/128
nv set interface lo type loopback
nv set interface swp1 type swp
nv set platform
nv set router bgp autonomous-system 65505
nv set router bgp enable on
nv set router bgp router-id 6.0.0.5
nv set system aaa user cumulus hashed-password '*'
nv set system aaa user cumulus role system-admin
nv set system config auto-save state enabled
nv set system config snippet
nv set system control-plane acl acl-default-dos inbound
nv set system control-plane acl acl-default-whitelist inbound
nv set system hostname sn1
nv set system ssh-server permit-root-login enabled
nv set vrf default router bgp address-family ipv4-unicast enable on
nv set vrf default router bgp address-family ipv4-unicast redistribute connected enable on
nv set vrf default router bgp address-family ipv6-unicast enable on
nv set vrf default router bgp address-family l2vpn-evpn enable on
nv set vrf default router bgp enable on
nv set vrf default router bgp neighbor 6.0.0.4 peer-group rclients
nv set vrf default router bgp neighbor 6.0.0.4 type numbered
nv set vrf default router bgp neighbor 6.0.0.9 peer-group rclients
nv set vrf default router bgp neighbor 6.0.0.9 type numbered
nv set vrf default router bgp neighbor swp1 peer-group rcsuper
nv set vrf default router bgp neighbor swp1 type unnumbered
nv set vrf default router bgp peer-group fabric address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group fabric address-family ipv6-unicast enable on
nv set vrf default router bgp peer-group fabric bfd enable on
nv set vrf default router bgp peer-group fabric remote-as external
nv set vrf default router bgp peer-group rclients address-family ipv4-unicast attribute-mod nexthop off
nv set vrf default router bgp peer-group rclients address-family ipv4-unicast enable off
nv set vrf default router bgp peer-group rclients address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group rclients address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group rclients bfd enable on
nv set vrf default router bgp peer-group rclients multihop-ttl 10
nv set vrf default router bgp peer-group rclients remote-as external
nv set vrf default router bgp peer-group rclients update-source lo
nv set vrf default router bgp peer-group rcsuper address-family ipv4-unicast enable on
nv set vrf default router bgp peer-group rcsuper address-family l2vpn-evpn add-path-tx off
nv set vrf default router bgp peer-group rcsuper address-family l2vpn-evpn enable on
nv set vrf default router bgp peer-group rcsuper bfd enable on
nv set vrf default router bgp peer-group rcsuper remote-as external
root@sn1:mgmt:/var/home/cumulus#
```

Route Server SN1 - bgp and bfd sessions

```
root@sn1:mgmt:/var/home/cumulus# nv show vrf
Name     Table  Summary
-------  -----  ----------------------
default  254    IP Address: 127.0.0.1/8
                IP Address: 2001::5/128
                IP Address:  6.0.0.5/32
                IP Address:     ::1/128
mgmt     1001   IP Address: 127.0.0.1/8
                IP Address: 127.0.1.1/8
                IP Address:     ::1/128
root@sn1:mgmt:/var/home/cumulus# nv show vrf default router bgp neighbor

AS - Remote Autonomous System, PeerEstablishedTime - Peer established time in
UTC format, UpTime - Last connection reset time in days,hours:min:sec, Afi-Safi
- Address family, PfxSent - Transmitted prefix counter, PfxRcvd - Recieved
prefix counter

Neighbor  AS     State        PeerEstablishedTime   UpTime    MsgRcvd  MsgSent  Afi-Safi      PfxSent  PfxRcvd
--------  -----  -----------  --------------------  --------  -------  -------  ------------  -------  -------
6.0.0.4   65012  established  2025-04-23T04:52:24Z  14:38:47  46720    46727    l2vpn-evpn    32       10
6.0.0.9   65011  established  2025-04-23T04:58:14Z  14:33:33  46729    46759    l2vpn-evpn    32       13
swp1      65300  established  2025-04-23T04:41:41Z  14:49:12  46743    46745    ipv4-unicast  6        5
                                                                                 l2vpn-evpn    32       9
root@sn1:mgmt:/var/home/cumulus#
root@sn1:mgmt:/var/home/cumulus# ptmctl -b
------------------------------------------------------------------------------
port  peer                  state  local                type       diag  vrf

------------------------------------------------------------------------------
swp1  fe80::202:ff:fe00:11  Up     fe80::202:ff:fe00:8  singlehop  N/A   N/A
N/A   6.0.0.4               Up     6.0.0.5              multihop   N/A   N/A
N/A   6.0.0.9               Up     6.0.0.5              multihop   N/A   N/A
root@sn1:mgmt:/var/home/cumulus#
```

8.9.5.3.7 Control plane policing (CoPP)

Control plane policing is a mechanism to guard the control plane. This is achieved by policing the traffic coming towards the control plane. Control plane constitutes of routers, management protocol handlers and system services. CoPP ensures only necessary traffic reaches to the control plane without exceeding predefined packet rate. CoPP uses filtering and rate limiting traffic coming towards control plane. This prevents overloading of CPU and other system resources with excessive packets i.e. Denial of service (DoS) attacks. The excess traffic is rate limited and dropped according to predefined or user defined configurations. CoPP configuration is applied globally to the control plane.

We have following separate policers for each protocol or group of packets. Unit of each value is in packet per second.

Sr. No.	Policer Name	Protocol	Default State	Default Rate	Default Burst
1.	arp	ARP, RARP	Enabled	800	800
2.	bgp	BGP	Enabled	2000	2000
3.	bfd	BFD	Enabled	2000	2000
4.	icmp	ICMP, ICMPv6	Enabled	1000	1000
5.	icmp6-neigh	IPv6 NDP	Enabled	1000	1000
6.	dhcp	DHCPv4, DHCPv6	Enabled	2000	2000
7.	ip2me	All other IPv4 or IPv6 traffic destined for HBN	Enabled	1000	1000
8.	catch-all	All other traffic not matched by above rules	Enabled	100	100

We enabled the CoPP configuration on HBN by default with predefined values.

8.9.5.3.7.1 CoPP configuration

Enabling a policer

In this example we are enabling a bgp policer.

```
nv set system control-plane policer bgp enable on
```

Updating a policer

In this example we are changing the rate and burst of the bgp policer.

```
nv set system control-plane policer bgp rate 3000
nv set system control-plane policer bgp burst 3000
```

Disabling a policer

```
nv set system control-plane policer bgp enable off
```

8.9.5.3.7.2 CoPP show commands

CoPP summary

Following CLI shows the summary of all the policers. This includes the configurations and respective statistics

```
nv show system control-plane policer
```

Corresponding sample output

```
Policer      State  Policer Rate  Policer Burst  To CPU Pkts  To CPU Bytes  Violated Packets  Violated Bytes
-----------  -----  ------------  -------------  -----------  ------------  ----------------  --------------
arp          on     800           800            1327         133663        0                 0
bfd          on     2000          2000           0            0             0                 0
bgp          on     2000          2000           167470       16689652      0                 0
catch-all    on     100           100            0            0             0                 0
dhcp         on     2000          2000           0            0             0                 0
icmp         on     1000          1000           0            0             0                 0
icmp6-neigh  on     1000          1000           426605       36641706      0                 0
ip2me        on     1000          1000           0            0             0                 0
```

Show CLI for a specific policer

Following CLI shows details of individual policer. In this specific example bgp policer details will be shown

```
nv show system control-plane policer bgp
```

Corresponding sample output

```
                 operational  applied
---------------  -----------  -------
state            on           on
burst            3000         3000
rate             3000         3000
statistics
  to-cpu-pkts    167686
  to-cpu-bytes   16710895
  violated-pkts  0
  violated-bytes 0
```

8.9.5.3.8 DHCP Relay on HBN

DHCP is a client server protocol that automatically provides IP hosts with IP addresses and other related configuration information. A DHCP relay (agent) is a host that forwards DHCP packets between clients and servers. DHCP relays forward requests and replies between clients and servers that are not on the same physical subnet.

DHCP relay can be configured using either flat file (supervisord configuration) or through NVUE.

8.9.5.3.8.1 Configuration

HBN is a non-systemd based container. Therefore, the DHCP relay must be configured as explained in the following subsections.

Flat File Configuration (Supervisord)

The HBN initialization script installs default configuration files on BlueField in **/var/lib/hbn/etc/supervisor/conf.d/**. BlueField directory is mounted to **/etc/supervisor/conf.d** which achieves configuration persistence.

By default, DHCP relay is disabled. Default configuration applies to one instance of DHCPv4 relay and DHCPv6 relay in the default VRF.

NVUE Configuration

The user can use NVUE to configure and maintain DHCPv4 and DHCPv6 relays with CLI and REST API. NVUE generates all the required configurations and maintains the relay service.

DHCPv4 Relay Configuration

NVUE Example

The following configuration starts a relay service which listens for the DHCP messages on **p0_if**, **p1_if**, and **vlan482** and relays the requests to DHCP server 10.89.0.1 with **gateway-interface** as **lo**.

```
nv set service dhcp-relay default gateway-interface lo
nv set service dhcp-relay default interface p0_if
nv set service dhcp-relay default interface p1_if
nv set service dhcp-relay default interface vlan482 downstream
nv set service dhcp-relay default server 10.89.0.1
```

Flat Files Example

```
[program: isc-dhcp-relay-default]
command = /usr/sbin/dhcrelay --nl -d -i p0_if -i p1_if -id vlan482 -U lo 10.89.0.1
autostart = true
autorestart = unexpected
startsecs = 3
startretries = 3
exitcodes = 0
stopsignal = TERM
stopwaitsecs = 3
```

Where:

Option	Description
-i	Network interface to listen on for requests and replies
-iu	Upstream network interface
-id	Downstream network interface
-U [address]%%ifname	Gateway IP address interface. Use **%%** for **IP%%ifname**. **%** is used as an escape character.
--loglevel-debug	Debug logging. Location: **/var/log/syslog**.
-a	Append an agent option field to each request before forwarding it to the server with default values for **circuit-id** and **remote-id**
-r remote-id	Set a custom remote ID string (max of 255 chars). To use this option, you must also enable the **-a** option.

Option	Description
--use-pif-circuit-id	Set the underlying physical interface which receives the packet as the **circuit-id**. To use this option you must also enable the **-a** option.

DHCPv4 Relay Option 82

NVUE Example

The following NVUE command is used to enable option 82 insertion in DHCP packets with default values:

```
nv set service dhcp-relay default agent enable on
```

To provide a custom **remote-id** (e.g., host10) using NVUE:

```
nv set service dhcp-relay default agent remote-id host10
```

To use the underlying physical interface on which the request is received as **circuit-id** using NVUE:

```
nv set service dhcp-relay default agent use-pif-circuit-id enable on
```

Flat Files Example

```
[program: isc-dhcp-relay-default]
command = /usr/sbin/dhcrelay --nl -d -i p0_if -i p1_if -id vlan482 -U lo -a --use-pif-circuit-id -r host10
10.89.0.1
autostart = true
autorestart = unexpected
startsecs = 3
startretries = 3
exitcodes = 0
stopsignal = TERM
stopwaitsecs = 3
```

DHCPv6 Relay Configuration

NVUE Example

The following NVUE command starts the DHCPv6 Relay service which listens for DHCPv6 requests on **vlan482** and sends relayed DHCPv6 requests towards **p0_if** and **p1_if**.

```
nv set service dhcp-relay6 default interface downstream vlan482
nv set service dhcp-relay6 default interface upstream p0_if
nv set service dhcp-relay6 default interface upstream p1_if
```

Flat Files Example

```
[program: isc-dhcp-relay6-default]
command = /usr/sbin/dhcrelay --nl -6 -d -l vlan482 -u p0_if -u p1_if
autostart = true
autorestart = unexpected
startsecs = 3
startretries = 3
exitcodes = 0
stopsignal = TERM
stopwaitsecs = 3
```

Where:

Option	Description
`-l [address]%%ifname[#index]`	Downstream interface. Use `%%` for `IP%%ifname`. `%` is used as escape character.
`-u [address]%%ifname`	Upstream interface. Use `%%` for `IP%%ifname`. `%` is used as escape character.
`-6`	IPv6
`--loglevel-debug`	Debug logging located at `/var/log/syslog`

8.9.5.3.8.2 DHCP Relay and VRF Considerations

DHCP relay can be spawned inside a VRF context to handle the DHCP requests in that VRF. There can only be 1 instance each of DHCPv4 relay and DHCPv6 relay per VRF. To achieve that, the user can follow these guidelines:

- DHCPv4 on default VRF:

```
/usr/sbin/dhcrelay --nl -i <interface> -U [address]%%<interface> <server_ip>
```

- DHCPv4 on VRF:

```
/usr/sbin/ip vrf exec <vrf> /usr/sbin/dhcrelay --nl -i <interface> -U [address]%%<interface> <server_ip>
```

- DHCPv6 on default VRF:

```
/usr/sbin/dhcrelay --nl -6 -l <interface> -u <interface>
```

- DHCPv6 on VRF:

```
/usr/sbin/ip vrf exec <vrf> /usr/sbin/dhcrelay --nl -6 -l <interface> -u <interface>
```

8.9.6 HBN Service Troubleshooting

8.9.6.1 HBN Container Stuck in init-sfs

The HBN container starts as `init-sfs` and should transition to `doca-hbn` within 2 minutes as can be seen using `crictl ps`. But sometimes it may remain as `init-sfs`.

This can happen if interface `p0_if` is missing. Run the command `ip -br link show dev p0_if` in BlueField and inside the container to check if `p0_if` is present or not. If its missing, make sure the firmware is upgraded to the latest version. Perform BlueField system-level reset for the new firmware to take effect.

8.9.6.2 Host-side PF/VF Down After BlueField Reboot

In general, the host can use any interface manager to manage host interfaces belonging to BlueField. When the host uses an interface manager other than Netplan or NetworkManager, some ports may remain down after BlueField reboot.

Apply the following workaround if interfaces stay down:

1. Restart openibd:

```
systemctl restart openibd
```

2. Recreate SR-IOV interfaces if they are needed.
3. Replay interface config. For example:
   - If using ifupdown2:

```
ifreload -a
```

   - If using Netplan:

```
netplan apply
```

8.9.6.3 BGP Session not Establishing

One of the main causes of a BGP session not getting established is a mismatch in MTU configuration. Make sure the MTU on all interfaces is the same. For example, if BGP is failing on **p0**, check and verify that there is a matching MTU value for **p0**, **p0_if_r**, **p0_if**, and the remote peer of **p0**.

8.9.6.4 Generating Support Information

The HBN container image can be collected from **/etc/image-version** using the **hbn-support** command inside container:

```
root@bf2:/tmp# hbn-support
Please send /var/support/hbn_support_doca-hbn-service-bf2-s15-1-ipmi_20240820_211214.txz to Cumulus support.
```

The generated dump would be available under **/var/support** in the HBN container and should contain any process core dump and log files. The generated cores can be found under **/var/support/core** and collected by **hbn-support**. The **/var/support** directory is also mounted on the BlueField Arm side at **/var/lib/hbn/var/support**.

For BlueField, the BFB version can be checked from **/etc/mlnx-release**.

The firmware version can be collect from **mlxfwmanager**.

BlueField support dump can be collect using the **sos** command:

```
root@bf2:/tmp/#sos report -a --all-logs --batch
```

Example output:

```
sos report (version 4.8.0)

This command will collect system configuration and diagnostic
information from this Ubuntu system.
...
...
  Finished running plugins

Creating compressed archive...

Your sos report has been generated and saved in:
        /tmp/sosreport-bf2-s15-1-ipmi-2024-08-20-cpdvegw.tar.xz

 Size   19.37MiB
 Owner  root
 sha256 0890a855623a1a2dd5089c9cd6d57d81e71f3805ac06c2d9fc0dab556ccd5ffc

Please send this file to your support representative.
```

8.9.6.5 SFC Troubleshooting

To troubleshoot flows going through SFC interfaces, the first step is to disable the **nl2doca** service in the HBN container:

```
root@bf2:/tmp# supervisorctl stop nl2doca
nl2doca: stopped
```

Stopping **nl2doca** effectively stops hardware offloading and switches to software forwarding. All packets would appear on **tcpdump** capture on BlueField interfaces.

tcpdump can be performed on SF interfaces as well as VLAN, VXLAN, and uplinks to determine where a packet gets dropped or which flow a packet is taking.

8.9.6.6 General nl2doca Troubleshooting

The following steps can be used to make sure the nl2doca daemon is up and running:

1. Make sure there are no errors in the nl2doca log file at **/var/log/hbn/nl2docad.log** .
2. To check the status of the nl2doca daemon under supervisor, run:

   ```
   supervisorctl status nl2doca
   ```

3. Use **ps** to check that the actual nl2doca process is running:

   ```
   ps -eaf | grep nl2doca
   root          18       1  0 06:31 ?        00:00:00 /bin/bash /usr/bin/nl2doca-docker-start
   root        1437      18  0 06:31 ?        00:05:49 /usr/sbin/nl2docad
   ```

4. The core file should be in **/var/support/core/** .
5. Check if the **/cumulus/nl2docad/run/stats/punt** is accessible. Otherwise, nl2doca may be stuck and should be restarted:

   ```
   supervisorctl restart nl2doca
   ```

8.9.6.7 nl2doca Offload Troubleshooting

If a certain traffic flow does not work as expected, disable nl2doca (i.e., disable hardware offloading):

```
supervisorctl stop nl2doca
```

With hardware offloading disabled, you can confirm it is an offloading issue if the traffic starts working. If it is not an offloading issue, use **tcpdump** on various interfaces to see where the packet gets dropped.

Offloaded entries can be checked in following files, which contain the programming status of every IP prefix and MAC address known to system.

- Bridge entries are available in the file **/cumulus/nl2docad/run/software-tables/17**. It includes all the MAC addresses in the system including local and remote MAC addresses.
  Example format:

  ```
  - flow-entry: 0xaaab0cef4190
        flow-pattern:
          fid: 112
          dst mac: 00:00:5e:00:01:01
        flow-actions:
          SET VRF: 2
          OUTPUT-PD-PORT: 20(TO_RTR_INTF)
          STATS:
            pkts: 1719
            bytes: 191286
  ```

- Router entries are available in the file **/cumulus/nl2docad/run/software-tables/18**. It includes all the IP prefixes known to the system.
  Example format for Entry with ECMP:

  ```
  Entry with ECMP:
  - flow-entry: 0xaaaada723700
    flow-pattern:
       IPV6: LPM
       VRF: 0
       destination-ip: ::/0
    flow-actions :
       ECMP: 2
       STATS:
          pkts: 0
          bytes: 0

  Entry without ECMP: - flow-entry: 0xaaaada7e1400
      flow-pattern:
         IPV4: LPM
         VRF: 0
         destination-ip: 60.1.0.93/32
      flow-actions :
          SET FID: 200
          SMAC: 00:04:4b:a7:88:00
          DMAC: 00:03:00:08:00:12
          OUTPUT-PD-PORT: 19(TO_BR_INTF)
     STATS:
         pkts: 0
         bytes: 0
  ```

- ECMP entries are available in the file **/cumulus/nl2docad/run/software-tables/19**. It includes all the next hops in the system.
  Example format:

  ```
  - ECMP: 2
    ref-count: 2
    num-next-hops: 2
    entries:
    - { index: 0, fid: 4100, src mac: 'b8:ce:f6:99:49:6a', dst mac: '00:02:00:00:00:0a' }
    - { index: 1, fid: 4101, src mac: 'b8:ce:f6:99:49:6b', dst mac: '00:02:00:00:00:0e' }
  ```

To check counters for packets going to the kernel, run:

```
cat /cumulus/nl2docad/run/stats/punt
PUNT miss pkts:3154 bytes:312326
PUNT miss drop pkts:0 bytes:0
PUNT control pkts:31493 bytes:2853186
PUNT control drop pkts:0 bytes:0
ACL PUNT pkts:68 bytes:7364
ACL drop pkts:0 bytes:0
```

For a specific type of packet flow, programming can be referenced in block specific files. The typical flow is as follows:

For example, to check L2 EVPN ENCAP flows for remote MAC **8a:88:d0:b1:92:b1** on port **pf0vf0_if**, the basic offload flow should look as follows: RxPort (**pf0vf0_if**) -> BR (Overlay) -> RTR (Underlay) -> BR (Underlay) -> TxPort (one of the uplink **p0_if** or **p1_if** based on ECMP hash).

Step-by-step procedure:

1. Navigate to the interface file **/cumulus/nl2docad/run/software-tables/20**.
2. Check for the RxPort (**pf0vf0_if**):

   ```
   Interface: pf0vf0_if
       PD PORT: 6
       HW PORT: 16
       NETDEV PORT: 11
       Bridge-id: 61
       Untagged FID: 112
   ```

   FID 112 is given to the receive port.
3. Check the bridge table file **/cumulus/nl2docad/run/software-tables/17** with destination MAC **8a:88:d0:b1:92:b1** and FID 112:

   ```
   flow-pattern:
         fid: 112
           dst mac: 8a:88:d0:b1:92:b1
         flow-actions:
           VXLAN ENCAP:
             ENCAP dst ip: 6.0.0.26
             ENCAP vni id: 1000112
           SET VRF: 0
           OUTPUT-PD-PORT: 20(TO_RTR_INTF)
           STATS:
             pkts: 100
             bytes: 10200
   ```
4. Check the router table file **/cumulus/nl2docad/run/software-tables/18** with destination IP **6.0.0.26** and VRF 0:

   ```
   flow-pattern:
           IPV4: LPM
           VRF: 0
           ip dst: 6.0.0.26/32
         flow-actions :
           ECMP: 1
           OUTPUT PD PORT: 2(TO_BR_INTF)
           STATS:
             pkts: 300
             bytes: 44400
   ```
5. Check the ECMP table file **/cumulus/nl2docad/run/software-tables/19** with ECMP 1:

   ```
   - ECMP: 1
         ref-count: 7
         num-next-hops: 2
         entries:
           - { index: 0, fid: 4100, src mac: 'b8:ce:f6:99:49:6a', dst mac: '00:02:00:00:00:2f' }
   ```

```
        - { index: 1, fid: 4115, src mac: 'b8:ce:f6:99:49:6b', dst mac: '00:02:00:00:00:33' }
```

6. The ECMP hash calculation picks one of these paths for next-hop rewrite. Check bridge table file for them (**fid=4100, dst mac: 00:02:00:00:00:2f** or **fid=4115, dst mac: 00:02:00:00:00:33**):

```
flow-pattern:
        fid: 4100
        dst mac: 00:02:00:00:00:2f
flow-actions:
    OUTPUT-PD-PORT: 36(p0_if)
    STATS:
       pkts: 1099
       bytes: 162652
```

This will show the packet going out on the uplink.

8.9.6.8 NVUE Troubleshooting

To check the status of the NVUE daemon, run:

```
supervisorctl status nvued
```

To restart the NVUE daemon, run:

```
supervisorctl restart nvued
```

8.10 DOCA Management Service Guide

This guide explains how to use the DOCA Management Service (DMS) with the NVIDIA® BlueField® networking platform or NVIDIA® ConnectX® SmartNICs.

> Existing DMS users: If you have used DMS before and are looking for the legacy, OpenConfig-based tool, see legacy dms tool guide.

8.10.1 1. Introduction

Configuring a BlueField DPU or ConnectX NIC requires a different tool for each task: mlxconfig for FW parameters, devlink for E-switch settings, and mlnx_qos for traffic priorities. Each tool has its own syntax and quirks. Managing hundreds of devices this way is painful.

DOCA Management Service (DMS) is a centralized solution for configuring and operating BlueField and ConnectX devices.
It replaces the fragmented toolset with a single CLI (dms-cli) and a unified interface: one tool, one syntax, every feature, and every device.

8.10.1.1 1.1. Goals and Use Cases

DMS is built around two goals:

- Unified interface: One tool and one syntax replace the fragmented toolset, regardless of the feature or operation.

- Works at scale: Every command returns structured JSON, making it easy to script and automate across a fleet of devices.

What can you do with DMS?

- Install firmware or a BFB image: Update a ConnectX NIC or provision a BlueField DPU.
- Configure network settings: set port speed, enable or disable a port, configure bonding.
- Tune RoCE and QoS parameters: congestion control, priorities, and adaptive routing.
- Reset a device: factory reset or reboot.
- Automate at scale: script consistent configuration across a fleet of devices.

8.10.1.2 1.2. Key Concepts

8.10.1.2.1 1.2.1. Yang - the configuration map

DMS describes everything a device can do using YANG model files.
Think of YANG as a structured configuration map of your device. It is a tree-based data modeling language (XML syntax). Each feature is a branch, and each parameter is a leaf. The model is expressed in a human-readable format, making it easy to browse and understand what a device supports.
For example, the DPU mode configuration model looks like this:

```
container mode {
  leaf operating-mode {
    type enumeration {
      enum nic;    // Standard NIC, ARM cores disabled
      enum dpu;    // DPU mode, ARM cores enabled
    }
  }
}
```

Every parameter DMS supports is defined this way - the YANG model is the single source of truth for what you can read, write, or trigger on a device.

8.10.1.2.2 1.2.2. XPath - your address in the map

Each parameter in the YANG tree has a unique path: an XPath.
It directly reflects the structure of the model: start from a feature area and navigate down to the specific parameter.
For example, the mode model above translates directly to: **/nvidia/mode/operating-mode** .

Lists: When a feature has multiple instances - for example, several CC algorithm slots or multiple ports, the YANG model represents them as a list.
The key name (id, name, etc.) is defined by the YANG model for that list.
You select a specific entry by adding a key in brackets:

```
/nvidia/cc/slot[id=0]/config/enabled    # slot 0
/nvidia/cc/slot[id=1]/config/enabled    # slot 1
```

To explore all the available paths on your device, use the command

```
/opt/mellanox/doca/services/dms/dms-cli --tree /nvidia
```

8.10.1.2.3 1.2.3. Target - which device you are talking to

Use -t / --target to specify the target device by its PCIe BDF (Bus:Device.Function) address. When managing a VF or SF, append the representor suffix:

```
/opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 /nvidia/mode/operating-mode        # target a specific
device by BDF
/opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0,pf0vf5 /nvidia/icm/quota          # target VF5 on PF0
/opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0,pf0sf3 /nvidia/icm/quota          # target SF3 on PF0
```

The representor suffix format is pf<N>vf<M> for virtual functions and pf<N>sf<M> for scalable functions, where N is the PF index and M is the VF/SF index.

Scope	Target
System-level	not required
Device / Port	pci/<BDF>
VF	pci/<BDF>,pf<N>vf<M>
SF	pci/<BDF>,pf<N>sf<M>

8.10.2 2. System Components

8.10.2.1 2.1. dms-cli

dms-cli is the single entry point for all DMS operations. You run it directly on the host or Arm side - no server or daemon is required.
When you run a command, it:

1. Reads the YANG model to validate the command and determine the operation type.
2. Runs the appropriate operation on the device HW.

Examples:

```
# Read the current DPU operating mode
/opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 /nvidia/mode/operating-mode

# Switch to DPU mode
/opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 /nvidia/mode/operating-mode dpu

# Reset the device to factory defaults
/opt/mellanox/doca/services/dms/dms-cli /nvidia/reset/factory-reset
```

8.10.2.2 2.2. Yang Models

The configuration map that dms-cli reads to understand which paths exist, what types they accept, and which operations they support (Get, Set, or Action).

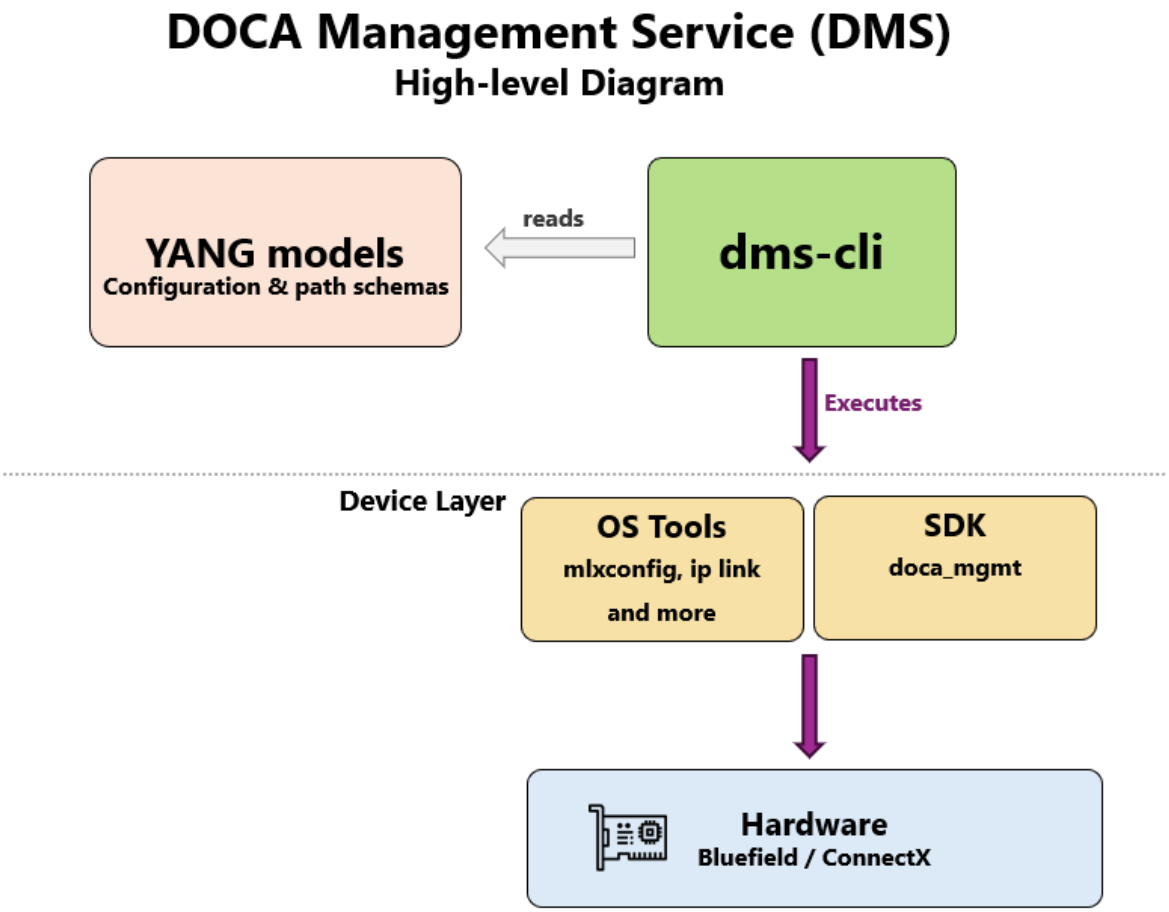
```
/opt/mellanox/doca/services/dms/dms-cli --tree /nvidia                # browse all available paths
/opt/mellanox/doca/services/dms/dms-cli --describe /nvidia/mode       # explain a specific path
```

DMS uses NVIDIA-native YANG models - paths follow the /nvidia/... structure rather than the

OpenConfig conventions used in the legacy interface.
For the full list of supported models and paths, see YANG Models.

8.10.2.3 2.3. OS Tools

DMS runs the appropriate operation on the device according to the command sent, either by invoking system tools such as mlxconfig, devlink, and dcb, or via the DOCA Management SDK for operations that require direct hardware access (more on that in DOCA Management SDK).



8.10.3 3. YANG Models

Each YANG model covers one functional area under the flat `/nvidia/` tree. All models are NVIDIA-native (not OpenConfig).
The YANG schema is the single source of truth for what you can read, write, or trigger. Explore it using the --tree command:

```
/opt/mellanox/doca/services/dms/dms-cli --tree /nvidia              # browse all available paths
/opt/mellanox/doca/services/dms/dms-cli --describe /nvidia/cc       # explain a specific container
```

Each YANG model covers one functional area under the flat /nvidia/ tree.
The table below lists all supported models grouped by category.

8.10.3.1 3.1. Networking & Interfaces

Model	Path	Covers
nvidia-root	`/nvidia`	Root container and entry point for all paths
nvidia-link	`/nvidia/link`	Link admin/operational state, netdev MTU, physical port (PAOS), link type, keep-link-up policy, IPG, breakout
nvidia-lag	`/nvidia/lag`	LAG resource allocation strategy (HWLB prerequisite. not classic bonding config)
nvidia-eswitch	`/nvidia/eswitch`	E-switch mode (legacy/switchdev), flow-steering mode, multiport, flex parser profile
nvidia-multiplane	`/nvidia/multiplane`	Multi-plane / multipath: number of planes and load-balance mode
nvidia-qos	`/nvidia/qos`	QoS: trust-mode, PFC priorities, ETS priority-to-TC mapping
nvidia-roce	`/nvidia/roce`	RoCE: adaptive routing, multipath DSCP, RTT, ToS, ACCL (slow-restart, tx-window, selective-repeat, etc.)
nvidia-cc	`/nvidia/cc`	Congestion control: user-programmable CC, shaper coalesce, algo slots/params, NP/RP global status, NP CNP settings
nvidia-data-direct	`/nvidia/data-direct`	VF/SF data-direct enable/disable
nvidia-icm	`/nvidia/icm`	ICM (Initiator Connection Manager) quota settings

8.10.3.2 3.2. Device & Platform

Model	Path	Covers
nvidia-platform	`/nvidia/platform`	Platform / Orchid system-conf operations
nvidia-mode	`/nvidia/mode`	BlueField operational mode (`nic` / `dpu`). pending next-boot value
nvidia-pci	`/nvidia/pci`	PCIe / SR-IOV: enable, max VFs, runtime VF count, BAR size, PF count
nvidia-nvconfig	`/nvidia/nvconfig`	Bulk NVConfig apply (typed DMS intent and optional raw TLVs via mlxconfig)
nvidia-emulation	`/nvidia/emulation`	Typed emulation-manager controls (e.g. virtio-net, NVMe) for delegated PF targets

Model	Path	Covers
nvidia-misc	`/nvidia/misc`	Miscellaneous device settings (e.g. flex parser profile)
nvidia-raw	`/nvidia/raw`	Escape hatch: raw register access, HCA capabilities, raw NVConfig query/ modify

8.10.3.3 3.3. Lifecycle & Operations

Model	Path	Covers
nvidia-artifacts	`/nvidia/artifacts`	Local artifact cache: stage, list, get-info, delete (BFB/FW/ISO/ PLDM/cfg)
nvidia-provisioning	`/nvidia/provisioning`	Firmware / BFB / image install and version queries (ConnectX and BlueField)
nvidia-blueprints	`/nvidia/blueprints`	Profile-based NIC provisioning plans (e.g. SPC-X / MRC): plan, install, verify, cleanup
nvidia-reset	`/nvidia/reset`	Device reset: required/ suggested status, capabilities, and `execute` action
nvidia-operation	`/nvidia/operation`	Async operation status lookup by operation ID
nvidia-diagnostics	`/nvidia/diagnostics`	Diagnostics data interface (multi-domain diagnostic ownership)

> ***Note: Paths shown at runtime may be filtered by target and device capability. Use*** `-t` ***with*** `--tree` / `--describe` ***to see what applies to a specific device.***

8.10.4 4. DMS-CLI Flags

`dms-cli` is path-centric: GET, SET, and ACTION are deduced from the path and arguments (no `--get` / `--set` / `--action` flags).
`dmsc` is an installed alias for `dms-cli` (same binary invocation).

Default install path: `/opt/mellanox/doca/services/dms/dms-cli` (and `dmsc` beside it).

8.10.4.1 4.1. Common flags

Flag	Short	Description
`--target`	`-t`	Device target. Grammar: `pci/<BDF>[,<rep-suffix>][?port=N&scope=S]` (see 1.2.3)
`--json`		Output GET/SET/ACTION results as JSON (ignored for introspection)
`--dry-run`		Validate only. Do not write or execute
`--verbose`	`-v`	Show debug information
`--help`	`-h`	Usage

8.10.4.2 4.2. Introspection flags

Introspection lets you explore what DMS supports without touching any device hardware. All introspection operations read the installed YANG schema only. No device communication occurs, and `--json` has no effect.

More about introspection operations: DOCA Management Service Guide

Flag	Short	Description
`--describe`		describe the node or leaf at PATH
`--depth`		with --tree: maximum subtree depth (0 = unlimited)
`--tree`		print schema subtree for PATH

For the legacy OpenConfig-based workflow and older tooling, see the **legacy DMS tool guide**.

8.10.5 5. Operations

`dms-cli` does not use `get` / `set` / `action` subcommands. Operation type is deduced from the path and arguments:

Operation	Pattern
GET	Path ends at a container or leaf, no `=`

Operation	Pattern
SET	Path or arguments contain `=` (`leaf=value`)
ACTION	Path points to a YANG **action** node

Most device operations need `-t` / `--target`. Use `dms-cli list-targets` to discover BDFs.

DMS uses a flat `/nvidia/...` tree: writable leaves are config. read-only leaves use YANG `config false` (often named `*-pending`, `oper-status`, caps, etc.).
Some containers are still named `config` (for example, `/nvidia/cc/config/`). that is a feature name, not a config/state axis.

Default output is plain `key: value`. Add `--json` for structured JSON on GET/SET/ACTION (introspection ignores `--json`).

8.10.5.1 5.1. Get

Syntax

```
dms-cli -t pci/<BDF> <path>                  # container: all readable leaves
dms-cli -t pci/<BDF> <path> <leaf> [...]     # container + leaf filter
dms-cli -t pci/<BDF> <path>/<leaf>           # single leaf
```

Examples

```
# Container
dms-cli -t pci/0000:03:00.0 /nvidia/mode
# operating-mode: dpu
# operating-mode-pending: dpu

# Single leaf
dms-cli -t pci/0000:03:00.0 /nvidia/mode/operating-mode
# operating-mode: dpu

# JSON
dms-cli --json -t pci/0000:03:00.0 /nvidia/mode/operating-mode
# { "operating-mode": "dpu" }
```

Subtree GET returns all readable leaves under that container. Leaf GET returns one value. There is no separate timestamp field in the CLI response envelope.

8.10.5.2 5.2. Set

Syntax

```
dms-cli -t pci/<BDF> <path>/<leaf>=<value>              # single leaf
dms-cli -t pci/<BDF> <path> <leaf>=<value> [<leaf>=…]   # multi-leaf under container
dms-cli -t pci/<BDF> ... --dry-run                      # validate only
```

Examples

```
# MTU (netdev)
dms-cli -t pci/0000:03:00.0 /nvidia/link/netdev/mtu=9000

# QoS: trust-mode + PFC priorities
dms-cli -t pci/0000:03:00.0 /nvidia/qos trust-mode=dscp
dms-cli -t pci/0000:03:00.0 /nvidia/qos/pfc enabled-priorities=3,4,5

# Mode (BlueField)
```

```
dms-cli -t pci/0000:03:00.0 /nvidia/mode/operating-mode=dpu
# status: ok
```

Pending / reset-required leaves:

Note: Many NVConfig-backed leaves take effect only after device reset (or power cycle).
After SET, the running value may be unchanged while the companion `*-pending` leaf shows the next-boot value.
Check with GET on both the base leaf and `*-pending`, and with `/nvidia/reset/required` when applicable (Use `--describe` on a path to see the available leaves)

Example: Changing BlueField operating mode:
Switching from NIC to DPU mode requires a power cycle to take effect. To read the current mode and the pending value:

```
dms-cli -t pci/0000:03:00.0 /nvidia/mode
# operating-mode: nic
# operating-mode-pending: dpu
```

8.10.5.3 5.3. Actions

Syntax

```
dms-cli -t pci/<BDF> <action-path> [<param>=<value> ...]
```

Actions are one-shot operations with side effects (not leaf read/write).

Examples:

```
# Install an image/FW on a device (ConnectX FW / BlueField BFB / BF4 flows - see model inputs)
dms-cli -t pci/0000:03:00.0 /nvidia/provisioning/install source=staged:<name>

# Version queries
dms-cli -t pci/0000:03:00.0 /nvidia/provisioning/fw/get-version
dms-cli -t pci/0000:03:00.0 /nvidia/provisioning/image/get-versions
```

Other common actions: `/nvidia/blueprints/plan|install|verify|...`, `/nvidia/artifacts/list|delete|get-info`

8.10.5.4 5.4. Batch operations & target scoping

Single-container multi-leaf: several `leaf=value` arguments on one path (one backend call).

Cross-container batch: separate groups with `;` (escape as `\;` in the shell). All groups must be the same type (all GET or all SET). Mixed GET+SET in one batch is rejected. All groups share one `-t`.

```
# Multi-container GET
dms-cli -t pci/0000:03:00.0 /nvidia/roce \; /nvidia/cc/np \; /nvidia/link

# Multi-container SET
dms-cli -t pci/0000:03:00.0 /nvidia/roce adaptive-routing=true \; /nvidia/cc/config user-programmable=true
```

There is no `--request-file` flag. Partial batch failure returns a partial-success result (`status: partial` with successes/failures).

Target scoping: one invocation addresses one `-t`. For multiple NICs/DPUs, run once per target (or script over `list-targets`):

```
dms-cli list-targets
# pci/0000:03:00.0
# pci/0000:81:00.0

dms-cli -t pci/0000:03:00.0 /nvidia/mode/operating-mode
dms-cli -t pci/0000:81:00.0 /nvidia/mode/operating-mode
```

Representors: **-t pci/<BDF>,pf<N>vf<M>** or **pf<N>sf<M>**. Optional query tail: **?port=N** / **?scope=S** - see DOCA Management Service Guide

8.10.6 6. Introspection

DMS lets you explore the YANG schema directly from the command line.
Discover what paths are available, understand what a specific parameter does, and see what values it accepts.
No hardware access is involved. introspection reads the schema only.
--json does not apply.

8.10.6.1 6.1. tree

Print the YANG schema subtree rooted at PATH as an indented tree.
Use it to browse which containers, lists, leaves, and actions are available under a given feature area.

Use **--depth N** together with **--tree** to limit how many levels deep the output goes (0 = unlimited, which is the default).

--depth requires **--tree**.

```
# Full tree for path /nvidia/mode

> /opt/mellanox/doca/services/dms/dms-cli --tree /nvidia/mode
+--rw mode
   +--ro caps
      +--rw operating-mode   boolean
   +--rw operating-mode   device-mode
   +--ro operating-mode-pending   device-mode
```

8.10.6.2 6.2. describe

Describe a single node or leaf: its YANG type, enumerated values, whether it is read-only or writable, and any descriptive text from the model.
Use **--describe** when you know a path but want to understand which values are valid before reading or writing it.
It is also the best way to see the input parameters of an action before running it.

--tree and **--describe** are mutually exclusive.

```
# Leaf with enums + annotation (power-cycle required)

> /opt/mellanox/doca/services/dms/dms-cli --describe /nvidia/mode
+--rw mode
   +--ro caps
      +--rw operating-mode (boolean)?
   +--rw operating-mode (enumeration)? [reset] [cap-mode-operating-mode]
   |     Values: nic, dpu
   +--ro operating-mode-pending (enumeration)? [cap-mode-operating-mode]
   |     Values: nic, dpu
```

Use `--tree` to browse the structure of a path: what containers and leaves exist under it.
Use `--describe` to understand a specific node: its type, accepted values, and any constraints such as whether a reset is required after a change.

8.10.6.3 6.3. list-xpaths

Print every leaf XPath reachable under PATH (default: the full schema root).
Each line is a complete, directly addressable path to a readable or writable leaf. Use it to discover all settable and readable parameters in a feature area without reading the YANG model files.

```
# List all configurable paths under /nvidia/qos

> /opt/mellanox/doca/services/dms/dms-cli list-xpaths /nvidia/qos
/nvidia/qos/caps/both-trust-supported
/nvidia/qos/caps/dscp-trust-supported
/nvidia/qos/caps/pcp-trust-supported
/nvidia/qos/ets/prio-tc-map
/nvidia/qos/ets/tc-mapping/[<tc-index>]/tc-index
/nvidia/qos/ets/tc-mapping/[<tc-index>]/traffic-class
/nvidia/qos/pfc/enabled-priorities
/nvidia/qos/trust-mode
```

8.10.6.4 6.4. list-targets

Enumerate the device targets present on the local host.
The `target` field in each record is the value to supply to `-t / --target`. Use `--json` for machine-readable output suitable for scripting.

```
# List all available devices on the host

> /opt/mellanox/doca/services/dms/dms-cli list-targets

Device 1:
    chip_type=BF3
    func_type=PF
    iface_name=p0
    target=pci/0000:03:00.0
    ...
Device 2:
    chip_type=CX7
    func_type=PF
    iface_name=p1
    target=pci/0000:81:00.0
    ...
```

8.10.7 7. Provisioning and Reset

When you deploy a new device or update an existing one, you need more than just configuration. You need to install the right firmware or software image, verify what is currently running, and reset the device to apply changes.

DMS covers the full device lifecycle: staging and installing images, querying running and pending versions, checking whether a reset is needed, and executing it.
These operations are designed for deployment pipelines, fleet management scripts, and anyone bringing up or updating BlueField DPUs and ConnectX NICs at scale.

Unlike Get and Set, install and reset actions invoke external tools that can run for several minutes and may temporarily remove the device from the PCIe bus.
The lifecycle API is divided into four areas:

Area	Path	Scope	Behavior
Artifact cache	`/nvidia/artifacts`	System	Stages, lists, inspects, and deletes local files. These actions complete before returning.
Provisioning	`/nvidia/provisioning`	Device	Install ConnectX firmware, BlueField-3 BFBs, and BlueField-4 ISO or PLDM payloads. Installation is asynchronous.
Operation status	`/nvidia/operation/get-status`	Device	Reports the latest install or reset operation for the selected physical adapter.
Reset	`/nvidia/reset`	Device	Reports reset state and capabilities and starts an asynchronous device reset.

> *The* **`/nvidia/reset`** *API performs a firmware or device reset through* **`mlxfwreset`**.
> *It does not restore factory-default configuration.*

8.10.7.1 7.1. Lifecycle Workflow

The common lifecycle flow for provisioning and resetting a device:

1. Discover the target PCI address with **`list-targets`**.
2. Stage each local payload in **`/nvidia/artifacts`**.
3. Start **`/nvidia/provisioning/install`** on one target.
4. Poll **`/nvidia/operation/get-status`** on the same target until the operation reaches **`completed`** or **`failed`**.
5. Query **`/nvidia/reset/required`**, and read **`/nvidia/reset/message`**.
6. Run **`/nvidia/reset/execute`** when a device reset is appropriate. Follow **`/nvidia/reset/message`** when it calls for an external host reboot or platform power cycle instead.
7. Verify running versions after activation.

Use the installed client in all examples:

```
/opt/mellanox/doca/services/dms/dms-cli
```

Discover available targets:

```
> /opt/mellanox/doca/services/dms/dms-cli list-targets

pci/0000:03:00.0
pci/0000:61:00.0
```

The examples below use **`pci/0000:03:00.0`**. Replace it with the target returned for the device in the deployment.

BlueField-4 note: A BMC hostname does not identify the DMS target. BlueField-4 out-of-band operations still use **-t pci/<BDF>** for DMS operation tracking, while bmc-host identifies the BMC used by the backend. The caller must supply the BMC that belongs to the selected device.

Use **--describe** to view the exact inputs supported by the installed version:

```
/opt/mellanox/doca/services/dms/dms-cli --describe /nvidia/provisioning/install
/opt/mellanox/doca/services/dms/dms-cli --describe /nvidia/reset/execute
```

8.10.7.1.1 7.1.1. Supported Hardware Flows

Device or mode	Install payload	Install path	Version or comparison path	Activation notes
ConnectX	Firmware image (**fw**)	Local, target-scoped firmware burn	**/nvidia/provisioning/fw/get-version**	Query reset state after installation.
BlueField-3 DPU mode	BFB, with optional configuration file	Local, in-band through the target's RSHIM device	**/nvidia/provisioning/image/get-versions** and **/nvidia/provisioning/image/diff**	Query reset state after installation.
BlueField-4 DPU mode	ISO, PLDM, or both	Out-of-band through the DPU BMC	**/nvidia/provisioning/image/get-versions** with BMC inputs	PLDM activation requires an external full card power cycle.
BlueField-4 NIC mode	PLDM	Local, target-scoped update	Backend-dependent	The local device must be resolvable through MFT/MST.
BlueField-2	None in this release API	-	Image version and diff operations return **UNSUPPORTED** when BF2 is detected	Use a supported legacy workflow where required. (3.5.0) Doca management service - OpenConfig implementation (legacy)

Reset support is determined at runtime by **mlxfwreset** for the selected device.
Query **/nvidia/reset/capabilities** before selecting reset parameters.

8.10.7.2 7.2. Artifact Management

The **/nvidia/artifacts** actions manage the local cache used by provisioning. They are system-scoped, so they do not take **-t**.
The default cache directory is **/var/lib/dms/artifacts**, but callers use artifact keys rather than this internal path.
Artifact keys are names, not paths. They cannot contain path separators or traversal sequences.

A staged artifact is addressed by its key: `staged:<name>`
For example, an artifact staged as `bf3.bfb` is installed as `source=staged:bf3.bfb`.

Note: An absolute input path refers to a file visible on the host where DMS executes, not a path on a remote client machine.
Staging is recommended for repeatable operation and inspection.

Supported artifact types are:

Type	Purpose
`bfb`	BlueField Boot File for BlueField-3 installation.
`iso`	BlueField-4 operating-system ISO or CIDATA seed ISO.
`fw`	ConnectX firmware image.
`cfg`	Optional BlueField-3 BFB configuration file.
`pldm`	BlueField-4 PLDM firmware package.
`other`	Cached file without a provisioning-specific type.

8.10.7.2.1 7.2.1. Stage an Artifact

`/nvidia/artifacts/stage` validates the request and copies a local file into the artifact cache.
The copy is committed atomically, so a partially copied file is not exposed as a completed artifact.

Input	Required	Description
`input-file`	Yes	Absolute source path on the DMS execution host.
`staged-file`	No	Artifact key. Defaults to the base name of `input-file`.
`type`	No	Explicit `bfb`, `iso`, `fw`, `cfg`, `pldm`, or `other` type. When omitted, DMS infers the type from the extension where possible.
`force`	No	Replace an existing artifact with the same key. Defaults to `false`.

Stage a BlueField-3 BFB:

```
> /opt/mellanox/doca/services/dms/dms-cli /nvidia/artifacts/stage \
  input-file=/images/bf-bundle.bfb \
  staged-file=bf3.bfb \
  type=bfb

# output:
{
  "status": "completed",
  "type": "bfb",
  "error-message": ""
}
```

Stage a ConnectX firmware image:

```
/opt/mellanox/doca/services/dms/dms-cli /nvidia/artifacts/stage \
  input-file=/firmware/connectx.bin \
  staged-file=connectx.bin \
  type=fw
```

Stage BlueField-4 ISO and PLDM payloads:

```
/opt/mellanox/doca/services/dms/dms-cli /nvidia/artifacts/stage \
  input-file=/images/bf4-os.iso \
  staged-file=bf4.iso \
  type=iso

/opt/mellanox/doca/services/dms/dms-cli /nvidia/artifacts/stage \
  input-file=/firmware/bf4.fwpkg \
  staged-file=bf4.fwpkg \
  type=pldm
```

Status	Meaning
completed	The file was validated as applicable and committed to the cache.
use-cached	The key already existed and **force** was not set. DMS kept the existing artifact.
failed	Validation, copying, type detection, or cache registration failed. **error-message** describes the immediate error.

To replace an existing artifact:

```
/opt/mellanox/doca/services/dms/dms-cli /nvidia/artifacts/stage \
  input-file=/images/bf-bundle-new.bfb \
  staged-file=bf3.bfb \
  type=bfb \
  force=true
```

DMS performs native non-destructive validation before copying installable BFB and ConnectX firmware images:

Type	Validation during stage	Validation before install
bfb	**mlx-mkbfb** integrity check	Repeated before starting **doca-installer**.
fw	**flint** image verification	Repeated before starting the firmware burn.
iso, **pldm**, **cfg**, **other**	File, path, type, and cache checks	Payload-specific validation is performed by the applicable backend tool.

Repeating BFB and firmware validation before install protects against a cached file being modified after staging.

8.10.7.2.2 7.2.2. List Artifacts

/nvidia/artifacts/list returns the artifact key, recorded type, and size in bytes for every valid cached artifact.

```
/opt/mellanox/doca/services/dms/dms-cli --json /nvidia/artifacts/list
```

output:

```
{
  "files": [
    {
      "name": "bf3.bfb",
      "type": "bfb",
      "size": 1560281088
    },
    {
      "name": "connectx.bin",
      "type": "fw",
      "size": 4194304
    }
  ]
}
```

8.10.7.2.3 7.2.3. Get Artifact Information

`/nvidia/artifacts/get-info` inspects one cached artifact. `name` is the artifact key without the `staged:` prefix.

```
/opt/mellanox/doca/services/dms/dms-cli /nvidia/artifacts/get-info name=bf3.bfb
```

Type	Information
`bfb`	Size, BFB image versions, available firmware components, and raw `doca-installer` inspection output.
`fw`	Size, firmware version, PSID, component versions, and raw `flint` output.
`iso` , `pldm` , `cfg` , `other`	Type and size. Detailed ISO and PLDM validation remains part of the installation backend flow.

The `firmware-component` entries use `name` , `running-version` , and `pending-version` . For a static artifact, `running-version` is the version reported by the inspection tool and `pending-version` is normally `N/A` because the file itself has no running/pending state.

8.10.7.2.4 7.2.4. Delete an Artifact

`/nvidia/artifacts/delete` removes one cached artifact and its type metadata.

```
/opt/mellanox/doca/services/dms/dms-cli --json /nvidia/artifacts/delete name=bf3.bfb
```

Example response:

```
{
  "success": true,
  "error-message": ""
}
```

Deletion fails when the key is invalid or the artifact does not exist.

8.10.7.3 7.3. Provisioning Install

`/nvidia/provisioning/install` starts one installation on one target. It has two mutually exclusive modes:

- Local mode uses `source` for ConnectX firmware, BlueField-3 BFB, or BlueField-4 NIC-mode PLDM.
- Out-of-band mode uses BMC credentials together with an ISO, a PLDM package, or both for BlueField-4 DPU mode.

8.10.7.3.1 7.3.1. Install Inputs and Source Rules

Input	Mode	Description
`source`	Local	`staged:<name>` or an absolute local path.
`config-source`	Local BF3 BFB only	Optional staged key or absolute path for a BFB configuration file.
`bmc-host`	BF4 OOB	DPU BMC hostname or IP address. Required for OOB installation.
`bmc-user`	BF4 OOB	BMC username. Defaults to `admin`.
`bmc-password`	BF4 OOB	BMC password. Required for OOB installation.
`iso-source`	BF4 OOB	Local or staged ISO. `doca-installer` temporarily serves it to the BMC.
`iso-cidata-source`	BF4 OOB local ISO	Optional local or staged cloud-init CIDATA ISO.
`iso-url`	BF4 OOB	HTTPS URL pulled directly by the BMC.
`pldm-source`	BF4 OOB	Local or staged PLDM firmware package. May be combined with an ISO.
`pldm-force`	BF4 OOB PLDM	Requests backend ForceUpdate semantics when supported. Defaults to `false`.

Rules:

- `source` cannot be combined with BMC or OOB payload inputs.
- `iso-source` and `iso-url` are mutually exclusive.
- `iso-cidata-source` is valid only with `iso-source`. it cannot be used with `iso-url`.
- OOB mode requires `bmc-host`, `bmc-password`, and at least one of `iso-source`, `iso-url`, or `pldm-source`.
- `config-source` is valid only with a BFB `source`.
- Generic URLs and PXE are not valid `source` values. Remote ISO is supported only through HTTPS `iso-url`.
- Absolute paths bypass the artifact cache and must be visible on the DMS execution host. Staging is recommended for repeatable operation and inspection.

⚠ BMC credentials: **bmc-password** is a plaintext input passed to the backend tool. Use an account with minimum required permissions. Avoid retaining commands containing the password in shell history, logs, or automation output.

8.10.7.3.2 7.3.2. Asynchronous Start Behavior

A successful install request starts the backend and returns immediately:

```
{
  "status": "in-progress",
  "log-ref": "/var/lib/dms/operations/logs/install-pci_0000_03_00.0-<timestamp>-<pid>.log",
  "message": ""
}
```

in-progress means that DMS accepted the request and created the backend process. It does not mean that installation succeeded. The final result is available through **/nvidia/operation/get-status**.

If validation fails before a process is created, the install action returns **status: failed**, an empty **log-ref**, a non-empty **message**, and a nonzero command exit code.

Only one install or reset can run on a physical adapter at a time.

8.10.7.3.3 7.3.3. Install ConnectX Firmware

ConnectX firmware installation uses a firmware artifact and a target-scoped **flint** burn.

Stage and inspect the image:

```
/opt/mellanox/doca/services/dms/dms-cli /nvidia/artifacts/stage \
  input-file=/firmware/connectx.bin \
  staged-file=connectx.bin \
  type=fw

/opt/mellanox/doca/services/dms/dms-cli /nvidia/artifacts/get-info \
  name=connectx.bin
```

Start the installation:

```
/opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 /nvidia/provisioning/install source=staged:connectx.bin
```

DMS verifies the firmware image again immediately before starting the target-scoped firmware burn. After completion, query the running firmware version and reset state.

8.10.7.3.4 7.3.4. Install a BFB on BlueField-3

BlueField-3 BFB installation is a local, in-band operation. DMS maps the selected PCI target to its RSHIM device and starts **doca-installer** with the staged BFB.

Stage the BFB:

```
/opt/mellanox/doca/services/dms/dms-cli
  /nvidia/artifacts/stage \
  input-file=/images/bf-bundle.bfb \
  staged-file=bf3.bfb \
```

```
  type=bfb
```

Start the installation:

```
/opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 --json /nvidia/provisioning/install \
  source=staged:bf3.bfb
```

To include an optional configuration file, stage it and provide **config-source** in the same action:

```
/opt/mellanox/doca/services/dms/dms-cli /nvidia/artifacts/stage \
  input-file=/configs/bf.cfg \
  staged-file=bf3.cfg \
  type=cfg

/opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 /nvidia/provisioning/install \
  source=staged:bf3.bfb \
  config-source=staged:bf3.cfg
```

The BFB integrity check runs during stage and again before **doca-installer** is started. A missing or invalid RSHIM mapping rejects the request before installation begins.

8.10.7.3.5 7.3.5. Install on BlueField-4

BlueField-4 supports several install flows depending on the device mode and payload type. Use **--describe /nvidia/provisioning/install** to see exact inputs for your installed version.

8.10.7.3.5.1 7.3.5.1. DPU mode - Local ISO

BlueField-4 DPU-mode OS installation is out-of-band. The controller running DMS communicates with the BMC, and the BMC performs the platform operation.
Before starting, verify that:

- The selected BMC belongs to the PCI target used with **-t**.
- The DMS host can reach the BMC management interface.
- The BMC can reach the DMS host for a locally supplied ISO.
- The BMC firmware and ISO meet the requirements of the installed **doca-installer** release.
- The BMC credentials are valid.

```
# Stage the ISO:

/opt/mellanox/doca/services/dms/dms-cli /nvidia/artifacts/stage \
  input-file=/images/bf4-os.iso \
  staged-file=bf4.iso \
  type=iso

# Start the OOB installation:

/opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 /nvidia/provisioning/install \
  iso-source=staged:bf4.iso \
  bmc-host=bf4-bmc.example.com \
  bmc-user=admin \
  bmc-password='<BMC_PASSWORD>'
```

The backend flow:

1. DMS resolves the staged key to the local cached file.
2. **doca-installer** starts a temporary HTTP service for the ISO.
3. The BMC downloads the ISO from the DMS host and mounts it as virtual media.
4. The BMC boots the installation media and reports progress through Redfish.

5. **doca-installer** follows progress, ejects the media when appropriate, and stops the temporary service.
6. DMS records the final backend result for **/nvidia/operation/get-status**.

> The BMC must have a network route to the address and port selected on the DMS host.

8.10.7.3.5.2 7.3.5.2. DPU mode - With cloud-init seed (CIDATA)

Stage the CIDATA ISO alongside the OS ISO and add iso-cidata-source:

```
# To provide a cloud-init seed, stage the CIDATA ISO and include it with the local OS ISO:

/opt/mellanox/doca/services/dms/dms-cli /nvidia/artifacts/stage \
  input-file=/images/cidata.iso \
  staged-file=cidata.iso \
  type=iso

/opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 /nvidia/provisioning/install \
  iso-source=staged:bf4.iso \
  iso-cidata-source=staged:cidata.iso \
  bmc-host=bf4-bmc.example.com \
  bmc-user=admin \
  bmc-password='<BMC_PASSWORD>'
```

The seed image must satisfy the CIDATA format and volume-label requirements of the installed provisioning tool.

8.10.7.3.5.3 7.3.5.3. DPU mode - ISO from HTTP

When the ISO is already hosted at an HTTPS URL reachable by the BMC, staging is not required. The BMC downloads the ISO directly - DMS does not download, stage, or serve it.

```
# ISO from HTTPS (BMC downloads directly, no staging needed):

/opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 /nvidia/provisioning/install \
  iso-url=https://images.example.com/releases/bf4-os.iso \
  bmc-host=bf4-bmc.example.com \
  bmc-user=admin \
  bmc-password='<BMC_PASSWORD>'
```

The URL must use HTTPS. The BMC must be able to resolve and reach the URL host.

8.10.7.3.5.4 7.3.5.4. DPU Mode - PLDM Out-of-Band

In BlueField-4 DPU mode, PLDM firmware is staged through the BMC.

```
/opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 /nvidia/provisioning/install \
  pldm-source=staged:bf4.fwpkg \
  bmc-host=bf4-bmc.example.com \
  bmc-user=admin \
  bmc-password='<BMC_PASSWORD>'
```

Add **pldm-force=true** to request ForceUpdate behavior from the backend.

8.10.7.3.5.5 7.3.5.5. DPU Mode - ISO + PLDM Combined

An OOB PLDM package can be combined with a local ISO in one action. The backend stages PLDM first, then installs the OS.

```
/opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 /nvidia/provisioning/install \
  pldm-source=staged:bf4.fwpkg \
  iso-source=staged:bf4.iso \
```

```
  bmc-host=bf4-bmc.example.com \
  bmc-user=admin \
  bmc-password='<BMC_PASSWORD>'
```

The ISO flow can reset the Arm subsystem but does not activate the staged PLDM firmware. A full card power cycle is required after the combined operation completes. The currently running BMC must already satisfy the ISO installation requirements.

8.10.7.3.5.6 7.3.5.6. NIC Mode - PLDM local

In BlueField-4 NIC mode, the DPU BMC is not used. DMS resolves the PCI target to its local MST device and invokes the target-scoped in-band backend.

```
/opt/mellanox/doca/services/dms/dms-cli /nvidia/artifacts/stage \
  input-file=/firmware/bf4.fwpkg \
  staged-file=bf4.fwpkg \
  type=pldm

/opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 /nvidia/provisioning/install \
  source=staged:bf4.fwpkg
```

This flow requires the target to be available through the local MFT/MST device inventory.

bmc-password is currently a plaintext action input and is passed to the backend tool. Treat it as sensitive:

- Use an account with the minimum permissions required by the provisioning operation.
- Avoid retaining commands containing the password in shell history, logs, or automation output.
- Restrict access to DMS logs and process information on the execution host.

8.10.7.4 7.4. Version Queries

Version queries answer different questions depending on what you need to inspect:

Operation	Question answered
/nvidia/artifacts/get-info	What versions or metadata are present in this cached file?
/nvidia/provisioning/fw/get-version	What firmware is running on this ConnectX target?
/nvidia/provisioning/image/get-versions	What image or firmware components are running on this BlueField target?
/nvidia/provisioning/image/diff	How does a staged BlueField-3 BFB differ from the running BlueField-3 image?

These operations cover device firmware and image versions only - not host software packages such as MFT or the Linux kernel.

8.10.7.4.1 7.4.1. Get the Running ConnectX Firmware Version

```
/opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 /nvidia/provisioning/fw/get-version
```

The response includes:

- **psid** : device Parameter Set ID.
- **fw-version** : running firmware version.
- **firmware-component** : extensible component list.
- **raw-output** : complete backend **flint** output for diagnostics.
- **error-message** : empty on success.

8.10.7.4.2 7.4.2. Get BlueField-3 Image Versions

```
/opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 /nvidia/provisioning/image/get-versions
```

The response includes the classic **bsp** , **doca** , **uefi** , and **atf** fields, an extensible **firmware-component** list, **raw-output** , and **error-message** .
A value can be **unknown** or **N/A** when the backend does not report that component.
Use **error-message** and the command exit code to distinguish unavailable metadata from an operation failure.

8.10.7.4.3 7.4.3. Get BlueField-4 OOB Inventory

BlueField-4 inventory is read through its BMC:

```
/opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 /nvidia/provisioning/image/get-versions \
  bmc-host=bf4-bmc.example.com \
  bmc-user=admin \
  bmc-password='<BMC_PASSWORD>'
```

For BlueField-4, use the **firmware-component** list as the primary inventory. The fixed **bsp** , **doca** , **uefi** , and **atf** fields are the BlueField-3 compatibility quartet and may be **unknown** when the OOB backend does not provide them.
Each component contains its running version and a pending version when the BMC reports one, otherwise **pending-version** is **N/A** .

DMS does not verify that **bmc-host** belongs to the selected PCI target.

8.10.7.4.4 7.4.4. Compare a BlueField-3 BFB with the Running Image

/nvidia/provisioning/image/diff is a synchronous BlueField-3-only comparison. The BFB must already be staged, and **image-name** is the artifact key without **staged:** .

```
/opt/mellanox/doca/services/dms/dms-cli --json \
  /nvidia/artifacts/stage \
  input-file=/images/bf-bundle-new.bfb \
  staged-file=bf3-new.bfb \
  type=bfb

/opt/mellanox/doca/services/dms/dms-cli \
  -t pci/0000:03:00.0 --json \
  /nvidia/provisioning/image/diff \
  image-name=bf3-new.bfb
```

The response includes:

- **has-changes** : whether a difference was found.

- **diff-text** : human-readable backend comparison output.
- **component-diff** : parsed current and staged component versions when available.
- **error-message** : backend or validation failure detail.

has-changes: true is a successful comparison result, not an action failure.

8.10.7.5 7.5. Reset

The **/nvidia/reset** API performs a firmware or device reset through **mlxfwreset** . It does not restore factory-default configuration.

After an installation reaches **completed** , follow this order:

1. Check **required** - is a reset needed?
2. Read **message** - what action does the device recommend?
3. Read **suggested-reset** - is a normalized DMS reset available?
4. Check **capabilities** - what levels and types does this device support?
5. Run **execute** only when a device reset is the appropriate action.

8.10.7.5.1 7.5.1. Check Whether Reset Is Required

```
> /opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 --json /nvidia/reset/required
{
  "required": true
}
```

Value	Meaning
true	The backend reports that a reset, reboot, power cycle, or other activation action is required.
false	The backend explicitly reports that no action is required.
unknown	Status timed out, is unavailable, or cannot be parsed reliably. Read **message** and do not assume that no reset is required.

8.10.7.5.2 7.5.2. Read the Reset Message

```
> /opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 --json /nvidia/reset/message
{
  "message": "Driver restart and PCI reset (Level 3). Pending NVCONFIG parameter change"
}
```

message contains free-form reasons and platform guidance. This is where DMS reports instructions that cannot be represented as a normalized device reset, including external host reboot or platform power-cycle requirements.

8.10.7.5.3 7.5.3. Get the Suggested Reset

```
> /opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 --json /nvidia/reset/suggested-reset
```

Stops the NIC driver, resets the device over PCIe, and restarts the driver. The device temporarily disappears from the bus during the sequence.

suggested-reset is filled in only when DMS can derive a concrete reset command from the device status. There are two possible cases:

Case 1: A normalized DMS reset is available.

```
{
  "level": "driver-reset",
  "type": "full",
  "sync": "driver",
  "method": "link-disable"
}
```

In this case, verify the returned values against **/nvidia/reset/capabilities**, then use them with **/nvidia/reset/execute**.

Case 2: Reset is required, but no normalized DMS reset is available.

suggested-reset returns an empty object **{}**. This means the required action is not a normalized DMS device reset — for example, the platform may require an external host reboot or a platform power cycle. Follow /nvidia/reset/message instead. Do not invent reset parameters from an empty suggestion.

8.10.7.5.4 7.5.4. Get Reset Capabilities

```
> /opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 --json /nvidia/reset/capabilities

{
  "default-level": "driver-reset",
  "default-type": "full",
  "supported-levels": ["driver-reset"],
  "supported-types": ["full", "phy-less", "arm-only", "arm-os-shutdown"]
}
```

Capabilities are best-effort and target-specific. DMS exposes device reset levels supported by its public model. Host warm reboot is intentionally outside this API and is reported as text guidance when applicable.

8.10.7.5.5 7.5.5. Execute a Reset

Prefer the values returned by **/nvidia/reset/suggested-reset**, and verify them against **/nvidia/reset/capabilities**.

Input	Values	Description
level	**live-patch**, **driver-reset**	Public reset level.
type	**full**, **phy-less**, **nic-only**, **arm-only**, **arm-os-shutdown**	Reset scope. BlueField-only types are exposed only where applicable.

Input	Values	Description
sync	**tool** , **driver** , **fw**	Component responsible for reset synchronization.
method	**link-disable** , **hot-reset**	PCIe reset method.
force	**true** , **false**	Reserved for future DMS pre-check bypass. It does not bypass operation locks or backend validation.

Execute the normalized suggestion from the earlier example:

```
> /opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 --json /nvidia/reset/execute \
  level=driver-reset \
  type=full \
  sync=driver \
  method=link-disable

{
  "status": "in-progress",
  "log-ref": "/var/lib/dms/operations/logs/reset-pci_0000_03_00.0-<timestamp>-<pid>.log",
  "message": ""
}
```

If **level** is omitted, DMS chooses a safe type-specific backend level:

- **arm-only** and **arm-os-shutdown** use the internal immediate level chosen by the backend.
- Other types use **driver-reset** .

Reset can temporarily remove the target from the PCI bus. Continue polling **/nvidia/operation/get-status** with the same target. Operation status remains available from persisted DMS state even while the hardware is absent.

After the reset reaches a terminal state, re-query reset required and running versions:

```
/opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 --json /nvidia/operation/get-status

/opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 --json  /nvidia/reset/required
```

8.10.7.6 7.6. Monitor Operations

Both **/nvidia/provisioning/install** and **/nvidia/reset/execute** are asynchronous. The selected target is the operation handle. There is no separate operation ID.

Poll the same target used to start the action:

```
/opt/mellanox/doca/services/dms/dms-cli -t pci/0000:03:00.0 --json /nvidia/operation/get-status
```

Output response:

```
# While an install is running:
{
  "status": "in-progress",
  "op-type": "install",
  "message": "",
  "log-ref": "/var/lib/dms/operations/logs/install-pci_0000_03_00.0-<timestamp>-<pid>.log"
}

# Completed reset
{
  "status": "completed",
  "op-type": "reset",
```

```
  "message": "",
  "log-ref": "/var/lib/dms/operations/logs/reset-pci_0000_03_00.0-<timestamp>-<pid>.log"
}
```

Status	op-type	Meaning
idle	none	No operation has been recorded for this target.
in-progress	install or reset	The backend process is still running.
completed	install or reset	The backend finished successfully. message may still contain a completion warning.
failed	install or reset	The backend failed, exited unexpectedly, or is no longer running. Read message and log-ref.

The log is stored on the DMS execution host. Use the exact log-ref returned by the action or status response:

```
tail -f /var/lib/dms/operations/logs/<operation-log>.log
```

For automation, evaluate the returned status field. /nvidia/operation/get-status is a successful query even when it reports status: failed, so the command itself can return exit code 0. By contrast, an install or reset request rejected before asynchronous work starts returns a nonzero command exit code.

8.10.7.7 7.7. Troubleshooting

Result	Meaning and next check
Artifact stage returns use-cached	The existing key was retained. Use get-info, choose another key, or stage with force=true.
Install returns failed with an empty log-ref	DMS rejected the request before starting a backend. Correct the source, type, target, or required inputs.
Install or reset returns in-progress	The operation only started. Poll /nvidia/operation/get-status.
Operation status remains in-progress	Inspect log-ref for backend progress. Long BFB and ISO operations may produce lengthy logs.
Operation status returns failed	Read message and the operation log. The status query itself can still return command exit code 0.
operation already in progress	Another install or reset owns the physical adapter's operation slot. Wait for its terminal status.
staged source not found	Confirm the artifact key with /nvidia/artifacts/list. Do not include a filesystem path after staged:.
Source type mismatch	Use the input that matches the artifact type, such as iso-source for ISO and pldm-source for OOB PLDM.
BF3 install cannot resolve RSHIM	Verify RSHIM availability and its PCI mapping on the DMS host.
Local PLDM or image query cannot resolve MST	Verify MFT/MST installation and that the target appears with a usable MST device.

Result	Meaning and next check
BF4 OOB operation cannot reach an image	Verify controller-to-BMC connectivity and, for local ISO, BMC-to-controller connectivity. For `iso-url`, verify BMC-to-URL connectivity.
Reset `required` is `unknown`	Read `message`; check `mlxfwreset` availability and target support. Do not interpret `unknown` as `false`.
Reset is required but suggestion is `{}`	The required action is not a normalized DMS device reset. Follow `/nvidia/reset/message`.

8.10.7.8 7.8. Path Reference

Path	Operation	Scope	Purpose
`/nvidia/artifacts/stage`	Action	System	Validate as applicable and cache a local artifact.
`/nvidia/artifacts/list`	Action	System	List artifact keys, types, and sizes.
`/nvidia/artifacts/get-info`	Action	System	Inspect one artifact's metadata and available versions.
`/nvidia/artifacts/delete`	Action	System	Delete one artifact.
`/nvidia/provisioning/install`	Action asynchronous	Device	Start ConnectX, BF3, or BF4 provisioning.
`/nvidia/provisioning/fw/get-version`	Action	ConnectX device	Query running ConnectX firmware.
`/nvidia/provisioning/image/get-versions`	Action	BlueField device	Query BF3 local or BF4 OOB image and firmware inventory.
`/nvidia/provisioning/image/diff`	Action	BlueField-3 device	Compare a staged BFB with the running image.
`/nvidia/operation/get-status`	Action	Device	Query the latest asynchronous install or reset state for the adapter.
`/nvidia/reset/required`	Get	Device	Return `true`, `false`, or `unknown`.
`/nvidia/reset/message`	Get	Device	Return reset reasons and platform guidance.
`/nvidia/reset/suggested-reset`	Get	Device	Return a normalized reset command when available.
`/nvidia/reset/capabilities`	Get	Device	Return supported and default reset levels and types.
`/nvidia/reset/execute`	Action, asynchronous	Device	Start a target-scoped device reset.

8.10.8 8. Prerequisites and Dependencies

DMS requires a Linux-based environment with DOCA for Host installed using either the doca-networking or doca-all profile.
Refer to the DOCA Installation Guide for Linux for setup instructions.

The following packages must be present on the host.
dms-cli itself has no runtime dependencies. the packages below are invoked by the OS Tools layer when executing specific operations.

Category	Packages or tools	Purpose	Availability
DOCA SDK	`doca-common`, `doca-mgmt`	DOCA core runtime, including memory and platform abstractions, plus the DOCA Management API for dispatching Get, Set, and Action operations.	Required
MFT (Mellanox Firmware Tools)	`mlxconfig`, `mlxreg`, `flint`, `mst`	NIC firmware parameter get/set, direct register access, firmware version queries and ConnectX firmware flashing, and device enumeration for provisioning.	Required for relevant operations. `mst` is required for provisioning.
Network Tools (iproute2)	`devlink`, `dcb`	E-switch mode, flow steering, NIC device parameters, and QoS trust-mode get/set. `dcb` is preferred over `mlnx_qos`.	Required for relevant operations
MLNX_OFED	`mlnx_qos`, `mlnx-tools` / `mlnx_fw_updater`	DCB trust-mode fallback when `dcb` is unavailable, and the NIC firmware update path.	`mlnx_qos`: fallback. Firmware tools: required for firmware updates.
Provisioning (BlueField only)	`doca-installer`, `rshim` (kernel driver), `mlx-mkbfb`	BFB and PLDM DPU provisioning, local BF3 BFB installations, and BFB repackaging when injecting a configuration file.	Required for relevant operations
Runtime Libraries	`libyaml`, `libpam`	YAML parsing for YANG-to-tool mapping files and PAM authentication for agent-based mode.	`libyaml`: required. `libpam`: required for agent-based mode.
Optional	`jq`	JSON pretty-printing in reset tools.	Optional. Operations degrade gracefully if absent.

8.11 DOCA Pipeline Language Services Guide

This page outlines the DOCA Pipeline Language (DPL) approach to packet processing programmability for NVIDIA BlueField-3 and ConnectX-9 devices. DPL introduces a software development solution based on a domain-specific programming language (DSL), supported by a set of DOCA services.

For more details, refer to the DPL Developer Container.

8.11.1 Introduction

DPL is derived from the P4-16 language specification.

P4 is an open-source, domain-specific programming language (DSL) designed for programming and customizing network data planes. It provides a high-level abstraction for programmable packet processing, allowing developers to add, modify, and extend networking functionalities.

For fixed-function devices, P4 serves as a documentation tool, offering a structured description of the data plane's functional blocks.

8.11.1.1 Key Features of P4

- High-level abstraction: Simplifies the programming of complex network data planes with clear and concise syntax
- Programmable packet processing: Enables customization of packet processing and traffic management
- Documentation of fixed functions: Offers a standardized method for documenting the fixed functional blocks of network devices

8.11.1.2 P4 Compiler

A P4 compiler (p4c) is a critical component in the P4 ecosystem. It automatically generates the data plane program and corresponding control plane interface, ensuring seamless coordination between the data plane and control plane.

Key benefits of a compiler:

- Automatic generation: Streamlines development by automatically generating essential components and optimizing resource usage
- Custom pipeline behavior: Allows developers to extend data plane functionality with customized pipeline behaviors
- Dynamically loadable pipelines: Supports hot-swappable pipelines, enabling updates without rebuilding or redeploying an entire application
- Control plane integration: Facilitates communication between the data plane and control plane via an open-source API, ensuring effective management of customized pipelines

8.11.1.3 Focus on NVIDIA's DOCA Pipeline Language

The remainder of this document focuses on NVIDIA's implementation of the DOCA Pipeline Language (DPL). While DPL's syntax is derived from P4-16, its pipeline semantics align with NVIDIA's DPU architecture rather than standard P4 execution models. For example, while P4 semantics imply a staged pipeline based on a feed-forward RMT (Reconfigurable Match-Action Table) architecture, NVIDIA's DPU architecture follows a run-to-completion dRMT (disaggregated RMT) model, offering greater flexibility and enhanced capabilities.

8.11.1.4 DPL Highlights

DPL introduces a unique programming paradigm distinct from traditional SDKs, APIs, libraries, drivers, or utilities. It is a specialized programming language with a runtime system, designed for

rapid development, testing, and deployment of packet-processing pipelines. DPL is a ready-to-use, customizable solution as part of DOCA Services.

8.11.1.4.1 Key Features of DPL

- DPL Services: A system-level solution that includes a compiler, runtime agent, and debugging tools, enabling rapid programming of the DPU pipeline
- Optimized for NVIDIA devices: Specifically designed and fine-tuned for programming network data planes on NVIDIA hardware
- Advanced networking functionality: Leverages DPL's capabilities to enhance and extend networking features on NVIDIA DPUs
- Comprehensive documentation: Provides detailed descriptions of BlueField's fixed functional blocks within the DPU data plane
- A P4Runtime compatible server utilizing the open-source APIs with vendor specific extensions

8.11.1.4.2 Developer Resources

The DPL programming guide serves as a comprehensive resource for developers who want to use DPL for programming network data planes. By utilizing the DPL `p4c` compiler and the P4-16 specification, developers can:

- Enhance network device functionality and efficiency
- Meet the evolving demands of modern network infrastructures
- Ensure seamless integration and optimization within NVIDIA's DPU ecosystem

8.11.2 Prerequisites

The DPL compiler can run on any Linux OS that supports Docker. Language specifications, runtime APIs, and tutorials are available at the P4 GitHub Repository.

8.11.2.1 Development Environment Requirements

To set up the development environment, the following components are required:

- Host computer with Ubuntu 22.04 or later with Docker installed (required for the DOCA development container)
- Server with root/hypervisor access to install the DPL Runtime Service package
- One or more BlueField-3 devices, installed in the target server for DPL execution

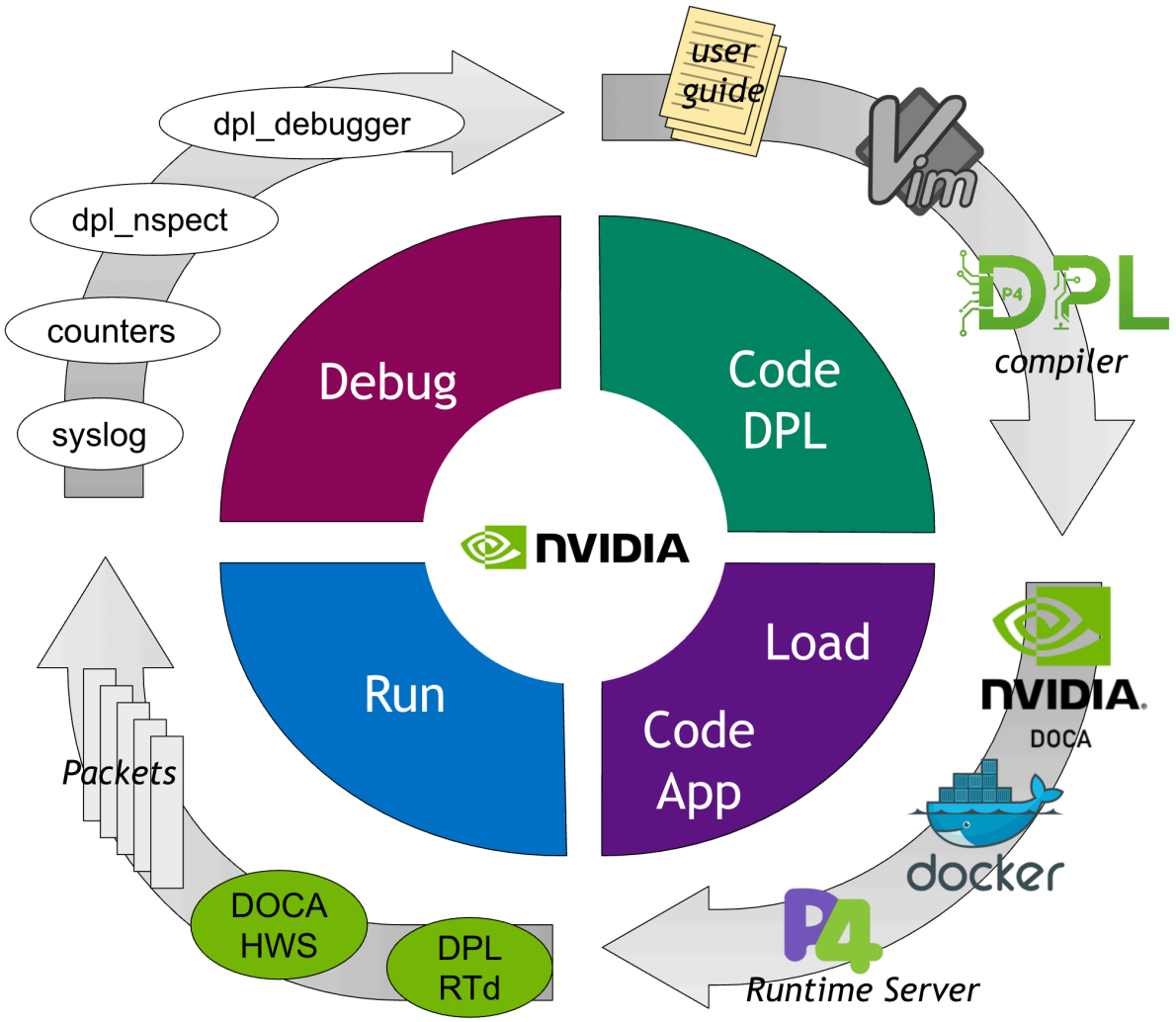
8.11.2.2 Suggested Workflow

The suggested workflow is as follows:

1. Coding
2. a. Use the DPL programming guide and sample applications to create a DPL program remotely.
   b. The program is compiled using `dplp4c`, iterating until it successfully produces a binary.
3. Loading

4. a. The compiled binary is transferred to the BlueField system.
   b. Using the P4Runtime API (via an open-source or proprietary P4Runtime controller), the pipeline is sent from the remote machine to the DPL Service running on BlueField.
   c. The user checks for P4Runtime error messages.
5. Running
6. a. Inspect the logs for any DPL Service error messages.
   b. Use the `dpl_nspect` tool to verify that P4 tables and entries are present in the hardware.
   c. Use the `dpl_debugger` tool to understand the packet processing pipeline, which shows the state of packets and their metadata.

This process is repeated until the DPL application is fully verified.



8.11.3 DPL Programming Model

P4, and by extension DPL, is a domain-specific language (DSL) designed for programming network data planes. It enables customized packet processing, allowing developers to define how packets are handled at different pipeline stages.

However, P4 programs are not universally portable across different architectures. Instead, they are typically compatible within the same target architecture family.

The BlueField programmable pipeline follows a hybrid model that leverages both hardware and software processing capabilities. It consists of three main stages:

1. Parsing
2. Match-action processing (Steering)
3. Forwarding database (FDB)

8.11.3.1 Parsing

The BlueField native parser is the first stage of the packet processing pipeline. It is responsible for identifying and extracting packet headers, progressing through the protocol stack until the entire frame is parsed.

Key features:

- Pre-defined protocol headers and standard transitions based on IETF specifications
- On-demand reparsing at any stage, which eliminates the need for reinjection or a final deparser stage

8.11.3.1.1 Flex Parsing

Flex parsing allows developers to integrate custom protocol headers into BlueField's hardware parsing engine. It consists of four components:

- Flex Arc In: Defines the transition from a native header to a Flex header
- Flex Header: Specifies header characteristics such as length and next protocol location
- Flex Sampler: Extracts specific bytes from the hardware, enabling their use in control blocks or table keys
- Flex Arc Out: Defines the transition from a Flex header back to a native header (or another Flex header)

The DPL compiler automatically generates Flex parsing components based on the developer's defined parse nodes and transitions.

8.11.3.1.2 Operational Mode

- The DPL parser operates in a hybrid mode with a default native parser
- The compiler automatically integrates native headers and fields into DPL constructs
- The Flex parse graph consists of:
    - Nodes (either native or flex)
    - Arcs (transitions)
    - Samplers for custom parsing operations

This design eliminates the need to redefine and reimplement standard IETF protocols and headers.

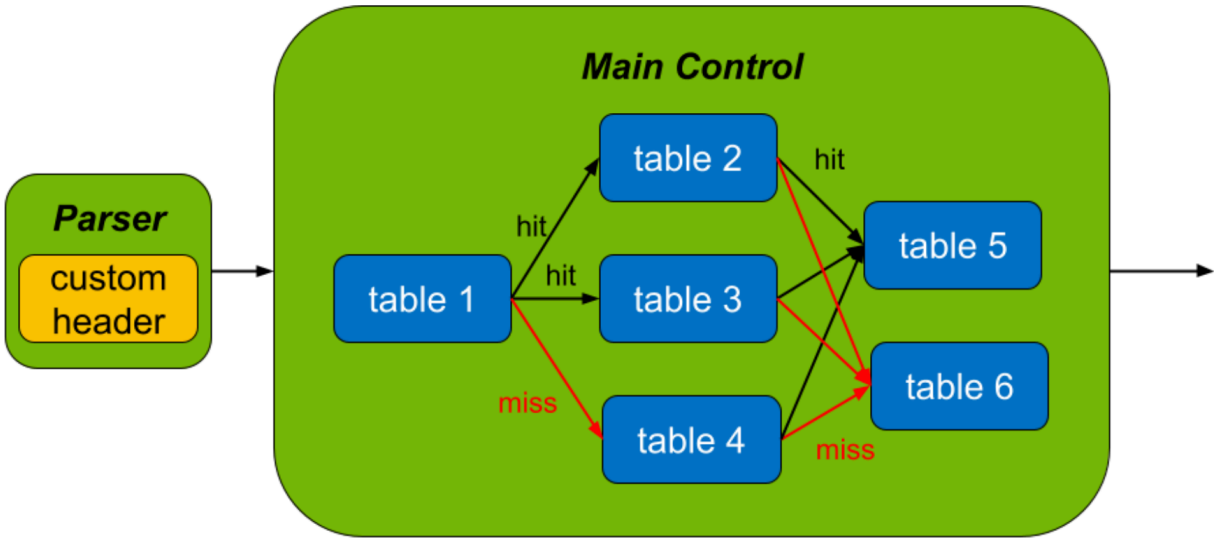
8.11.3.2 Match-Action Processing (Steering)

After parsing, packet processing decisions are made based on match-action tables, commonly referred to as "Steering".

Key Features:

- Match fields: Define packet attributes for classification (for example, source/destination MAC, VLAN, IP, and protocol headers)
- Tables: Store rules for packet handling and decision-making
- Actions: Define processing rules (for example, forwarding, header modification, dropping packets)
- Programmability: Allows dynamic updates to match-action rules based on network conditions
- Efficient processing: Packet handling occurs directly in hardware, reducing latency
- P4Runtime integration: DPL tables are populated via the P4Runtime API, supporting SDN controllers

> In this document, the terms "flow tables" and "P4 tables" are interchangeable.



8.11.3.3 Forwarding Database

The Forwarding Database (FDB) is the final stage within the embedded switch (eSwitch). It enables accurate and efficient packet routing within the network infrastructure. It is responsible for:

- Storing and managing MAC addresses
- Ensuring efficient packet forwarding based on network topology
- Maintaining records of port locations for destination-based forwarding

8.11.3.4 BlueField DPU and ConnectX SuperNIC Pipeline Behavior

The BlueField pipeline is designed for flexibility, allowing developers to customize packet processing to meet specific application needs.

Key characteristics:

- Extended parser support: Developers can expand the native parser using Flex Parsing
- Immediate execution model: No deferred actions; all modifications take effect immediately.
- Mid-pipeline reparsing: Packet headers are reparsed immediately after modification, ensuring correct metadata updates
- No deparser control in TA: Unlike traditional architectures, BlueField does not require a separate deparser step

8.11.3.5 DPL Services

Rather than providing a traditional SDK or driver-level APIs, DPL offers a high-level, services-based approach to programming the DPU pipeline.

The DPL Services consist of 2 packages that form the DPL solution. The services are provided as containers and are deployed separately.

See the following sections on each service:

- DPL System for a high-level overview of the components that make up the DPL Services
- DPL Runtime Service for instructions on deploying and configuring the backend DPL Runtime Service which interacts with the hardware.
- DPL Development Container for information about the DPL language, the compiler tools, and methodology building DPL programs

See also:

- DOCA Pipeline Language Developer Tools to learn about the various tools and methods for debugging DPL programs

8.11.4 DPL Release Notes

8.11.4.1 1. Changes and New Features

This section describes the changes and new features in this software release.

General Features & Enhancements

- Added support for modifying action parameters for both default and non-default entries.

8.11.4.2 2. Related Documentation

- DPL Nspect Release Notes
- DPL Debugger Release Notes

8.11.4.3 3. Capabilities and Limitations

8.11.4.3.1 3.1. General Constraints

- P4 Controller support: Only one P4 Controller can be connected to the DPL Runtime daemon at a time.
- Table and counter object sizes:
  - Must be a power of 2.
  - Must be smaller than 2^{23}.
  - A P4 table of size *N* allows for *N-1* regular entries and 1 default entry.

8.11.4.3.2 3.2. Supported RPC Messages

The following RPC messages are supported by the DPL Runtime daemon:

RPC Message	Support Details & Limitations
Write RPC	• Atomicity: Only `CONTINUE_ON_ERROR` is supported • Batching: Supported
Read RPC	Batching: Supported
Pipeline Config	`SetForwardingPipelineConfig` and `GetForwardingPipelineConfig` are supported.
StreamChannel RPC	Supported updates: `PacketIn`, `PacketOut`, `IdleTimeoutNotification`, `StreamError`.

8.11.4.3.3 3.3. Entity & Operation Support Matrix (Write/Read RPC)

Instead of listing operations and entities separately, developers can consult this matrix to see exactly what operations are permitted on which entity types.

Entity Type	Read RPC	Write RPC: INSERT	Write RPC: DELETE	Write RPC: MODIFY
TableEntry	Supported	Regular entries only	Regular entries only	Default entry only
CounterEntry (Direct/Indirect)	Supported	Unsupported	Unsupported	Supported
MeterEntry (Direct/ Indirect)	Supported	Unsupported	Unsupported	Supported

8.11.4.3.4 3.4. Metering Limits

This table defines the maximum thresholds based on the selected mode.

Mode	Max CIR	Max CBurst	Max PIR	Max PBurst
Bytes	255,000,000,000	2,147,483,648	510,000,000,000	4,294,967,296
Packets	1,992,187,500	16,777,216	3,984,375,000	33,554,432

We don't have a way to export this macro.

8.11.4.3.5 3.5. Unsupported Features

The following P4 Runtime features are currently not supported in this release:

Category	Unsupported Items
RPC Messages	**Capabilities** RPC
StreamChannel Updates	• **MasterArbitrationUpdate** • **DigestList** / **DigestListAck** • **google.protobuf.Any**
Entities	• **ExternEntry** • **ActionProfileMember** • **ActionProfileGroup** • **PacketReplicationEngineEntry** • **ValueSetEntry** • **RegisterEntry** • **DigestEntry**

8.11.4.4 4. Bug Fixes

Reference	Issue
4540834	Description: Entries insertion/deletion rate may degrade when using tables with idle timeout or delayed counters.
	Discovered in version: 1.2.0

8.11.4.5 5. DPL Runtime Daemon Known Issues

The following are known limitations of the DPL Runtime daemon service.

Reference	Issue
4259599	Description: Packets received with double VLAN tags (Dot1AD followed by Dot1Q) are processed with the tags in reversed order. This causes the hardware to misinterpret the inner and outer VLAN structure.
	Workaround: Disable RX VLAN offload on the affected VF network device using the following command: `ethtool -K <vf_netdev> rxvlan off`
	Reported in version: 1.2.0
4437411	Description: Packets with broadcast MAC **ff:ff:ff:ff:ff:ff** transmitted from VFs are not supported by DPL actions **nv_add_entry** and **nv_send_to_controller**.
	Workaround: N/A

Reference	Issue
	Reported in version: 1.2.0
4439323	Description: Modifying eSwitch multiport mode (enable/disable) while DPL RT Service is active causes system hang.
	Workaround: Stop the DPL RT Service before changing **esw_multiport** settings.
	Reported in version: 1.2.0
1141	Description: Debugging packets from the second wire port P1 is not supported.
	Workaround: Use wire port P0 for debugging packets.
	Reported in version: N/A

8.11.4.6 6. DPL Compiler Known Issues

The following are known limitations of the DPL Compiler.

Reference	Issue
DPLCOMP-1848	Description: The fields **ipv4.protocol** and **ipv6.next_header** cannot be used as match keys in an **nv_add_entry** table.
	Workaround: Use a constant value (e.g., **NV_TCP_PROTOCOL**) to define the protocol context instead of using the protocol field as a variable key.
	Reported in version: 1.3.0
DPLCOMP-2026	Description: GRE encapsulation does not set the GRE key properly using a runtime parameter.
	Workaround: None
	Reported in version 1.4.0
DPLCOMP-2052	Description: Load failure with syndrome 0x178645: Multiple flex parser nodes with same field layout (bit offset and length for the selection field) may fail to load.
	Workaround: None
	Reported in version 1.5.0
DPLCOMP-2153	Description: Missing support for IPsec UDP tunnel mode.
	Workaround: None
	Reported in version 1.6.0

8.11.5 DPL System

This document provides a high-level overview of the NVIDIA® DOCA Pipeline Language (DPL) solution, which enables programmable packet processing on NVIDIA® BlueField® DPUs and SuperNICs using P4-based semantics.

8.11.5.1 Introduction

DPL is a domain-specific programming framework based on the P4-16 language. While reusing P4 syntax, DPL introduces NVIDIA-specific semantics and architecture tailored for the BlueField platform. Rather than relying on C-based APIs, developers use DPL to define packet parsing, matching, and processing behavior directly in a high-level, declarative manner. This speeds up development and simplifies pipeline optimization.

A deployed DPL program is dynamically controlled using P4Runtime, an open-source, gRPC-based control plane interface and with a custom shared memory interface for high performance situations. This allows rules and configurations to be updated at runtime without recompilation. Although DPL is compatible with any P4Runtime-compliant controller, future support for vendor-specific extensions is planned. The DPL compiler also generates optimized, ready-to-use APIs that simplify integration with custom control applications (for example, DPDK-based implementations). These generated APIs link the user's application to a low-latency, BlueField-optimized interface that exposes control and data-plane operations in a structured manner, enhancing rule insertion rates, configuration update performance, and overall usability.

The DPL toolkit includes:

- A compiler for building BlueField optimized pipelines
- A runtime service to manage hardware deployment and control-plane interaction
- Debugging and inspection developer tools for introspecting the pipeline

Together, these components deliver a complete programmable pipeline platform optimized for BlueField devices.

8.11.5.2 BlueField Network Configuration

BlueField-3 combines a ConnectX networking ASIC with an embedded Arm subsystem, providing high-performance networking and programmable packet processing. It supports offloading and acceleration of SDN, storage, and security functions.

8.11.5.2.1 DOCA Device Types

DPL supports two kinds of DOCA devices:

- Local device – A PF, VF, or SF that is locally exposed to the system (host or BlueField).
- Netdev representor – A proxy network interface on BlueField for a device on the host side. Representors allow the DPU to control and inspect traffic flowing to/from host-side devices.

8.11.5.2.2 Representor Naming

- Uplink Representor – **p<port_number>**
- PF Representor – **pf<port_number>hpf**
- VF Representor – **pf<port_number>vf<function_number>**
- SF Representor – **en<domain>f<function>pf<port_number>sf<function_number>**

⚠ The host function's MTU must be smaller than that of both its uplink and representor.

8.11.5.3 DPL Architecture

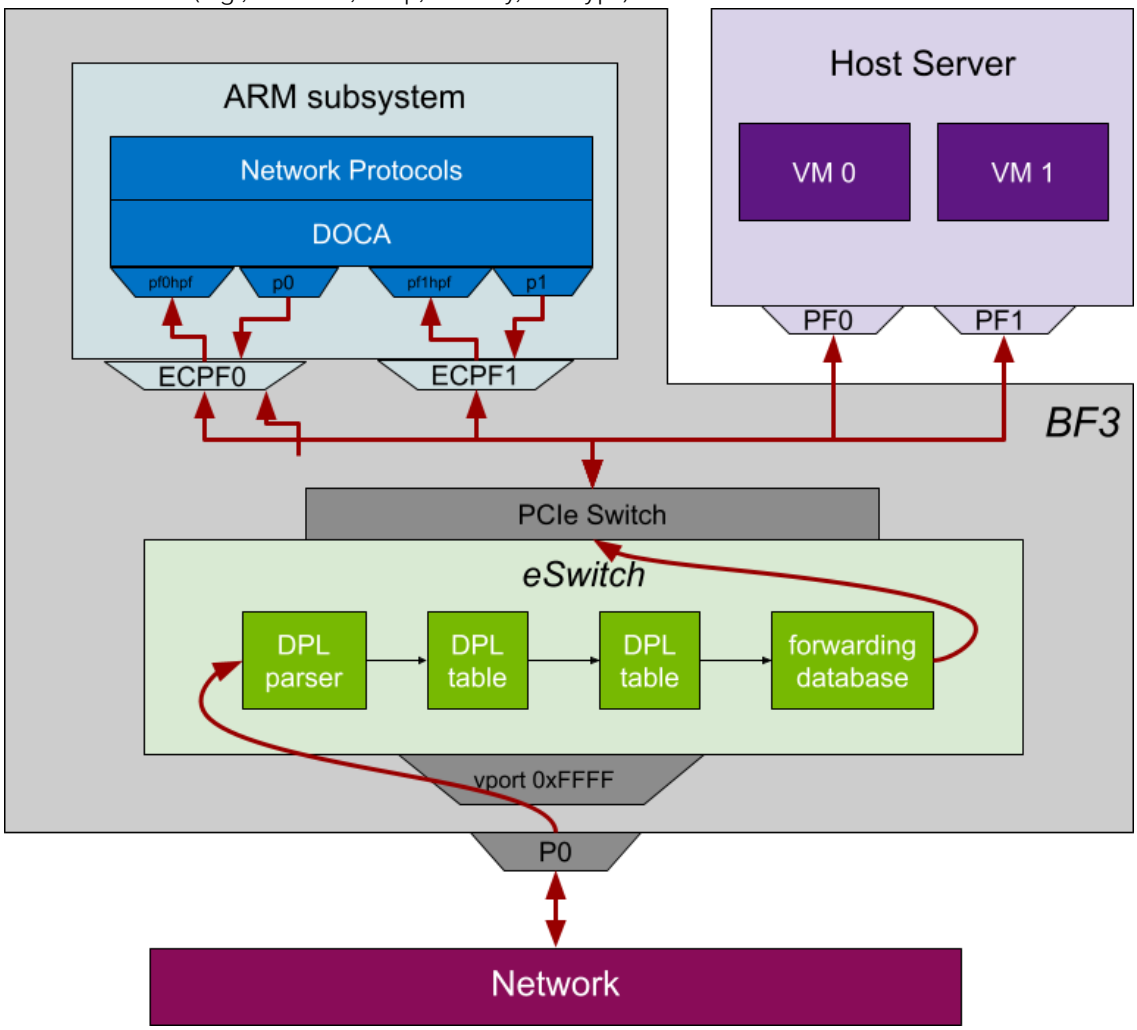
In DPU mode, the BlueField owns NIC resources, and all host traffic flows through a virtual switch on the Arm subsystem before reaching the host.

Key characteristics:

- Host-side NIC drivers load only after the BlueField completes NIC configuration
- BlueField allocates all NIC memory (ICM)
- BlueField configures the embedded switch, and all host traffic is intercepted

A compiled DPL pipeline executes within the embedded switch (eSwitch) and consists of:

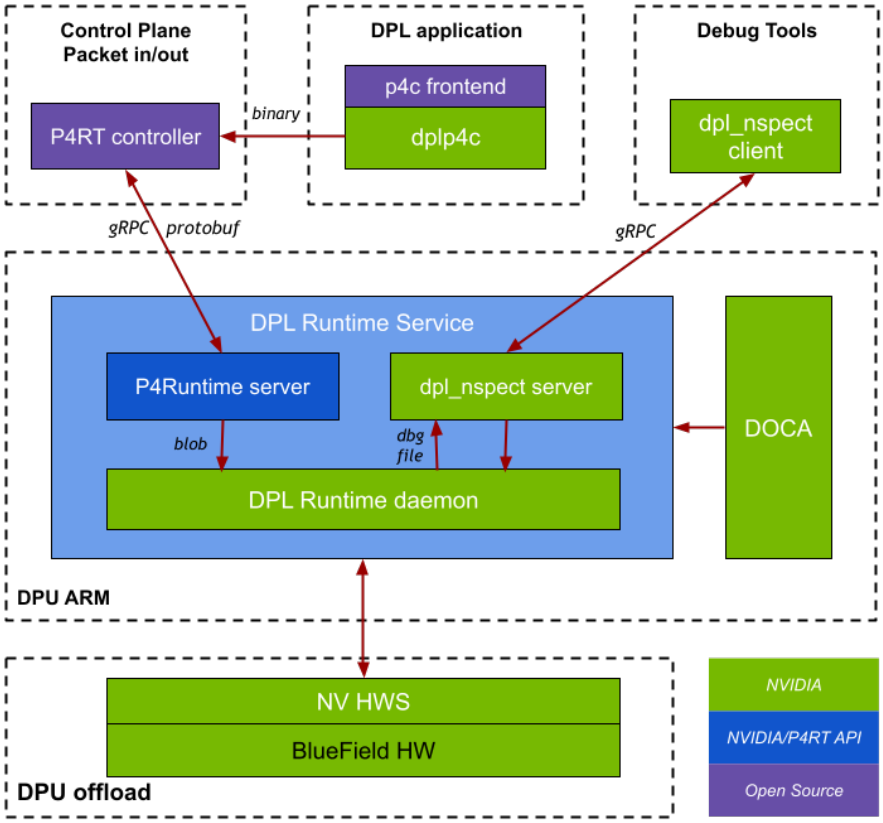
- A programmable packet parser (parse graph)
- Match/action tables
- Packet actions (e.g., forward, drop, modify, encrypt)



- **pf0hpf** and **pf1hpf** are the representors facing the x86 host
- **pf0** and **pf1** are the representors facing the network

8.11.5.4 DPL Components

The DPL solution on the BlueField DPU consists of multiple components working in tandem. These components are outlined below and shown in the following diagram:



8.11.5.4.1 DPL Compiler

The compiler, typically run from a host development system, generates:

- P4Info file – Target-independent protobuf description used by controllers
- Device config (**.dplconfig**) – Binary blob for loading into BlueField hardware
- Optional debug file – Used by inspection debug tools

P4 language reference: P4-16 v1.2.4 Documentation.

8.11.5.4.2 P4Runtime Controller

The P4Runtime Controller manages real-time updates and control of the pipeline. It must conform to P4Runtime v1.3.0 and communicates with BlueField using gRPC over TCP (default port 9559). Capabilities include:

- Installing/removing rules
- Querying table state
- Handling packet-in/out traffic

8.11.5.4.3 DPL Nspect

DPL Nspect is a host-side CLI tool for:

- Viewing table contents
- Debugging DPL pipelines
- Tracing packet path and action outcomes

It communicates with the runtime service's internal debug server.

8.11.5.4.4 DPL Runtime Service

8.11.5.4.4.1 DPL Runtime Daemon

The runtime daemon (**dpl_rtd**) runs on the BlueField and:

- Hosts the gRPC P4Runtime server
- Processes incoming controller commands
- Loads compiled pipeline artifacts
- Interfaces with the DOCA HWS driver
- Manages parse graphs, match tables, and state
- Provides debug and monitoring services

8.11.5.4.4.2 P4Runtime Server

Implements gRPC endpoints for standard P4Runtime protocol messages. Converts P4Info operations into hardware actions using the DOCA HWS driver. The server is hosted within the DPL Runtime Service.

8.11.5.4.4.3 DPL Nspect Server

A backend service in the runtime daemon that handles debug and monitoring traffic. Communicates with the Nspect client and debugger tools over gRPC.

8.11.5.4.5 NV HWS Driver

The NV HWS Driver provides low-level APIs for pipeline construction and table management in BlueField hardware. The DPL compiler and runtime interface with this driver to install flows, modify match criteria, and define actions.

8.11.5.4.6 eSwitch Management

Developers are free to manage the eSwitch using tools like:

- OVS (e.g., DOCA OVS)
- DPDK-based apps
- DOCA management interfaces

DPL does not constrain how virtual interfaces or bridges are configured.

8.11.6 DPL Runtime Service

This page explains how to deploy, configure, and operate the DOCA Pipeline Language (DPL) Runtime Service on NVIDIA® BlueField® DPUs and SuperNICs. The service enables runtime management of data plane pipelines compiled using DPL.

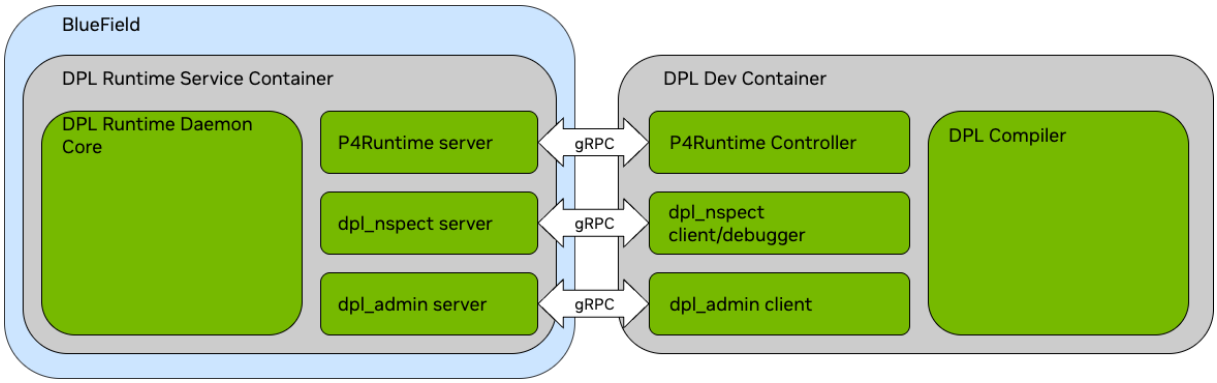
8.11.6.1 Introduction

The DPL Runtime Service programs and manages the DPU / NIC datapath at runtime. It supports dynamic rule insertion, packet monitoring, and forwarding logic using both standard and proprietary APIs.

The service consists of the following core components:

Component	Description
Runtime core	Manages requests from all interfaces and programs hardware resources
P4Runtime server	gRPC server implementing the P4Runtime 1.4.1 API for remote pipeline configuration and table control.
Nspect server	Provides runtime visibility and debugging through the `dpl_nspect` client
Admin server	Offers administrative and configuration control over the daemon via gRPC

High-level system illustration:



8.11.6.2 Requirements

8.11.6.2.1 Supported Platforms

- DPL is supported on the following BlueField SKUs:

NVIDIA SKU	Legacy OPN	PSID	Description
900-9D3B6-00CV-AA0	N/A	MT_0000000884	BlueField-3 B3220 P-Series FHHL DPU; 200GbE (default mode) / NDR200 IB; Dual-port QSFP112; PCIe Gen5.0 x16 with x16 PCIe extension option; 16 Arm cores; 32GB on-board DDR; integrated BMC; Crypto Enabled
900-9D3B6-00SV-AA0	N/A	MT_0000000965	BlueField-3 B3220 P-Series FHHL DPU; 200GbE (default mode) / NDR200 IB; Dual-port QSFP112; PCIe Gen5.0 x16 with x16 PCIe extension option; 16 Arm cores; 32GB on-board DDR; integrated BMC; Crypto Disabled
900-9D3B6-00CC-AA0	N/A	MT_0000001024	BlueField-3 B3210 P-Series FHHL DPU; 100GbE (default mode) / HDR100 IB; Dual-port QSFP112; PCIe Gen5.0 x16 with x16 PCIe extension option; 16 Arm cores; 32GB on-board DDR; integrated BMC; Crypto Enabled
900-9D3B6-00SC-AA0	N/A	MT_0000001025	BlueField-3 B3210 P-Series FHHL DPU; 100GbE (default mode) / HDR100 IB; Dual-port QSFP112; PCIe Gen5.0 x16 with x16 PCIe extension option; 16 Arm cores; 32GB on-board DDR; integrated BMC; Crypto Disabled

- These NVIDIA DPUs are supported with DOCA on the host
- DPL is supported on the following NIC SKU:

NVIDIA SKU	Legacy OPN	PSID	Description
900-9X91Q-00CN-ST0	N/A	-	NVIDIA ConnectX-9 C9240 HHHL SuperNIC, Dual-port 400GbE (default) / 400Gb/s IB, QSFP112, PCIe 6 x16 with x16 PCIe SocketDirect extension option, Crypto Enabled, Secure Boot Enabled, Tall bracket

8.11.6.2.2 Hardware Prerequisites

- For DPU system requirements, see DPU's hardware user guide.
- For SuperNIC system requirements, see adapter's user guide.

8.11.6.2.3 Software Prerequisites

8.11.6.2.3.1 DPU Software Prerequisites

Component	Minimum Version
Ubuntu OS	24.04
DOCA BFB Bundle	3.2.0+

Component	Minimum Version
DOCA Networking	3.2.0+
Firmware	32.47.1000+

Firmware is included in the BFB bundle available on the NVIDIA DOCA Downloads page. Use **bfb-install** to install the bundle. For example:

Example of BFB installation

```
bfb-install --bfb bf-bundle-x.x.x-x_x.x_ubuntu-24.04_prod.bfb --rshim rshim0
```

8.11.6.2.3.2 SuperNIC Software Prerequisites

Component	Minimum Version
Ubuntu OS	24.04
DOCA Networking	3.2.0+
Firmware	82.47.0000+

Firmware can be found on the NVIDIA DOCA Downloads page.

Helpful resources:

- DOCA Installation Guide for Linux
- Setting Up DPU Management Access and Updating BlueField-Bundle
- DOCA Container Deployment Guide

8.11.6.2.4 Firmware Settings

The following firmware configuration is required and applied by the **dpl_system_setup.sh** script documented at DPL Container Deployment:

```
FLEX_PARSER_PROFILE_ENABLE=4
PROG_PARSE_GRAPH=true
SRIOV_EN=1  # Required only if using VFs
```

To enable multi-port eSwitch mode, also set:

```
LAG_RESOURCE_ALLOCATION=1
```

8.11.6.3 Supported Port Forwarding

The following table lists the possible ingress and egress ports for a given packet that is processed by a BlueField (DPU mode) pipeline/SuperNIC pipeline:

Ingress Port	Egress Port			
	Wire Port P0	pf0hpf	pf0vf_n	pf0vf_m
Wire port P0	Allowed	Allowed	Allowed	Allowed
pf0hpf	Allowed	Disabled	Allowed	Allowed
pf0vf_n	Allowed	Allowed	Disabled	Allowed
pf0vf_m	Allowed	Allowed	Allowed	Disabled

All forwarding combinations supported for SR-IOV VFs are also supported for scalable functions (SFs).

8.11.6.4 Deploying the DPL Runtime Service

8.11.6.4.1 Prerequisites

- BlueField must be in DPU Mode.
- SR-IOV or SFs must be enabled and created on the host.
- DOCA must be installed on the host and BlueField.
- Ensure `doca-rt-service` configuration files are available. Refer to DPL Service Configuration for details.

8.11.6.4.2 Installation

Refer to the DPL Container Deployment for detailed steps on deploying the DPL Runtime Service container.

If the DPL Runtime Service container is to be deployed on the host, refer to DOCA Container Deployment Guide (DOCA Services for Host).

8.11.6.5 DPL Dev Container

The `dpl-dev` container provides essential developer tools for compiling, loading, and debugging DPL programs, including:

- DPL Compiler
- Nspect CLI
- Admin CLI
- DPL Debugger
- Runtime Controller SDK

Refer to the DPL Installation Guide for installation instructions.

8.11.6.6 Proceeding with DPL

Once the **dpl-rt-service** is running and the **dpl-dev** container is installed, you can:

1. Compile your DPL programs. Refer to Compiling DPL Applications.
2. Load the compiled program into the runtime. Refer to Loading DPL Applications.
3. (Optional) Use DOCA Pipeline Language Developer Tools for debugging and verification.

8.11.6.7 Implementing Your Own P4Runtime Controller

Advanced users may implement their own controller to communicate with the DPL Runtime via P4Runtime 1.3.0 over gRPC (TCP port **9559**).

Resources:

- Protobuf definitions available at **p4lang/p4runtime** GitHub
- Local Path (container): **/opt/mellanox/third_party/dpl_rt_service/p4runtime/**
- Toolchain: Use standard **gRPC** + **protobuf** toolchains (e.g., IN C++)

8.11.6.8 DPL Container Deployment

8.11.6.8.1 1. Introduction

The DPL Runtime Service supports Bluefield devices (DPU mode) and ConnectX-9 (Host mode).

> There are few differences when preparing the system for running the DPL Runtime Service on DPU mode (BlueField) vs Host mode (ConnectX-9), these differences are outlined in the sections below where applicable.

When deploying on the host, please refer to the instructions in section "DOCA Services for Host" in the DOCA Container Deployment Guide.

8.11.6.8.2 2. BlueField Preparations

This section is specific to DPU mode when running on BlueField devices.

8.11.6.8.2.1 2.1. Setting BlueField to DPU Mode

BlueField must run in DPU mode to use the DPL Runtime Service. For details on how to change modes, refer to BlueField Modes of Operation documentation.

8.11.6.8.2.2 2.2. Determining Your BlueField Variant

Your BlueField may be installed in a host server or it may be a standalone server.

If your BlueField is a standalone server, ignore the parts that mention the host server or SR-IOV. You may still use Scalable Functions (SFs) if your BlueField is a standalone server.

8.11.6.8.2.3 2.3. Setting Up DPU Management Access and Updating BlueField-Bundle

These pages provide detailed information about DPU management access, software installation, and updates:

- Host-side Interface Configuration
- BF-Bundle Installation and Upgrade
- NVIDIA DOCA Downloads

> ⚠ Systems with a host server typically use RShim (i.e., the **tmfifo_net0** interface). Standalone systems must use the OOB interface option for management access.

8.11.6.8.3 3. Device Configuration

8.11.6.8.3.1 3.1. Changing the eSwitch to switchdev mode

> ℹ Do this before creating SR-IOV Virtual functions. In case Virtual Functions already exist for the interface, remove them before trying to change the mode.

The DPL Runtime Service can only start if the eSwitch is in switchdev mode. If it's not, an error will be logged on startup and the process will exit.

If the platform is Bluefield in DPU mode, run this command in the DPU shell, otherwise, (e.g. ConnectX-9), use the host shell.
Your Bluefield DPU may be pre-configured in switchdev mode after the bfb installation. If this is the case, this step may be unnecessary.

Find the PCI address of the interface that you'd like to use with the DPL Runtime Service and use the following command (replace the pci/<addr> part with correct values)

3.1.1.1.1. Example

```
sudo devlink dev eswitch set pci/0000:03:00.0 mode switchdev
```

Here are a few options for commands that may help you find your PCI address:

- **lspci -D**
- **mst status -v**
- **ip -d link**
- **ethtool -i <interface name>**

> ⚠ **devlink** settings are not persistent across reboots.

8.11.6.8.3.2 3.2. Enabling Multiport eSwitch Mode (Optional)

> This step is optional and depends on your DPL program and setup needs.

Multiport eSwitch mode allows for traffic forwarding between multiple physical ports and their VFs/SFs (e.g., between `p0` and `p1`).

Before enabling this mode:

1. Ensure `LAG_RESOURCE_ALLOCATION` is enabled in firmware:
   Example

   ```
   sudo mlxconfig -d 0000:03:00.0 s LAG_RESOURCE_ALLOCATION=1
   ```

   > Refer to the Using mlxconfig guide for more information.

2. After reboot or firmware reset, enable `esw_multiport` mode:
   Example

   ```
   sudo devlink dev param set pci/0000:03:00.0 name esw_multiport value 1 cmode runtime
   ```

> `devlink` settings are not persistent across reboots.

8.11.6.8.3.3 3.3. Creating SR-IOV Virtual Functions

To use SR-IOV, first create Virtual Functions (VFs) on the host server:

3.3.1.1.1. Example

```
sudo -s  # enter sudo shell
echo 4 > /sys/class/net/eth2/device/sriov_numvfs
exit     # exit sudo shell
```

> Entering a sudo shell is necessary because `sudo` only applies to the `echo` command, and not the redirection (`>`), which would otherwise result in "Permission denied."

This example creates 4 VFs under Physical Function `eth2`. Adjust the number as needed.

> If a PF already has VFs and you'd like to change the number, first set it to `0` before applying the new value.

8.11.6.8.3.4 3.4. Creating Scalable Functions (Optional)

> This step is optional and depends on your DPL program and setup needs.

For more information, see the BlueField Scalable Functions, TODO: CX9.

If you create SFs, refer to their representors in the configuration file.

8.11.6.8.4 4. Installing the DPL Runtime Service

8.11.6.8.4.1 4.1. Downloading Container Resources from NGC

Start by downloading and installing the ngc-cli tools.
For example:

- For DPU mode, download the Arm **ngc-cli** tool:
  Example

```
wget --content-disposition https://api.ngc.nvidia.com/v2/resources/nvidia/ngc-apps/ngc_cli/versions/4.5.2/
files/ngccli_arm64.zip -O ngccli_arm64.zip
unzip ngccli_arm64.zip
```

- For Host mode, download the appropriate **ngc-cli** tool for your system architecture:
  Example for x86_64

```
wget --content-disposition https://api.ngc.nvidia.com/v2/resources/nvidia/ngc-apps/ngc_cli/versions/4.5.2/
files/ngccli_linux.zip -O ngccli_linux.zip
unzip ngccli_linux.zip
```

Once the **ngc-cli** tool has been downloaded, use it to download the latest dpl_rt_service resources:

```
./ngc-cli/ngc registry resource download-version "nvidia/doca/dpl_rt_service"
```

This creates a directory in the format **dpl_rt_service_va.b.c-docax.y.z** . Where **a.b.c** is the DPL Runtime Service version number, and **x.y.z** is the DOCA version number (e.g., **dpl_rt_service_v1.2.0-doca3.1.0**).

> You can find available versions at NGC Catalog.

> Each release includes a **kubelet.d** YAML file that is used by the **dpl_rt_service_ctl.sh** script for retrieving the correct container image for either DPU or Host mode.

8.11.6.8.4.2 4.2. Running the Preparation Script

Run the **dpl_system_setup.sh** script to configure the system:

- For DPU mode:

```
cd dpl_rt_service_va.b.c-docax.y.z
chmod +x ./scripts/dpl_system_setup.sh
sudo ./scripts/dpl_system_setup.sh
sudo systemctl restart kubelet.service
sudo systemctl restart containerd.service
```

For DPU mode, restarting **kubelet** and **containerd** is required whenever **hugepages** configuration changes for the changes to take effect.

- For Host mode, specify the ConnectX device(s) that should be configured for DPL use using the **--dev** option (this option can be repeated):

```
cd dpl_rt_service_va.b.c-docax.y.z
chmod +x ./scripts/dpl_system_setup.sh
sudo ./scripts/dpl_system_setup.sh --dev 0000:08:00.0
```

The **dpl_system_setup.sh** script will perform the following:

- Configures **mlxconfig** values:
  - **FLEX_PARSER_PROFILE_ENABLE=4**
  - **PROG_PARSE_GRAPH=true**
  - **SRIOV_EN=1**
- Enables SR-IOV
- Sets up initial DPL Runtime Service configuration folder at **/etc/dpl_rt_service/**
- Configures hugepages

Please note that the **dpl_system_setup.sh** script takes optional arguments to control the hugepages.

For DPU mode, if you set it larger than 4GB of hugepages, you also have to modify **dpl_rt_service.yaml** with a higher limit for **spec->resources->limits->hugepages-2Mi**

8.11.6.8.4.3 4.3. Editing the Configuration Files

Create device(s) configuration file based on the provided template config file. See DPL Service Configuration for details.

For example:

4.3.1.1.1. Example

```
sudo cp /etc/dpl_rt_service/devices.d/NAME.conf.template /etc/dpl_rt_service/devices.d/1000.conf
# Then update /etc/dpl_rt_service/devices.d/1000.conf as needed.
sudo vim /etc/dpl_rt_service/devices.d/1000.conf
```

You must create at least one device configuration file. Otherwise, the DPL Runtime Service Container will not be able to start.

8.11.6.8.4.4 4.4. Firewall Configuration to Open gRPC Server Ports

The DPL Runtime Service has several gRPC servers, each listening on a dedicated TCP port, supporting a corresponding DPL Developer tool. It is critical to make sure that these ports are accessible from the system(s) you plan to run the DPL Developer tools on as tThe tools will connect to the DPL Runtime Service using the corresponding tool's gRPC server TCP port.

The ports are configurable (see **server_tcp_port** settings at DPL Service Configuration). By default, they have the following values:

gRPC server	TCP Port
P4 Runtime	9559
DPL Admin	9600
DPL Nspect/Debugger	9560

4.4.1.1.1. Example for allowing the ports on RHEL-9

```
sudo firewall-cmd --permanent --add-port=9559/tcp
sudo firewall-cmd --permanent --add-port=9600/tcp
sudo firewall-cmd --permanent --add-port=9560/tcp
sudo firewall-cmd --reload

# List Configurations to confirm ports were allowed.
sudo firewall-cmd --list-all
```

8.11.6.8.4.5 4.5. Starting the DPL Runtime Service Container

Once your configuration files are ready, use the **dpl_rt_service_ctl.sh** script to start the container:

Before running the script for the first time, user must grant it execution rights:

```
sudo chmod +x ./scripts/dpl_rt_service_ctl.sh
```

```
sudo ./scripts/dpl_rt_service_ctl.sh --start
```

For DPU mode, the script will copy the YAML file into the **/etc/kubelet.d/** directory, which will trigger automatic creation and start of DPL RT Service Pod and container.

For Host mode, the script will start a Docker container named **dpl-rt-service**.

Allow a few minutes for the container to start. To monitor status:

- For DPU mode:

- - Check logs:

```
sudo journalctl -u kubelet --since -5m
```

- List images:

```
sudo crictl images
```

- List pods:

```
sudo crictl pods
```

- For Host mode:
  - Check logs:

```
sudo docker logs dpl-rt-service
```

  - List images:

```
sudo docker images
```

- View runtime logs:

```
/var/log/doca/dpl_rt_service/dpl_rtd.log
```

If the container fails to start due to configuration errors, then the log file at **/var/log/doca/dpl_rt_service/dpl_rtd.log** may be empty or missing the relevant error logs. In this case, you can view logs with the relevant errors using the relevant tool:

- For DPU:

```
sudo crictl logs $(sudo crictl ps -a | grep dpl-rt-service | awk '{print $1}')
```

- For Host:

```
sudo docker logs dpl-rt-service
```

8.11.6.8.4.6 4.6. Stopping the DPL Runtime Service Container

Stop the container by using the **dpl_rt_service_ctl.sh** script:

```
sudo ./scripts/dpl_rt_service_ctl.sh --stop
```

For DPU mode, the script will remove the YAML file from the **/etc/kubelet.d/** directory.

For Host mode, the script will stop the Docker container named **dpl-rt-service**.

To confirm the pod is gone (this might take a few seconds to complete):

- For DPU:

```
sudo crictl pods | grep dpl-rt-service
```

- For Host:

```
sudo docker ps | grep dpl-rt-service
```

8.11.6.8.4.7 4.7. Restarting the DPL Runtime Service After Configuration Changes

Once the DPL Runtime Service container is up and running, any change to any file under the **/etc/dpl_rt_service/** configuration folder requires restarting the container in order for the new changes to take effect.

Perform the following steps to restart the container:

1. Stop the container.

```
sudo ./scripts/dpl_rt_service_ctl.sh --stop
```

2. Wait for the container to stop.
3. Start the container:

```
sudo ./scripts/dpl_rt_service_ctl.sh --start
```

8.11.6.8.4.8 4.8. End-to-End Installation Steps

⚠ Replace device IDs and filenames as appropriate for your setup.

- DPU example:

```
# Download NGC CLI tool:
wget --content-disposition https://api.ngc.nvidia.com/v2/resources/nvidia/ngc-apps/ngc_cli/versions/4.5.2/
files/ngccli_arm64.zip -O ngccli_arm64.zip
unzip ngccli_arm64.zip

# Download the DPL Runtime Service Resources bundle:
./ngc-cli/ngc registry resource download-version "nvidia/doca/dpl_rt_service"

# Prepare DPU and restart services:
cd dpl_rt_service_va.b.c-docax.y.z
chmod +x ./scripts/dpl_system_setup.sh
sudo ./scripts/dpl_system_setup.sh
sudo systemctl restart kubelet.service
sudo systemctl restart containerd.service

# Create a device configuration file with relevant interfaces info:
sudo cp /etc/dpl_rt_service/devices.d/NAME.conf.template /etc/dpl_rt_service/devices.d/1000.conf
sudo vim /etc/dpl_rt_service/devices.d/1000.conf

# Launch the Pod and container:
sudo ./scripts/dpl_rt_service_ctl.sh --start
```

- Host example:

```
# Download NGC CLI tool:
wget --content-disposition https://api.ngc.nvidia.com/v2/resources/nvidia/ngc-apps/ngc_cli/versions/4.5.2/
files/ngccli_linux.zip -O ngccli_linux.zip
unzip ngccli_linux.zip

# Download the DPL Runtime Service Resources bundle:
./ngc-cli/ngc registry resource download-version "nvidia/doca/dpl_rt_service"

# Prepare DPU and restart services:
cd dpl_rt_service_va.b.c-docax.y.z
```

```
chmod +x ./scripts/dpl_system_setup.sh
sudo ./scripts/dpl_system_setup.sh --dev 0000:08:00.0

# Create a device configuration file with relevant interfaces info:
sudo cp /etc/dpl_rt_service/devices.d/NAME.conf.template /etc/dpl_rt_service/devices.d/1000.conf
sudo vim /etc/dpl_rt_service/devices.d/1000.conf

# Launch the Docker container:
sudo ./scripts/dpl_rt_service_ctl.sh --start
```

8.11.6.8.5 5. Troubleshooting

For additional troubleshooting steps and deeper explanations, refer to BlueField Container Deployment Guide.

Checkpoint	Command
View recent kubelet logs (DPU only)	`sudo journalctl -u kubelet --since -5m`

View logs of the **dpl-rt-service** container (helpful if **/var/log/doca/dpl_rt_service/dpl_rtd.log** is missing or incomplete)	• For DPU:
	`sudo crictl logs $(sudo crictl ps -a | grep dpl-rt-service | awk '{print $1}')`
	• For Host:
	`sudo docker logs dpl-rt-service`

List pulled container images	• For DPU:
	`sudo crictl images`
	• For Host:
	`sudo docker images`

List all created pods (DPU only)	`sudo crictl pods`

List running containers	• For DPU:
	`sudo crictl ps`
	• For Host:
	`sudo docker ps`

View DPL service logs	`/var/log/doca/dpl_rt_service/dpl_rtd.log`

Make sure the following conditions are met before or during deployment:

- VFs were created before deploying the container (if using SR-IOV).
- All required configuration files exist under `/etc/dpl_rt_service/`, are correctly named, and include valid device IDs.
- Network interface names and MTU settings match the physical and virtual network topology.
- Firmware is up to date and matches DOCA compatibility requirements.
- For DPU mode, BlueField is operating in the correct mode (DPU mode) using `sudo mlxconfig -d <pci-device> q`.

8.11.6.9 DPL Service Configuration

8.11.6.9.1 1. Introduction

The DPL Runtime Service utilizes three distinct types of configuration files. These files adhere to a simplified INI-style format, which supports repeated sections (e.g., `[INTERFACE]`) and uses `#` for comments rather than `;`.

The available configuration files are as follows:

- DPL Device Configuration: Located at `/etc/dpl_rt_service/devices.d/<device-id>.conf`.
- Service Daemon Configuration: Located at `/etc/dpl_rt_service/dpl_rt.conf`.
- System Configuration: Located at `/etc/dpl_rt_service/system.conf`.

The `dpl_dpu_setup.sh` script installs default (recommended) daemon and system configuration files. However, users must create their own DPL Device configuration file based on the provided template, which is also installed by the setup script.

> ⚠ Any changes made to files within the `/etc/dpl_rt_service/` directory require a restart of the DPL Runtime Service Container to take effect. Please refer to the section "Restarting the DPL Runtime Service After Configuration Changes" for instructions.

8.11.6.9.2 2. DPL Device Configuration

Path: `/etc/dpl_rt_service/devices.d/<device-id>.conf` (e.g., `/etc/dpl_rt_service/devices.d/1000.conf`).

This file defines the device, its interfaces, and the mapping to DPL Port IDs used in DPL programs. A configuration template is available at `/etc/dpl_rt_service/devices.d/NAME.conf.template`.

> ⚠ When using SFs or SR-IOV VFs, ensure the configuration references their representor interfaces, not the SF/VF interfaces themselves.

8.11.6.9.2.1 2.1. DPL Port ID Mapping

Each physical or virtual interface is assigned a logical DPL Port ID (`dpl_logical_port_id`). Consistency between the DPL program and this configuration is mandatory.

Supported interface types include:

- Uplink netdev interface (e.g., `p0`).
- PF representor (e.g., `pf0hpf`).
- VF representor (e.g., `pf0vf0`).
- SF representor (e.g., `en3f0pf0sf1`).

Unless multiport eswitch is enabled, all listed interfaces must belong to the same uplink port.

8.11.6.9.2.2 2.2. DPL Port ID Rules

The following rules apply to `dpl_device_id` and `dpl_logical_port_id` values:

Condition	Requirement
Reserved value	`UINT32_MAX`
DPL Device ID	Must be a positive integer
DPL Interface ID	Integer between `0` and `UINT32_MAX`
Uplink ports per config	Exactly one (unless multiport eswitch is enabled)

8.11.6.9.2.3 2.3. Example DPL Device Configuration File

The following is an example of a valid device configuration file.

```
# [DEVICE] section: Must appear only once.
[DEVICE]
# The DPL Device ID used by the controller to manage tables.
dpl_device_id=1000
# Cache counter timeout to decrease HW accesses.
dpl_counter_cache_timeout=0
# Counter polling interval for idle-timeout (in seconds).
idle_timeout_polling_interval=2
# Polling interval for TCP state objects from HW (in milliseconds).
tcp_state_polling_interval_msec=200

# [P4_RT_CONTROLLER] section: Must appear only once.
[P4_RT_CONTROLLER]
# DPL Port ID for traffic originating from the controller.
p4_controller_port_id=9876

# [INTERFACE] section: Repeated for each DPL Port (network interface).
[INTERFACE]
interface=p0
dpl_logical_port_id=0
mtu=1514
# mac=00:00:00:00:00:00

[INTERFACE]
interface=pf0hpf
dpl_logical_port_id=65535
mtu=1514

[INTERFACE]
interface=pf0vf0
dpl_logical_port_id=1
mtu=1514
```

8.11.6.9.2.4 2.4. Section DEVICE

This section defines the attributes of the DPL Device, which consists of multiple ports and supports DPL program loading. It must appear exactly once per device `.conf` file.

- `dpl_device_id` The Device ID used by controller applications to identify the target device for connection and program management.

- **dpl_counter_cache_timeout** This parameter reduces hardware accesses for counter reads. When the cache expires, a hardware access occurs upon the next request. Note that enabling this may result in the retrieval of outdated counter values.
- **idle_timeout_polling_interval** Sets the polling interval (in seconds) for reading entry counters from hardware and identifying stale entries (those with no traffic for a period exceeding the defined threshold). This interval affects both Entry Timeout and Delayed Counter Statistics. Larger values impact the accuracy of timeout notifications and the refresh rate of counter statistics.
- **tcp_state_polling_interval_msec** Defines the interval (in milliseconds) for reading TCP state objects from the hardware.

8.11.6.9.2.5 2.5. Section P4_RT_CONTROLLER

This section configures communication with P4 Controllers.

p4_controller_port_id P4 controller applications can send packets ("Packet Out") to the DPL Runtime Service via RPC messages. These packets are assigned a port ID equal to **p4_controller_port_id**. A DPL programmer can identify traffic originating from a P4 Controller by comparing **std_meta.ingress_port** to this value.

8.11.6.9.2.6 2.6. Section INTERFACE

This section defines the attributes of a port belonging to the device. This section may be repeated for multiple interfaces.

- **interface** : The net-device interface name on the system.
- **dpl_logical_port_id** : The ID used to reference the port in DPL programs and table updates. It is used for matching ingress traffic and directing egress traffic. Programmers can verify the source port by comparing **std_meta.ingress_port** to this value.
- **mtu** : Defines the Ethernet frame size (MTU).

  Currently, this field is ignored.

- **mac** : Defines the MAC address.

  Currently, this field is ignored.

8.11.6.9.3 3. Service Daemon Configuration

Path: **/etc/dpl_rt_service/dpl_rt.conf** .

This file configures core behaviors for the DPL Runtime Service, including logging and gRPC server binding addresses.

8.11.6.9.3.1 3.1. Section LOGGING

- **log_file_path** : Defines the path to the **dpl_rtd** log file.

- **log_level** : Sets the default log level upon startup. Supported values (case-insensitive) include: **DISABLE** , **CRITICAL** , **ERROR** , **WARNING** , **INFO** , **DEBUG** , and **TRACE** .

  ⚠ Log levels can be changed dynamically via the DPL Admin client but will not persist across restarts unless updated in this file.

8.11.6.9.3.2 3.2. RPC Server Sections

The daemon supports three distinct gRPC servers:

1. **[P4RT_RPC_SERVER]** : Listens for P4Runtime client connections. A sample open-source client is included in the DPL Dev container.
2. **[DPL_ADMIN_RPC_SERVER]** : Listens for **dpl_admin** tool connections.
3. **[DPL_NSPECT_RPC_SERVER]** : Listens for DPL Debugger tool connections.

3.2.1. Common gRPC Server Parameters

The following parameters apply to all three server sections:

- **server_address** : The TCP binding address. Use **[::]** (IPv6 ANY) to allow connections from all interfaces (including IPv4), or specify a specific IP address.
- **server_tcp_port** : The TCP binding port. If a non-default port is configured, client applications must specify this port when connecting.

3.2.2. TLS Authentication

TLS authentication can be enabled independently for each server. To enable TLS, the following mutually inclusive parameters must be added to the relevant server section:

- **server_cert** : Path to the **server.crt** file.
- **ca_cert** : Path to the **ca.crt** file.
- **server_key** : Path to the **server.key** file.

ℹ Make sure to provide the client side with the corresponding parameters.

Example TLS Setup

```
# OpenSSL PKI Setup (Root CA, Server & Client Certificates)

# This guide creates a simple self-signed PKI:
# - Root CA
# - Server certificate
# - Client certificate

password=abcd

## Establish a Private Root CA
### 1. Generate CA Private Key
openssl genrsa -passout pass:$password -des3 -out ca.key 4096

### 2. Generate Self-Signed Root Certificate
openssl req -passin pass:$password -new -x509 -days 365 -key ca.key -out ca.crt \
  -subj "/C=US/ST=CA/L=PaloAlto/O=Test/OU=Test/CN=Root CA"

## Generate the Server Certificate

### 1. Create Server key
openssl genrsa -passout pass:$password -des3 -out server.key 4096

### 2. Create Certificate Signing Request (CSR)
openssl req -passin pass:$password -new -key server.key -out server.csr \
```

```
  -subj "/C=US/ST=CA/L=PaloAlto/O=Test/OU=Server/CN=localhost"

### 3. Sign the Server CSR with your Root CA
openssl x509 -req -passin pass:$password \
  -extfile <(printf "subjectAltName=DNS:localhost,IP:127.0.0.1") \
  -days 365 -in server.csr -CA ca.crt -CAkey ca.key -set_serial 01 -out server.crt

### 4. Remove passphrase for automated service use
openssl rsa -passin pass:$password -in server.key -out server.key

## Generate client certificate

### 1. Create Client Key
openssl genrsa -passout pass:$password -des3 -out client.key 4096

### 2. Create Client CSR
openssl req -passin pass:$password -new -key client.key -out client.csr \
  -subj "/C=US/ST=CA/L=PaloAlto/O=Test/OU=Client/CN=my-client"

### 3. Sign the Client CSR with your Root CA
openssl x509 -passin pass:$password \
  -req -days 365 -in client.csr -CA ca.crt -CAkey ca.key -set_serial 01 -out client.crt

### 4. Remove passphrase
openssl rsa -passin pass:$password -in client.key -out client.key
```

Transmission over a non-secure channel may expose sensitive cryptographic material, including encryption keys. Configure the gRPC server and client to use TLS with certificate validation enabled before transmitting sensitive data.

8.11.6.9.3.3 3.3. Section DPL_PACKET_IO

- **enabled** : When set to **true** , initializes the Packet IO infrastructure. This is required for P4 Controller Packet IO, the "Add Entry" feature (from DPL programs), and DPL Debugger tools.
- **extra_dpdk_eal_options** : Defines a whitespace separated list of extra DPDK EAL options for Packet IO operations (e.g for setting debug level, set **--log-level=*:debug**). Refer to the DPDK Logging Guide for details.

8.11.6.9.3.4 3.4. Section DPL_SHM

- **enabled** : When set to **true** , initializes the Shared Memory (SHM) infrastructure, required for controller applications utilizing the DPL Runtime Controller SDK.
- **ring_size** : Controls the size of SHM requests ring/queue used for communication between a dpl-rt-controller and the dpl-rt-service for passing messages to add/delete High Update Rate (HUR) tables entries. When this ring/queue is full, entry add/delete API calls will return **DOCA_ERROR_AGAIN** error code, in such cases, the client application must retry calling the same operation until it gets a different error code.
- **extra_dpdk_eal_options** : Defines a whitespace separated list of extra DPDK EAL options for SHM infrastructure (e.g for supporting running Address Sanitizer on an SHM controller applicaiton set **--base-virtaddr=0x700000000000**). Refer to the DPDK Logging Guide for details.

8.11.6.9.3.5 3.5. Example Service Daemon Configuration File

```
# Example of a possible DPL RT Service GENERAL configuration file

[LOGGING]
# Path to the dpl_rtd log file.
log_file_path=/var/log/doca/dpl_rt_service/dpl_rtd.log
# Log level for the DPL RT Service.
# Possible log_level values (case insensitive):
# DISABLE
# CRITICAL
```

```
# ERROR
# WARNING
# INFO
# DEBUG
# TRACE
log_level=INFO

[P4RT_RPC_SERVER]
server_address=[::]    # IPv6 "ANY" allows IPv4 connections
server_tcp_port=9559
## If you would like to enable TLS authentication for the gRPC server connections, uncomment the following line and
provided the required info:
#server_cert=/path/to/server.crt
#ca_cert=/path/to/ca.crt
#server_key=/path/to/server.key

[DPL_ADMIN_RPC_SERVER]
server_address=[::]    # IPv6 "ANY" allows IPv4 connections
server_tcp_port=9600
## If you would like to enable TLS authentication for the gRPC server connections, uncomment the following line and
provided the required info:
#server_cert=/path/to/server.crt
#ca_cert=/path/to/ca.crt
#server_key=/path/to/server.key

[DPL_NSPECT_RPC_SERVER]
server_address=[::]    # IPv6 "ANY" allows IPv4 connections
server_tcp_port=9560
## If you would like to enable TLS authentication for the gRPC server connections, uncomment the following line and
provided the required info:
#server_cert=/path/to/server.crt
#ca_cert=/path/to/ca.crt
#server_key=/path/to/server.key

[DPL_PACKET_IO]
# Enable Packet IO processing.
enabled=true
# Extra DPDK EAL options to pass when working with Packet IO (space-separated).
# These options are appended verbatim to the EAL arguments.
# For example, to raise the DPDK log level use: --log-level=*:debug
# See more details and possible values at https://doc.dpdk.org/guides/linux_gsg/linux_eal_parameters.html
#extra_dpdk_eal_options=--log-level=*:debug

# Buffer size for holding Packet-In packets to be sent to the P4Runtime Controller.
# Once this buffer is full, incoming Packet-In packets will be dropped by the P4Runtime Server.
# Note that a big buffer size can lead to a memory usage increase.
packet_in_buffer_size=131072

[DPL_SHM]
# Enable Shared Memory (SHM) infrastructure for High Update Rate (HUR) tables support using dpl-rt-controller SDK.
enabled=true
# Shared Memory ring size:
# The size of SHM requests ring used for communication between a dpl-rt-controller and the dpl-rt-service for
passing
# messages to add/delete High Update Rate (HUR) tables entries.
ring_size=2048

# Extra DPDK EAL options to pass when initializing the SHM infrastructure (space-separated).
# These options are appended verbatim to the EAL arguments.
# For example, to enable using Address Sanitizer on client app, use: --base-virtaddr=0x700000000000
# See more details and possible values at https://doc.dpdk.org/guides/linux_gsg/linux_eal_parameters.html
#extra_dpdk_eal_options=--base-virtaddr=0x700000000000
```

8.11.6.9.4 4. System Configuration

Path: **/etc/dpl_rt_service/system.conf** .

This file handles hardware interaction tuning for the DPL Runtime Service.

8.11.6.9.4.1 4.1. Section LOGICAL_CORES

- **packet_io_lcores** : Specifies the logical CPU core for Packet IO processing. Do not use lcore 0.
- **rule_insertion_lcores** : Specifies the logical CPU core for rule insertions/deletions. Do not use lcore 0.

> ⚠ This core must be different from cores used by any controller application utilizing the DPL Runtime Controller SDK.

8.11.6.9.4.2 4.2. Section HAL

This section configures Hardware Steering (HWS) settings.

> Do not modify settings in this section unless instructed by NVIDIA Support. These settings directly impact performance characteristics such as rule update rates and latency.

- **queue_size** : HWS queue size.
- **queues_num** : Number of HWS queues for rule insertions/deletions.
- **burst_size** : Burst size for HWS rule insertions/deletions.

8.11.6.9.4.3 4.3. Example System Configuration File

```
[LOGICAL_CORES]
# Logical CPU core for processing Packet IO (do not use lcore 0).
packet_io_lcores=1
# Logical CPU core for processing rule insertions/deletions (do not use lcore 0).
rule_insertion_lcores=2

[HAL]
queue_size=1024
queues_num=1
burst_size=32
```

8.11.7 DPL Development Container

The DPL Development Container is designed for developers building data path applications using the DOCA Pipeline Language (DPL) — NVIDIA's P4-derived language for programmable networking pipelines. DPL is supported on NVIDIA® BlueField® platforms, including DPUs and SuperNICs operating in DPU mode.

To help you get started and build effectively with DPL, refer to the following key resources:

- DPL Target Architecture – Explains the architecture that DPL programs run on, including the hardware building blocks and capabilities of BlueField
- P4 Language Support in DPL – Details which features from the P4-16 language are supported or modified within DPL
- DPL Installation Guide – Provides instructions on how to install and set up the DPL Development container
- Compiling DOCA Applications – Describes how to compile DPL source files into binary artifacts, with explanations of available compiler options
- Loading DOCA Applications – Walks through how to deploy compiled DPL binaries to BlueField hardware
- Sample DPL Applications – Offers reference programs and examples that demonstrate practical use cases and patterns for DPL development
- Inspecting DPL Programs – A step-by-step guide to use the DPL Nspect, a CLI tool used for both inspecting DPL loaded programs state (like tables, entries, keys etc.), and recording a debug session (to be used by the DPL Debugger).
- Debugging DPL Programs – A step-by-step guide to use the DPL Debugger, a GUI tool which allows both offline and runtime debugging.

- DPL Program Metrics – Explains the estimated resource usage and performance-impacting processing paths for a compiled DPL program's implementation in BlueField hardware
- DPL Performance User Guide - Guidelines and discussion for how to write performant DPL applications.

If you are new to DPL, start with the target architecture and sample applications to build context before diving into compilation or runtime deployment.

8.11.7.1 DOCA Target Architecture

This page introduces the NVIDIA DPL Model for NVIDIA® BlueField®-3 networking platforms (DPUs or SuperNICs).

8.11.7.1.1 1. Introduction

The DOCA Target Architecture is a fundamental concept in P4 programming, serving as a specification that defines:

- Programmable components (e.g., parsers, ingress control flows)
- Interfaces between hardware and the P4 software layer

This specification serves as a formal contract between the P4 program and the target hardware, ensuring compatibility and efficient utilization of the hardware's data plane capabilities.

For each hardware platform, NVIDIA provides:

- A P4-based compiler tailored to NVIDIA architecture
- A DPL hardware-specific architecture definition

This page focuses on the DPL DOCA Target Architecture, providing a detailed guide on:

- Programming BlueField-3 DPUs for customized packet processing and forwarding
- Utilizing NVIDIA's DPL compiler to interact with various interfaces and ports on BlueField devices
- Considerations for Target Architectures for future BlueField generations

While different Target Architectures may share similarities, they are generally vendor-specific and are not designed for cross-vendor portability.

8.11.7.1.2 2. ConnectX-9 Target Architecture

The default DOCA target architecture described in this document applies to the BlueField-3 DPU. The NVIDIA® ConnectX®-9 Target Architecture extends the DPL DOCA Target Architecture to ConnectX-9 SuperNIC adapters, in addition to BlueField®-3 DPUs. ConnectX-9 is optimized for AI/ML and high-scale networking workloads, operating in Host mode. DPL programs targeting ConnectX-9 use a distinct target string in the compilation flow:

- `doca-bf3` – BlueField-3 DPU
- `doca-cx9` – ConnectX-9

Unless otherwise noted, P4-level semantics (parser model, extern behavior, table semantics) are compatible between BlueField-3 and ConnectX-9. Differences are primarily in scale and in the availability of additional Extern object types. These differences will be noted in the documentation below with **CX-9 feature**.

8.11.7.1.3 3. Annotations

Annotations in the P4 programming language serve as a mechanism to attach metadata, compiler hints, and documentation to program entities without altering the core packet processing logic. They are similar to “attributes” in other programming languages. DPL specific annotations always begin with @nv_. The compiler expects these annotations to be associated with specific DPL entities. If placed on unsupported entities, the annotation is ignored and a compiler warning may be produced. For unrecognized/undocumented annotations (for example, from another target architecture), the annotation is silently ignored. The following is a list of supported DPL annotations, with further details under the specific feature:

DPL Annotation	Purpose
@nv_extract	Specifies a parser extraction point
@nv_transition	Defines a parser state transition
@nv_transition_from	Specifies which parser state to transition from
@nv_header_data_fields	Provides values for each header field in encapsulation templates
@nv_controller_metadata	Marks a struct as packet-in or packet-out controller metadata
@nv_high_entropy_bits	Specifies which bits of a match key had high entropy (for hash optimizations)
@nv_modifiable	Marks an action parameter as atomically modifiable at runtime

The following core P4 annotations are supported:

Annotation	Purpose
@name("...")	Assigns a control-plane visible name to an entity
@optional	Parameter is optional
@pkginfo(key=value)	Package metadata to be included in the p4info
@brief("...")	Short description
@description("...")	Long description

8.11.7.1.4 4. DOCA Pipeline Model

The DOCA pipeline model encompasses a set of control stages that are supported by the hardware. Each top-level object within this model is designed to receive a specific set of parameters:

8.11.7.1.4.1 4.1. Parser

The parser processes incoming packets by extracting header information.

- Operates without access to metadata during parsing

- Focuses exclusively on the packet's content and structure

Syntax:

```
parser NvDocaParser<HEADERS>(
    packet_in packet,
    out HEADERS headers);
```

8.11.7.1.4.2 4.2. Control Stage

The control stage processes metadata and applies user logic to modify packet behavior.

It manages three distinct types of metadata:

- Standard Metadata (read-only) – Provides packet-related information obtained from earlier processing stages (e.g., parser, crypto engine)
- User Metadata (read-write) – Stores application-specific variables that allow for custom processing logic
- Packet Out Metadata (read-write) – Carries additional information from the SDN controller to the DOCA pipeline when sending packets to the data plane

Syntax:

```
control NvDocaMainControl<HEADERS, USER_META, PKT_OUT_META>(
    inout HEADERS headers,
    in nv_standard_metadata_t std_meta,
    inout USER_META user_meta,
    inout PKT_OUT_META pkt_out_meta
);
```

BlueField hardware supports in-line packet modification and reparsing, eliminating the need for a separate deparser control stage. This simplifies pipeline design and removes the need for packet recirculation.

8.11.7.1.4.3 4.3. Main Package (Entry Point)

The main package acts as the entry point of every P4 program for DOCA Target Architecture.

It consists of:

- A parser (**NvDocaParser**)
- A control stage (**NvDocaMainControl**)

Syntax:

```
package NvDocaPipeline<HEADERS, USER_META, PKT_OUT_META>(
    NvDocaParser<HEADERS> parse,
    NvDocaMainControl<HEADERS, USER_META, PKT_OUT_META> main
);
```

Packet modifications are applied immediately and do not require explicit deparser logic.

8.11.7.1.5 5. DOCA Core Library

The DOCA Core Library provides a foundational set of common types and metadata structures used within the DOCA Target Architecture. These types and constants are defined in **doca_core.p4** and are commonly used across various DOCA Target Architecture interfaces, such as extern functions.

8.11.7.1.5.1 5.1. Typedefs

The following typedefs define commonly used header fields and standard metadata fields:

```
typedef bit<32> nv_logical_port_t;
typedef bit<48> nv_mac_addr_t;
typedef bit<32> nv_ipv4_addr_t;
typedef bit<128> nv_ipv6_addr_t;
typedef bit<32> nv_tunnel_id_t;
typedef bit<12> nv_vlan_id_t;
typedef bit<24> nv_vxlan_id_t;
typedef bit<20> nv_mpls_label_t;
typedef bit<8> nv_debug_cookie_t;
```

8.11.7.1.5.2 5.2. Constants

The following DOCA Target Architecture-specific constants define standard metadata key fields:

```
const bit<2> L2_TYPE_UNICAST   = 2w0;
const bit<2> L2_TYPE_MULTICAST = 2w1;
const bit<2> L2_TYPE_BROADCAST = 2w2;

const bit<2> L3_TYPE_NONE   = 2w0;
const bit<2> L3_TYPE_IPV4   = 2w1;
const bit<2> L3_TYPE_IPV6   = 2w2;

const bit<2> L4_TYPE_NONE   = 2w0;
const bit<2> L4_TYPE_TCP    = 2w1;
const bit<2> L4_TYPE_UDP    = 2w2;
const bit<2> L4_TYPE_IPSEC  = 2w3;

const bit<4> L4_TYPE_EXT_NONE  = 4w0;
const bit<4> L4_TYPE_EXT_TCP   = 4w1;
const bit<4> L4_TYPE_EXT_UDP   = 4w2;
const bit<4> L4_TYPE_EXT_ICMP  = 4w3;

const bit<2> VLAN_TYPE_NONE  = 2w0;
const bit<2> VLAN_TYPE_SVLAN = 2w1;
const bit<2> VLAN_TYPE_CVLAN = 2w2;

const bit<2> ENCAP_TYPE_NONE      = 2w0;
const bit<2> ENCAP_TYPE_L2_TUNNEL = 2w1;
const bit<2> ENCAP_TYPE_L3_TUNNEL = 2w2;
const bit<2> ENCAP_TYPE_ROCE      = 2w3;

const bit<2> IPSEC_TYPE_NONE     = 2w0;
const bit<2> IPSEC_TYPE_OVER_IP  = 2w1;
const bit<2> IPSEC_TYPE_OVER_UDP = 2w2;

const bit<8> IPSEC_SYNDROME_OK  = 8w0;
const bit<8> PSP_SYNDROME_OK    = 8w0;
```

These values are used for interpreting type codes within standard metadata.

8.11.7.1.5.3 5.3. Defines

The following macros define industry-standard values for various RFC protocol fields (e.g., Ethertype, IP protocol numbers).

- IP Protocol Numbers

```
#define NV_IPV6_HBH_OPTION   0x00  /* IPv6 Hop-by-Hop option */
#define NV_ICMP_PROTOCOL     0x01  /* ICMP */
#define NV_IPV4_PROTOCOL     0x04  /* IPv4-over-IPv4 encapsulation */
#define NV_TCP_PROTOCOL      0x06  /* TCP */
#define NV_UDP_PROTOCOL      0x11  /* UDP */
```

```
#define NV_IPV6_PROTOCOL     0x29  /* IPv6-over-IPv6 encapsulation */
#define NV_GRE_PROTOCOL      0x2F  /* Generic Routing Encapsulation (GRE) */
#define NV_ESP_PROTOCOL      0x32  /* IPsec ESP */
#define NV_AH_PROTOCOL       0x33  /* IPsec AH */
#define NV_ICMP6_PROTOCOL    0x3A  /* ICMPv6 */
#define NV_SCTP_PROTOCOL     0x84  /* SCTP */
#define NV_ROCE_PROTOCOL     0xFE  /* RoCE v1.5 */
```

- EtherType Values

```
#define NV_TYPE_VLAN_CTAG    0x8100
#define NV_TYPE_VLAN_STAG    0x88A8
#define NV_TYPE_IPV4         0x0800
#define NV_TYPE_IPV6         0x86DD
#define NV_TYPE_ARP          0x0806
#define NV_TYPE_CONTROL      0x0808
#define NV_TYPE_MPLS         0x8847
#define NV_TYPE_MPLS_MC      0x8848
#define NV_TYPE_PTP          0x88F7
#define NV_TYPE_FCOE         0x8906
#define NV_TYPE_ROCE         0x8915  /* v1 */
#define NV_TYPE_MAC          0x6558  /* Transparent Ethernet Bridge for Generic Routing Encapsulation */
```

- UDP Port Numbers

```
#define NV_ROCE_PORT         4791   /* v2 bth */
#define NV_VXLAN_PORT        4789
#define NV_VXLAN_GPE_PORT    4790
#define NV_GENEVE_PORT       6081
#define NV_MPLS_TUNNEL_PORT  6635
#define NV_IPSEC_NAT_PORT    4500
#define NV_PTP_EVENT_PORT    319
#define NV_PTP_GEN_PORT      320
#define NV_IPV4_OPTION_MRI   31
#define NV_GTP_U_PORT        2152
#define NV_PSP_PORT          1000
#define NV_GUE_PORT          666
```

- Tunnel encapsulation protocol numbers

```
/* VXLAN GPE protocols */
#define NV_VXLAN_GPE_IPV4    0x01
#define NV_VXLAN_GPE_IPV6    0x02
#define NV_VXLAN_GPE_MAC     0x03
#define NV_VXLAN_GPE_NSH     0x04   /* RFC8300 */

/* PSP protocols */
#define NV_PSP_IPV4          0x04
#define NV_PSP_IPV6          0x29
#define NV_PSP_TCP           0x06
#define NV_PSP_UDP           0x11
```

These values help interpret standard protocol fields within BlueField's packet processing pipeline.

8.11.7.1.5.4 5.4. Metadata

The standard metadata (**struct nv_standard_metadata_t**) for the DOCA Target Architecture are found in **doca_metadata.p4** . The members of the standard metadata struct are all read only, and are separated into 4 logical categories:

- Pipeline metadata, information set by hardware units before the programmable part of the BlueField pipeline
- Outer packet header metadata, information set by the hardware parser for the outer packet
- Inner packet header metadata, information set by the hardware parser for the inner packet (if present)
- Union packet fields (packet fields which share the same memory location)

8.11.7.1.5.5 5.5. User Metadata

The DPL programmer can create and use their own metadata struct in the DOCA Target Architecture. The type can be `bit<>` or a struct, limited in size by the number of internal registers on the device. For convenience, the Target Architecture defines `nv_empty_metadata_t` which the user optionally use in the main package declaration if no metadata is needed.

8.11.7.1.6 6. DOCA Packet Parser

In the DOCA Target Architecture, headers represent the various packet protocol formats that are recognized by the parser. Once these headers are identified, the fields within the packet become candidates for further processing. They can be matched against entries in a match-action table, where they may be altered based on user-defined actions. The DOCA Target Architecture comes equipped with a default hardware-integrated parser that can understand a range of protocols standardized by the Internet Engineering Task Force (IETF). To accommodate custom requirements, architecture allows for the integration of user-defined ("flex") headers. These flex headers are seamlessly incorporated into the pre-existing parser graph, extending its capabilities to recognize and process additional protocol formats as specified by the user. This flexibility enables developers to tailor the data plane processing to specific applications and protocols beyond the standard set, and future-proofs the developer's investment in the hardware.

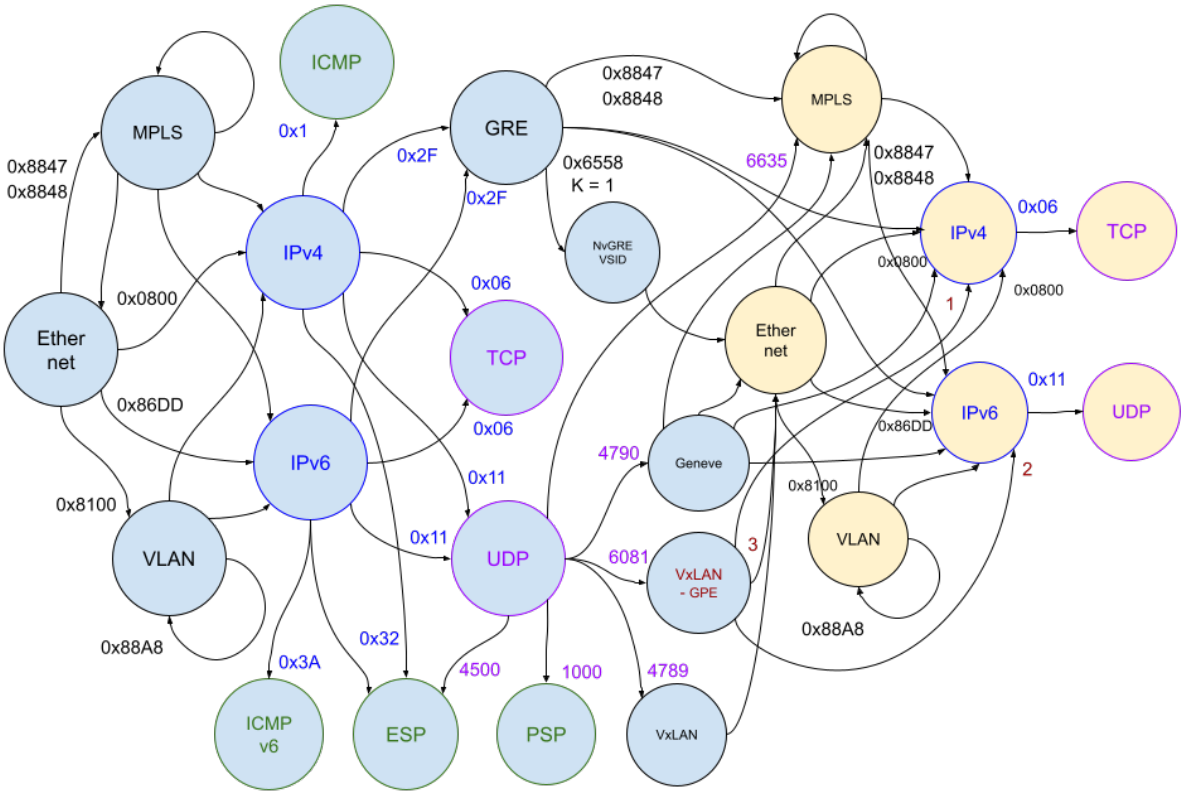
This section on the DPL Packet Parser focuses on BlueField-3 devices and higher. NVIDIA's BlueField architecture combines native and flexible parsing capabilities, allowing the user to enhance the hardware's packet parsing engine with custom protocol headers. This integration is seamless, utilizing built-in parsing for standardized headers.

8.11.7.1.6.1 6.1. Flex Parser Configuration

Flex nodes are user-configurable within the DOCA parser, covering elements including the next protocol field, fixed length headers, and calculations for variable header sizes. Arcs link flex nodes and feature properties such as the `is_tunnel` flag, transition value, source node, and destination node. Configurations that are not supported will trigger a compile-time error. Packet fields are extracted from the flex node to the hardware's sampler registers, facilitating the use of fields as match keys and enabling header modification actions.

8.11.7.1.6.2 6.2. Use Cases

Examples of custom flex headers include TCP options, SRv6, GTP-U, RTP, SCTP, GUE, eCPRI, GENEVE options, and proprietary tunnels. The hybrid architecture ensures smooth integration of user-defined protocols with the fixed parser graph. Below is an illustration of the default fixed parser for BlueField-3:



The DPL compiler enables the user to effectively utilize the P4 language to program packet parsers, taking full advantage of its native and flexible parsing capabilities to meet a variety of network processing needs. The remainder of this section describes the features of the BlueField-3 parser.

8.11.7.1.6.3 6.3. Native Parse Graph and Headers

BlueField's parser supports a hybrid architecture that incorporates both predefined (fixed) headers and standard transitions between headers as defined by the Internet Engineering Task Force (IETF). These transitions are determined based on the "next header type" field present within each protocol. For a comprehensive understanding of these transitions and the supported protocols, users are encouraged to refer to IANA Service Names and Port Numbers for a comprehensive list of protocol identifiers supported by the BlueField parser:

- Service Names and Port Numbers – IANA Service Names and Port Numbers
- IEEE 802 Numbers – IANA IEEE 802 Numbers
- Protocol Numbers – IANA Protocol Numbers

These resources provide detailed information on the various service names, port numbers, IEEE standards, and protocol numbers that are recognized and supported by BlueField's parser, facilitating a broad range of packet processing applications. The following table describes the native header and transition support in the DOCA Target Architecture:

Header Type	Next Protocol Field	Field Size (bits)	Fixed IETF Transitions	Flex Transition
MAC w VLAN	Ether Type	16	IPv4 (0x0800), IPv6 (0x86DD), MPLS UC (0x8847), MPLS MC (0x8848), VLAN (0x8100), SVLAN (0x88A8)	From, outer, inner
IPv4	Protocol	8	UDP (0x11), TCP (0x6), GRE(0x2F), ICMP(0x1), IPSec AH (0x33), IPSec ESP (0x32), IPv4 encap (0x4)	To, outer, inner
IPv6	Protocol	8	HOPOPT (0x0), UDP (0x11), TCP (0x6),GRE(0x2F), ICMP(0x3A), IPSec AH (0x33), IPSec ESP (0x32), IPv6-Route (0x2B), , IPv6 encap (0x29), IPv6-Frag(0x2C), IPv6-NoNxt (0x3B), IPv6-DestOpts (0x3c)	To, outer, inner
IP	Protocol	8	UDP, TCP, GRE, ICMP, IPSec AH, IPSec ESP	From, outer, inner (both IPv4 and IPv6)
UDP	Destination port	16	VXLAN (4789 + 3 additional custom ports) VXLAN-GPE (4790) GENEVE (6081) MPLS over UDP (6635) IPSEC ESP over UDP (4500) PSP(1000 + 2 additional custom ports)	To, from, outer, inner
TCP	Destination port	16	None	To, from, outer, inner
ICMP	None	N/A	None	Not supported - cannot transition to or from
GRE	Ether Type	16	IPv4 (0x0800), IPv6 (0x86DD) MPLS over GRE (0x8847, 0x8848)	To, from, outer
NVGRE (GRE)	Ether Type	16	Inner MAC (0x6558)	To, from, outer
NVGRE Options	Key present	1	Key present true	Cannot transition to or from
VXLAN	None	N/A	Inner MAC (fixed)	From, outer
VXLAN-GPE	Next Protocol	8	Reserved (0x00), IPv4 (0x01), IPv6 (0x02), Ethernet (0x03), NSH (0x04)	From, outer
GENEVE	Ether Type	16	Inner MAC (0x6558), IPv4 (0x0800), IPv6 (0x86DD), MPLS (0x8847, 0x8848)	From, outer
MPLS	Lookahead	4	IPv4, IPv6	To, from, outer
IPSEC ESP	Next Header	8	Not supported	From, outer
PSP	Next Header	8	Tunnel: IPv4 (4), IPv6 (41) Transport: TCP (6), UDP (17)	To, from, outer

Note that L3 control plane headers ARP (0x0806), MAC control (0x8808), LLDP (0x88CC), PTP (0x88F7) are recognized and steered to a special QP by hardware that are parsed but not matchable in the steering pipeline. Users should create a custom header if matching is required.

8.11.7.1.6.4 6.4. Using Default Native Parser

The DOCA Target Architecture defines a default native parser. The user need only include **doca_parser.p4** and reference **nv_fixed_parser** in the DPL package declaration to use the built-in hardware parser. The program should then use the native headers structure, **nv_headers_t**.

8.11.7.1.6.5 6.5. Reparsing Capability

A distinctive feature of DPL, setting it apart from other P4 hardware architectures, is its ability to perform reparsing as needed, even midway through the processing pipeline. This flexibility allows for dynamic adjustments to packet processing based on intermediate outcomes during the pipeline's execution. Unlike RMT-based P4 Target Architectures, there are no strict pipeline stages, as the pipeline runs to completion. This means:

- Tables may be applied multiple times, saving scale by not requiring the duplication of tables and their entries at different points in the pipeline.
- Packets do not need to be resubmitted to the pipeline after packet modification (e.g., for encapsulation or decapsulation actions). The push and pop header extern functions also control the reparsing behavior, along with a specific extern to force a reparse.
- Looping in the pipeline is permitted.

This capability greatly improves the user experience by allowing the DPL developer to focus on a logical view of the pipeline behavior, rather than dealing recirculation logic and saving state in metadata.

8.11.7.1.6.6 6.6. Defining a Custom Parser

The P4-16 language does not provide a method to extend a pre-existing, target architecture defined parser. Hence the DPL compiler maintains an internal representation of the default parser (based on a "read only" **doca_parser.p4** source file), and allows headers to be added or removed. In this case, header removal may be implicit, based on excluding a fixed header from the DOCA parser definition. There are 3 steps to defining a custom parser that will be used in conjunction with the native parser:

1. Define the custom header.
2. Add the header to a headers struct.
3. Define the state in a custom parser in the program.

6.6.1. Custom Headers and Headers Struct

A macro, **NV_FIXED_HEADERS**, is provided that references all the headers that the fixed parser can extract. This definition may be used directly as the body of type **headers_t**, or added to, for example:

```
#include <doca_parser.p4>
header custom_header_t {
    bit<16> x;
}
struct custom_headers_t
{
```

```
    NV_FIXED_HEADERS
    custom_header_t custom;
}
```

The DPL programmer is expected to use **NV_FIXED_HEADERS** as the basis of their **headers_t** definition. Failure to do so may result in unexpected behavior, for example:

- Headers/fields that are never extracted at runtime even if extracted by parser states
- Use of flex extractions for every single referenced header field
- Compilation errors

6.6.2. Custom Parser States

A macro, **NV_FIXED_PARSER**, is provided which describes the fixed parser. This definition may be used directly as the body of a parser, or extended by writing additional states, and linking them using the **@nv_transition_from** annotation:

```
parser my_parser(packet_in packet, out custom_headers_t headers)
{
    NV_FIXED_PARSER(packet, headers)

    @nv_transition_from("nv_parse_ethernet", 0x1234)
    state parse_custom {
        packet.extract(headers.custom);
        transition select(headers.custom.x) {
            1: nv_parse_ipv4;
            2: nv_parse_ipv6;
            default: accept;
        }
    }
}
```

The DPL programmer is expected to use **NV_FIXED_HEADERS** as the basis of their parser definition. The macro must take the same variable name use for packet_in and headers used in the parser control definition. Failure to do so may result in unexpected behavior, for example:

- Use of hardware flex parser resources for every parser state, even where hardware fixed parser resources could have been used.
- Definitions of states with names matching states in **NV_FIXED_PARSER** being ignored, and their fixed definition used instead.
- Compilation errors.

Finally, the custom parser should be instantiated in the main DPL package:

```
NvDocaPipeline(
    my_parser(),
    my_main_control()
) main;
```

8.11.7.1.6.7 6.7. Specifying Transitions to Flex Headers

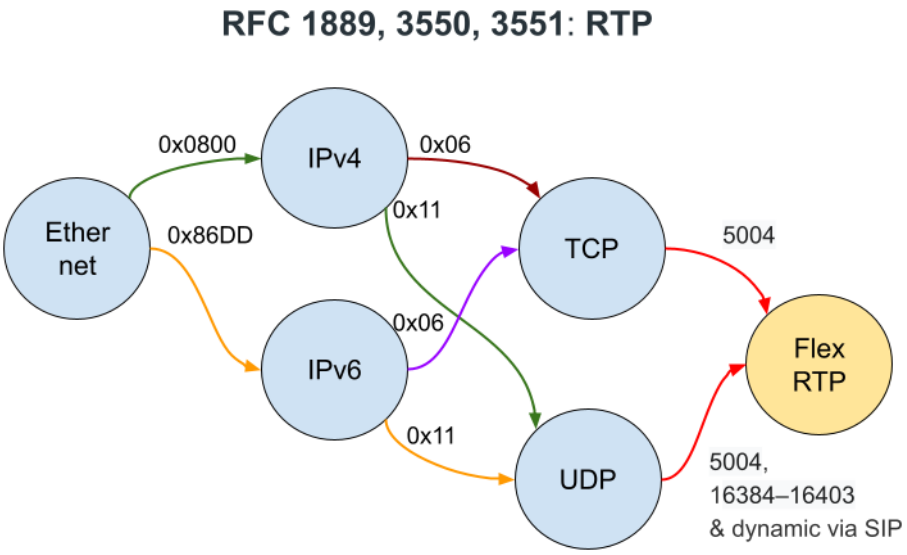
To extend the native parser with flex headers, the DOCA Target Architecture utilizes a combination of macros and DPL annotations to instruct the compiler how to stitch in the customized headers into an existing parse graph.

To create an unconditional transition from a source state, the parser state must be preceded by an **@nv_transition_from** annotation with 1 argument: the source state name. If the parser contains multiple state definitions with an **@nv_transition_from** annotation with the same source state, a single state will be selected (in an undefined manner) to be the target state of the transition. Put another way, each **@nv_transition_from** annotation from a given source state

annotation may be considered to override any earlier (in an undefined order) transition destination for that source state.

To create a conditional transition based on the source state's transition field, the parser state must be preceded by an `@nv_transition_from` annotation with 2 arguments: the source state name and transition value. Transitions of this type are prepended to the source state's list of transitions and thus are guaranteed to override any transitions defined by `NV_FIXED_PARSER` in an undefined order. Entries with identical transition values will override each other but may generate warnings due to unreachability.

The following diagram shows an example of extending the native parser to support the RTP protocol:



In the native parser, as an optimization, the user can specify a transition from both IPv4 and IPv6 headers to a flex header using a special 2 argument annotation of the format:

```
@nv_transition_from("nv_parse_ipv6, nv_parse_ipv4", 0x1234)
```

Where 0x1234 is an example of the custom IP protocol value for some user defined L4 protocol.

8.11.7.1.6.8 6.8. Specifying Transitions to Fixed Headers

In a limited number of cases, transitions from fixed headers can be configured. This uses a different annotation, `@nv_transition` , and is always used to annotate the top level parser object. For example, it is a common requirement to add nonstandard transitions from UDP to VxLAN:

```
@nv_transition("nv_parse_udp", "nv_parse_vxlan", 1234)
@nv_transition("nv_parse_udp", "nv_parse_vxlan", 1235)
@nv_transition("nv_parse_udp", "nv_parse_vxlan", 1236)
parser custom_parser(
    packet_in packet,
    out nv_headers_t headers
) {
    NV_FIXED_PARSER(packet, headers)
```

```
}
```

where the first parameter specifies the parser node to transition "from", and the second parameter specifies the parser node to transition "to". Note that the annotation **@nv_transition** can be used instead of **@nv_transition_from** for connecting headers.

8.11.7.1.6.9 6.9. Defining TLV Parsing for Headers with Optional Data

The DOCA Target Architecture allows the user to define protocol headers that have TLVs as sub headers. Note that a custom base "parent" header must be supplied in order to create custom TLVs. It is not supported to create custom TLVs for native headers. For example, the user can define a GENEVE header as follows:

```
header geneve_t {
    bit<2> ver;
    bit<6> opt_len;
    bit<1> o;
    bit<1> c;
    bit<6> reserved;
    bit<16> protocol_type;
    bit<24> vni;
    bit<8> reserved56;
};
```

And then define a struct for the base GENEVE option fields (TYPE and LENGTH) along with a customized VALUE data.

```
// Struct so it can be a field within other headers
struct geneve_option_t {
    bit<24> option_class;
    bit<8> option_type;
    bit<3> reserved24;
    bit<5> length;
};

header geneve_option_int_md_t {
    geneve_option_t base;
    bit<4> ver;
    bit<1> d;
    bit<1> e;
    bit<1> m;
    bit<12> reserved7;
    bit<5> hop_ml;
    bit<8> remaining_hop_count;
    // etc.
};
```

Next, the user must add the custom headers to the struct of headers and connect them in the parser. Both the base option struct and the newly defined headers must be present in the headers struct so that there is a reference to these types in the parser.

```
struct app_headers {
    NV_FIXED_HEADERS
    geneve_t custom_geneve;
    geneve_option_t geneve_opt;
    geneve_option_int_md_t geneve_opt_int_md;
};
```

The DOCA Target Architecture uses an **NvOptionsParser** extern object to define TLV options, since the P4 language itself does not provide a native way to define TLV parsing. An instance of the **NvOptionsParser** must be created in the parser, at the outermost scope, with the following parameters:

- **options_length_field** - the name of the field in the parent header (as a string) that holds the total length of the options. If the total options length is fixed (i.e., not specified in the header's data), then this field should be omitted.

- **options_length_shift** - the value by which the parser should apply a shift (i.e. multiply by power of 2) to the value of the option length field. If the length value is fixed or does not need to be shifted, then this field should be omitted.
- **options_length_add** - the constant to which the parser should add to the value option length field after shifting. If the length value does not need an addend, then this field should be omitted. If the total options length is fixed, then this field must specify the fixed length of the TLV options.
- **option_layout header_type** - the user defined P4 struct that holds the TLV field layout of the option header
- **option_length_field** - the name of the field in the base option struct (as a string) that holds the length of the options. If the options length is fixed (i.e., not specified in the base options struct), then this field should be omitted.
- **option_length_shift** - the value by which the parser should apply a shift (i.e. multiply by power of 2) to the option length field. If the length value is fixed or does not need to be shifted, then this field should be omitted.
- **option_length_add** - the constant to which the parser should add to the option length field. If the length value does not need an addend, then this field should be omitted. If the options length is fixed, then this field must specify the fixed length of the TLV options.
- **options** - a list of tuples, where the first value is the TYPE, and the second value is the name of the child header defined by the user

```
parser geneve_parser(
    packet_in packet,
    out app_headers headers
) {
    NvOptionParser<bit<24>, _>(
        "opt_len",           // options_length_field
        2,                   // options_length_shift
        0,                   // options_length_add
        "geneve_option_t",   // option_layout_header_type
        "length",            // option_length_field
        0,                   // option_length_shift
        4,                   // option_length_add
        "option_class_type", // option_type_field
        // options data
        (list<tuple<bit<24>, _>>){
            {24w0x010301, "headers.geneve_opt_int_md"}
            // list of additional options ...
        }
    ) geneveOptions;

    NV_FIXED_PARSER(packet, headers)

    @nv_transition_from("nv_parse_udp", 6082)
    state parse_custom_geneve {
        packet.extract(headers.custom_geneve);
        geneveOptions.parseOptions(packet, headers);
        transition accept;
    }
}
```

8.11.7.1.6.10 6.10. Restrictions and Unsupported Parser Features

DOCA parser features that either differ from the $P4_{16}$ specification, are restricted in the DOCA Target Architecture or are unsupported are listed below.

6.10.1. Unsupported Features

These features are not supported by the DOCA Target Architecture in comparison to the $P4_{16}$ specification. Utilizing these features will result in compilation errors.

- Variable Declarations and Extern Instantiations: Parsers and parser states cannot contain any variable declarations or extern instantiations other than **NvOptionsParser**.

- Lookahead and Advance: Programmable lookahead and advance functionalities are not supported.
- Parser Value Sets: Parser value sets for setting transition values from runtime data are not supported.
- State Declarations: States may not contain any declaration, or any statement besides extract, transition statements and calls to `NvOptionsParser` methods.
- Expressions: Conditionals and assignments are not supported.

6.10.2. Restricted Behavior

The following points highlight how the DOCA Target Architecture implementation of DPL parser features differs from the standard $P4_{16}$ specification.

- Reserved State Names: Parser state names beginning with `nv_*` are reserved by the DOCA Target Architecture. Custom states using this reserved prefix will be ignored.
- State and Header Coupling: Each state must extract exactly one header, and they are considered to be coupled in a 1:1 fashion.
- Fixed-Size Header Extraction: The extracted header must be fixed-size, known at compile-time, and defined in the P4 header type definition. I.e., `p.extract(headers.xxx)` is allowed, but `p.extract(headers.xxx, someLength)` is not.
- Transition Statements: Besides terminal pre-defined states (`accept` , `reject`), states must transition to another state using a transition statement, which may be unconditional or conditional.
- Transition Select: Transition select statements are limited to using a single field from the header the state extracts or a constant. Lists of expressions, constants, operators, references to declarations other than a field in the header the state extracts, and any other expression not explicitly allowed are disallowed.
- Default Case in Transition Select: Every transition select statement must have a default case, transitioning to the accept state.
- Extract Before Transition: A state's `extract` statement must precede its transition statement.
- Empty Accept and Reject States: The `accept` and `reject` states must be completely empty.
- Loops in Parser Graph: Loops are not allowed in the DPL program's parser graph. Each path through the parser must be acyclic.

6.10.3. Hybrid Parser Behavior

These restrictions and behaviors apply to the unique hybrid fixed/flex parser model supported by the DOCA Target Architecture.

- Start state of `nv_parse_ethernet` – The start state cannot be changed or overwritten.
- Redefining states – States may not be redefined. This includes no possibility of applying `@nv_transition_from` to the body of any fixed state.
- `NV_FIXED_PARSER` and `NV_FIXED_HEADERS` – States part of `NV_FIXED_PARSER` extract the pre-defined fixed header, irrespective of the state body. States not part of `NV_FIXED_PARSER` must extract headers not part of `NV_FIXED_HEADERS` .

- Fixed-to-fixed transitions – It is not possible to disable any fixed->fixed transition. These may be overridden to transition to a custom state by applying an `@nv_transition_from` annotation to the destination state.
- MPLSoUDP stops parsing at the inner MAC header if the pseudo-wire Control Word is set to 0. Additional inner packet fields will not be parsed.

8.11.7.1.7 7. DOCA Extern Functions

Extern functions serve as a mechanism for exposing DPL target specific functionality that may be beyond a standard P4 model.

Note that many of these externs mutate the packet, for which the HEADERS are a part of the extern signature.

8.11.7.1.7.1 7.1. nv_drop

Terminal extern function that stops packet processing and drops the packet.

Signature:

```
extern void nv_drop();
```

8.11.7.1.7.2 7.2. nv_send_to_port

Terminal extern function that stops packet processing and sends the packet to the specified port.

Signature:

```
extern void nv_send_to_port(in nv_logical_port_t port);
```

Parameters:

- **port[in]** : This parameter specifies the logical port to which the packet will be sent.

8.11.7.1.7.3 7.3. nv_send_to_controller

Terminal extern function that forwards packet metadata to a P4Runtime controller

Signature:

```
extern void nv_send_to_controller<PACKET_META>(in PACKET_META pkt_in_meta);
```

Parameters:

- **pkt_in_meta[in]** : The metadata accompanying the packet being sent to the controller for processing. Maximum of 32 bits of metadata.

8.11.7.1.7.4 7.4. nv_set_l2_ethertype

Extern function that sets the last extracted etherType field. The hardware only maintains the last etherType when parsing an L2 header with multiple VLAN tags.

Signature:

```
extern void nv_set_l2_ethertype<HEADERS>(inout HEADERS headers, in bit<16> last_ether_type);
```

Parameters:

- **headers[in/out]** : packet headers struct
- **last_ether_type[in]** : ethertype value to set

8.11.7.1.7.5 7.5. nv_set_ip_dscp

Extern function that sets the DSCP value in the IP header. Applies to both IPv4/IPv6 portion of the traffic class field.

Signature:

```
extern void nv_set_ip_dscp<HEADERS>(inout HEADERS headers, in bit<6> dscp_value);
```

Parameters:

- **headers[in/out]** : packet headers struct
- **dscp_value[in]** : DSCP value to set

8.11.7.1.7.6 7.6. nv_set_ip_ecn

Extern function that sets the ECN value in the IP header. Applies to both IPv4/IPv6 portion of the traffic class field.

Signature:

```
extern void nv_set_ip_ecn<HEADERS>(inout HEADERS headers, in bit<2> ecn_value);
```

Parameters:

- **headers** [in/out]: packet headers struct
- **ecn_value** [in]: ECN value to set

8.11.7.1.7.7 7.7. nv_set_ip_protocol

Extern function that sets the IP protocol value in an IP header. Applies to both IPv4 protocol field and IPv6 next header field.

Signature:

```
extern void nv_set_ip_protocol<HEADERS>(inout HEADERS headers, in bit<8> protocol);
```

Parameters:

- **headers** [in/out]: packet headers struct
- **protocol** [in]: IP protocol value to set

8.11.7.1.7.8 7.8. nv_set_l4_src_port

Extern function that sets the L4 source port value in the TCP/UDP header.

Signature:

```
extern void nv_set_l4_src_port<HEADERS>(inout HEADERS headers, in bit<16> src_port);
```

Parameters:

- **headers** [in/out]: packet headers struct
- **src_port** [in]: L4 port value to set

8.11.7.1.7.9 7.9. nv_set_l4_dst_port

Extern function that sets the L4 destination port value the TCP/UDP header.

Signature:

```
extern void nv_set_l4_dst_port<HEADERS>(inout HEADERS headers, in bit<16> dst_port);
```

Parameters:

- **headers** [in/out]: packet headers struct
- **dst_port** [in]: L4 port value to set

8.11.7.1.7.10 7.10. nv_l2_decap

Extern function that performs L2 decapsulation on the packet headers (i.e., removing Ethernet/VLAN/tunnel headers). The encapsulated inner packet must begin with a valid L2 Ethernet header. The packet will be reparsed at the end of this action.

Signature:

```
void nv_l2_decap<HEADERS>(inout HEADERS headers)
```

Parameters:

- **headers** [in/out]: packet headers struct that will be decapsulated.

8.11.7.1.7.11 7.11. nv_l3_decap

Extern function that performs Layer 3 decapsulation on the packet headers; i.e. removing ethernet/VLAN/tunnel headers. A new L2 header will be prepended with specified data. The encapsulated inner packet must begin with a valid L3 header. The packet will be reparsed at the end of this action.

```
void nv_l3_decap<HEADERS>(inout HEADERS headers, nv_mac_addr_t dst_mac, nv_mac_addr_t src_mac, bit<16>
l3_ether_type, @optional nv_vlan_id_t vid)
```

Parameters:

- **headers** [inout]: The packet headers that will be decapsulated from.

- **dst_mac** [in]: The destination MAC address in the Ethernet frame.
- **src_mac** [in]: The source MAC address in the Ethernet frame.
- **l3_ether_type** [in]: The EtherType value in the Ethernet frame
- **vid** [in]: Optional VLAN ID.

8.11.7.1.7.12 7.12. nv_push_vlan

Extern function that inserts a VLAN header immediately after the L2 header.

```
void nv_push_vlan<HEADERS>(inout HEADERS headers, in NvVlanTagId tpid, in bit<3> pcp, in bit dei, in nv_vlan_id_t
vid);
```

Parameters:

- **headers** [inout]: The packet headers that the VLAN will be pushed to.
- **tpid** [in]: Tag Protocol Identifier. SVLAN = 0x88a8, CVLAN = 0x8100.
- **pcp** [in]: Priority Code Point.
- **dei** [in]: Drop Eligible Indicator (formerly CFI).
- **vid** [in]: VLAN Identifier.

8.11.7.1.7.13 7.13. nv_pop_vlan

Extern function that removes the outermost VLAN header, immediately after the L2 header.

```
void nv_pop_vlan<HEADERS>(inout HEADERS headers);
```

Parameters:

- **headers** [inout]: The packet headers that the VLAN will be popped from.

8.11.7.1.7.14 7.14. nv_push_headers

Extern function that inserts one or more headers (native or custom) to the packet, starting at the specified anchor point.

```
void nv_push_headers<HEADERS, NV_HEADER_DATA_TEMPLATE, VALUES, NV_ANCHOR>(
    inout HEADERS headers,
    NV_HEADER_DATA_TEMPLATE data,
    in VALUES values,
    NV_ANCHOR anchor,
    bit<7> word_offset = 7w0,
    bool do_updates = true,
    bool do_reparse = true
);
```

The following anchor points are allowed:

```
enum NvAnchor {
    PACKET_START,
    L2_START_OF_HEADER,
    VLAN_1ST_START_OF_HEADER,
    VLAN_2ND_START_OF_HEADER,
    MPLS_START_OF_HEADER,
    IP_START_OF_HEADER,
    L4_START_OF_HEADER,
    ESP_START_OF_HEADER,
    TUNNEL_START_OF_HEADER,
    INNER_L2_START_OF_HEADER,
    INNER_VLAN_1ST_START_OF_HEADER,
    INNER_VLAN_2ND_START_OF_HEADER,
    INNER_MPLS_START_OF_HEADER,
```

```
    INNER_IP_START_OF_HEADER,
    INNER_L4_START_OF_HEADER,
    PSP_START_OF_HEADER,
    PSP_PAYLOAD_START,
    BTH_HEADER_START_OF_HEADER,
};
```

Parameters:

- **headers** [in/out]: The packet headers to which the additional headers are to be pushed.
- **data** [in]: The NvHeaderDataTemplate object that specifies the list of headers and the header values. See also **NvHeaderDataTemplate** annotation.
- **values** [in]: A list expression whose length matches the number of header fields marked "variable" in the **NvHeaderDataTemplate** object's annotation. The order of the header fields determines which value a field corresponds to (e.g., 3rd header field marked "variable" will be set to the 3rd value in the list). Each value is required to be runtime constant (i.e., it may either be an integer value or an action parameter value). Named parameters are not supported in the list expression.
- **anchor** [in]: An enum specifying the location in the packet where the header insertion should start. The user must separately verify the existence of the header (using isValid), otherwise the behavior is undefined.
- **word_offset** [in]: The offset from the anchor point, in words, where the header insertion should start.
- **do_updates** [in]: If true, then perform IP header field updates (e.g., checksum, length) once the push is completed (e.g., for encapsulations).
- **do_reparse** [in]: If true, then perform a packet reparse once the push is completed.

Limitations:

Only the following combinations of anchors, offsets, and packet updates are validated:

Start Anchor	Offset (bytes)	Do Update
PACKET_START	0	true
PACKET_START	0	false
PACKET_START	12	false
L4_START_OF_HEADER	0	true
IP_START_OF_HEADER	0	false
IP_START_OF_HEADER	20	false
IP_START_OF_HEADER	40	false
ESP_START_OF_HEADER	0	false

Use of parameters beyond the limitations specified above is considered experimental and is provided *as-is* for testing purposes only. This usage is known to contain defects and may result in unpredictable behavior, including—but not limited to—packet corruption, data loss, or network instability. Deploying such features in production environments is

strictly discouraged. These capabilities should be used only in isolated, non-critical test environments with appropriate safeguards in place to prevent potential disruptions to network operations.

8.11.7.1.7.15 7.15. nv_pop_header_words

Extern function that removes data from a packet, starting at the specific anchor point.

```
void nv_pop_header_words<HEADERS, NV_ANCHOR>(
    inout HEADERS headers,
    NV_ANCHOR anchor,
    bit<6> num_words,
    bit<7> word_offset = 7w0,
    bool do_reparse = true
);
```

Parameters:

- **headers** [in/out]: The packet headers to which the header data is to be removed.
- **anchor** [in]: An enum specifying the location in the packet where the header data removal should start. The user must separately verify the existence of the header (using isValid), otherwise the behavior is undefined.
- **num_words** [in]: The amount of packet data to remove, in words.
- **word_offset** [in]: The offset from the anchor point, in words, where the header removal should start.
- **do_reparse** [in]: If true, then perform a packet reparse once the removal is completed.

Limitations:

Only the following anchors listed below are validated.

Start Anchor
PACKET_START
VLAN_1ST_START_OF_HEADER
IP_START_OF_HEADER
L4_START_OF_HEADER
ESP_START_OF_HEADER

Use of parameters beyond the limitations specified above is considered experimental and is provided *as-is* for testing purposes only. This usage is known to contain defects and may result in unpredictable behavior, including—but not limited to—packet corruption, data loss, or network instability. Deploying such features in production environments is strictly discouraged. These capabilities should be used only in isolated, non-critical test environments with appropriate safeguards in place to prevent potential disruptions to network operations.

8.11.7.1.7.16 7.16. nv_pop_headers

Extern function that removes data from a packet, beginning at the specified start anchor point and stopping at the specified end anchor point.

```
void nv_pop_headers<HEADERS, NV_ANCHOR_START, NV_ANCHOR_END>(
    inout HEADERS headers,
    NV_ANCHOR_START start_anchor,
    NV_ANCHOR_END end_anchor,
    bool do_updates = true,
    bool do_reparse = true
);
```

Parameters:

- **headers** [in/out]: The packet headers to which the header data is to be removed.
- **start_anchor** [in]: An enum specifying the location in the packet where the header data removal should start. The user must separately verify the existence of the header (using isValid), otherwise the behavior is undefined.
- **end_anchor** [in]: An enum specifying the location in the packet where the header data removal should stop. The user must separately verify the existence of the header (using isValid), otherwise the behavior is undefined.
- **do_updates** [in]: If true, then perform IP header field updates (e.g., checksum, length) once the pop is completed (e.g., for decapsulations).
- **do_reparse** [in]: If true, then perform a packet reparse once the pop is completed.

Limitations:

Only the exact combinations of start anchor, end anchor and packet updates (e.g., due to decapsulation) are validated.

Start Anchor	End Anchor	Do Update
PACKET_START	INNER_L2_START_OF_HEADER	Yes
PACKET_START	INNER_IP_START_OF_HEADER	Yes
PACKET_START	PACKET_START	Yes
PACKET_START	IP_START_OF_HEADER	No
IP_START_OF_HEADER	L4_START_OF_HEADER	No
IP_START_OF_HEADER	IP_START_OF_HEADER	Yes
L4_START_OF_HEADER	PSP_PAYLOAD_START	Yes
PSP_START_OF_HEADER	PSP_PAYLOAD_START	No
PACKET_START	PSP_PAYLOAD_START	Yes

Use of parameters beyond the limitations specified above is considered experimental and is provided *as-is* for testing purposes only. This usage is known to contain defects and may result in unpredictable behavior, including—but not limited to—packet corruption, data loss, or network instability. Deploying such features in production environments is

strictly discouraged. These capabilities should be used only in isolated, non-critical test environments with appropriate safeguards in place to prevent potential disruptions to network operations.

8.11.7.1.7.17 7.17. nv_force_reparse

This extern is used in conjunction with **nv_push_headers**, **nv_pop_header_words**, and **nv_pop_headers**. Its purpose is to trigger a packet reparse after header modifications. When performing a series of header modifications with the **do_parse** flag **false**, it may be necessary to commit the final changes and have the packet reparsed. Certain fields such as standard checksums and length fields may be recalculated. However, triggering a reparse mid-modification may leave the packet in an indeterminate state.

Signature:

```
extern void nv_force_reparse<HEADERS>(inout HEADERS headers);
```

Parameters:

- **headers** [in/out]: The packet headers that will be reparsed

8.11.7.1.7.18 7.18. nv_set_vxlan_v4_underlay

Description:

Extern function that encapsulates the packet with an ethernet frame (optionally VLAN tagged), an ipv4 header and a VXLAN header.

Signature:

```
void nv_set_vxlan_v4_underlay<HEADERS>(inout HEADERS headers, in nv_mac_addr_t dst_mac, in nv_mac_addr_t src_mac,
in nv_ipv4_addr_t sip, in nv_ipv4_addr_t dip, in nv_vxlan_id_t vni, @optional in nv_vlan_id_t vid)
```

Parameters:

- **headers** [in/out]: The packet headers that are to be encapsulated with a VXLAN underlay.
- **dst_mac** [in]: Destination MAC address.
- **src_mac** [in]: Source MAC address.
- **sip** [in]: Source IPv4 address.
- **dip** [in]: Destination IPv4 address.
- **vni** [in]: VXLAN Network Identifier.
- **vid** [in]: Optional VLAN ID

8.11.7.1.7.19 7.19. nv_set_vxlan_v6_underlay

Description:
Extern function that encapsulates the packet with an ethernet frame (optionally VLAN tagged), an IPv6 header, and a VXLAN header.

Signature:

```
void nv_set_vxlan_v6_underlay<HEADERS>(inout HEADERS headers, in nv_mac_addr_t dst_mac, in nv_mac_addr_t src_mac,
in nv_ipv6_addr_t sip, in nv_ipv6_addr_t dip, in nv_vxlan_id_t vni, @optional in nv_vlan_id_t vid)
```

Parameters:

- **headers** [in/out]: The packet headers that are to be encapsulated with a VXLAN underlay.
- **dst_mac** [in]: Destination MAC address.
- **src_mac** [in]: Source MAC address.
- **sip** [in]: Source IPv6 address.
- **dip** [in]: Destination IPv6 address.
- **vni** [in]: VXLAN Network Identifier.
- **vid** [in]: Optional VLAN ID.

8.11.7.1.7.20 7.20. nv_set_gre_v4_underlay

Description:
Extern function that configures GRE IPV4 underlay encapsulation, with an optional VLAN tag and optional GRE key.

Signature:

```
void nv_set_gre_v4_underlay<HEADERS>(inout HEADERS headers, in nv_mac_addr_t dst_mac, in nv_mac_addr_t src_mac, in
nv_ipv4_addr_t sip, in nv_ipv4_addr_t dip, in NvInnerProtocolType proto_type, @optional in nv_vlan_id_t vid,
@optional in bit<32> key)
```

Parameters:

- **headers** [in/out]: The packet headers that are to be encapsulated with a GRE underlay.
- **dst_mac** [in]: Destination MAC address.
- **src_mac** [in]: Source MAC address.
- **sip** [in]: Source IPv4 address.
- **dip** [in]: Destination IPv4 address.
- **proto_type** [in]: Protocol type for the inner payload.
- **vid** [in]: Optional VLAN ID.
- **key** [in]: Optional GRE key.

8.11.7.1.7.21 7.21. nv_set_gre_v6_underlay

Description:
Extern function that configures GRE IPV6 underlay encapsulation, with an optional VLAN tag and optional GRE key.

Signature:

```
void nv_set_gre_v6_underlay<HEADERS>(inout HEADERS headers, in nv_mac_addr_t dst_mac, in nv_mac_addr_t src_mac, in
nv_ipv6_addr_t sip, in nv_ipv6_addr_t dip, in NvInnerProtocolType proto_type, @optional in nv_vlan_id_t vid,
@optional in bit<32> key)
```

Parameters:

- **headers** [in/out]: The packet headers that are to be encapsulated with a GRE underlay.
- **dst_mac** [in]: Destination MAC address.

- **src_mac** [in]: Source MAC address.
- **sip** [in]: Source IPv6 address.
- **dip** [in]: Destination IPv6 address.
- **proto_type** [in]: Protocol type for the inner payload.
- **vid** [in]: Optional VLAN ID.
- **key** [in]: Optional GRE key.

8.11.7.1.7.22 7.22. nv_set_geneve_v4_underlay

Description:
Extern function that configures GENEVE IPV4 underlay encapsulation with optional VLAN tagging and an optional GENEVE option with 32 bits of data.

Signature:

```
void nv_set_geneve_v4_underlay<HEADERS>(inout HEADERS headers, in nv_mac_addr_t dst_mac, in nv_mac_addr_t src_mac,
in nv_ipv4_addr_t src_ip, in nv_ipv4_addr_t dst_ip, in bit<6> opt_fields_len, in bit<1> oam, in bit<1> critical, in
NvInnerProtocolType proto_type, in bit<24> vni, @optional in nv_vlan_id_t vid, @optional in bit<16> opt_class,
@optional in bit<8> opt_type, @optional in bit<5> opt_len, @optional in bit<32> opt_data)
```

Parameters:

- **headers** [in/out]: The packet headers that are to be encapsulated with a GENEVE underlay.
- **dst_mac** [in]: Destination MAC address.
- **src_mac** [in]: Source MAC address.
- **src_ip** [in]: Source IPv4 address.
- **dst_ip** [in]: Destination IPv4 address.
- **opt_fields_len** [in]: Length of the options field in 4-byte units (must be set to 0 if no options are set)
- **oam** [in]: OAM flag.
- **critical** [in]: Critical options flag.
- **proto_type** [in]: Protocol type for the inner payload.
- **vni** [in]: Virtual Network Identifier.
- **vid** [in]: Optional VLAN ID.
- **opt_class** [in]: Optional GENEVE option class.
- **opt_type** [in]: Optional GENEVE option type.
- **opt_len** [in]: Optional GENEVE option length in 4-byte units. (must be 1 if set)
- **opt_data** [in]: Optional GENEVE option data.

8.11.7.1.7.23 7.23. nv_set_geneve_v6_underlay

Description:
Extern function that configures GENEVE IPV6 underlay encapsulation with optional VLAN tagging and an optional GENEVE option with 32 bits of data.

Signature:

```
void nv_set_geneve_v6_underlay<HEADERS>(inout HEADERS headers, in nv_mac_addr_t dst_mac, in nv_mac_addr_t src_mac,
in nv_ipv6_addr_t src_ip, in nv_ipv6_addr_t dst_ip, in bit<6> opt_fields_len, in bit<1> oam, in bit<1> critical, in
```

```
NvInnerProtocolType proto_type, in bit<24> vni, @optional in nv_vlan_id_t vid, @optional in bit<16> opt_class,
@optional in bit<8> opt_type, @optional in bit<5> opt_len, @optional in bit<32> opt_data)
```

Parameters:

- **headers** [in/out]: The packet headers that are to be encapsulated with a GENEVE underlay.
- **dst_mac** [in]: Destination MAC address.
- **src_mac** [in]: Source MAC address.
- **src_ip** [in]: Source IPv6 address.
- **dst_ip** [in]: Destination IPv6 address.
- **opt_fields_len** [in]: Length of the options field in 4-byte units (must be set to 0 if no options are set, otherwise must be set to 2)
- **oam** [in]: OAM flag.
- **critical** [in]: Critical options flag.
- **proto_type** [in]: Protocol type for the inner payload.
- **vni** [in]: Virtual Network Identifier.
- **vid** [in]: Optional VLAN ID.
- **opt_class** [in]: Optional GENEVE option class.
- **opt_type** [in]: Optional GENEVE option type.
- **opt_len** [in]: Optional GENEVE option length in 4-byte units. (must be 1 if set)
- **opt_data** [in]: Optional GENEVE option data.

8.11.7.1.7.24 7.24. nv_set_l2tunnel_underlay

Description:
Extern function that configures custom underlay followed by an L2 encapsulation of the original packet. It is optional, but recommended, that the new tunnel encapsulation be parsable by the DPL parser. For more details see the section on NvHeaderDataTemplate.

Signature:

```
void nv_set_l2tunnel_underlay<HEADERS, NV_HEADER_DATA_TEMPLATE, VALUES>(
    inout HEADERS headers,
    NV_HEADER_DATA_TEMPLATE tunnel,
    in VALUES values)
```

Parameters:

- **headers** [in/out]: The packet headers that are to be encapsulated.
- **tunnel** [in]: The **NvHeaderDataTemplate** object that specifies the underlay headers and the header values. See also NvHeaderDataTemplate annotation.
- **values** [in]: A list expression whose length matches the number of header fields marked "variable" in the **NvTunnelTemplate** object's annotation. The order of the header fields determines which value a field corresponds to (e.g., 3rd header field marked "variable" will be set to the 3rd value in the list). Each value is required to be runtime constant, meaning it may either be an integer value or an action parameter value. Named parameters are not supported in the list expression.

8.11.7.1.7.25 7.25. nv_set_l3tunnel_underlay

Description:
Extern function that configures custom underlay followed by an L3 encapsulation of the original packet. The L3 layer must be IP. It is optional, but recommended, that the new tunnel encapsulation be parsable by the DPL parser. For more details see the section on NvHeaderDataTemplate.

Signature:

```
void nv_set_l3tunnel_underlay<HEADERS, NV_HEADER_DATA_TEMPLATE, VALUES>(
    inout HEADERS headers,
    NV_HEADER_DATA_TEMPLATE tunnel,
    in VALUES values)
```

Parameters:

- **headers** [in/out]: The packet headers that are to be encapsulated
- **tunnel** [in]: The **NvHeaderDataTemplate** object that specifies the underlay headers and the header values. See also NvHeaderDataTemplate annotation.
- **values** [in]: A list expression whose length matches the number of header fields marked "variable" in the **NvTunnelTemplate** object's annotation. The order of the header fields determines which value a field corresponds to (e.g., 3rd header field marked "variable" will be set to the 3rd value in the list). Each value is required to be runtime constant, meaning it may either be an integer value or an action parameter value. Named parameters are not supported in the list expression.

8.11.7.1.7.26 7.26. nv_encap_esp_v4_transport

Description:
Extern function that encapsulates a clear text IPv4 packet for IPsec transport mode with an ESP header. The resulting packet is ready for IPsec encryption.

Signature:

```
extern void nv_encap_esp_v4_transport<HEADERS>(
    inout HEADERS headers,
    NvIPsecIcv icv_length, // Must match the instance's icv_length
    in bit<32> spi
);
```

Parameters:

- **headers** [in/out]: The packet headers that are to be encapsulated with a ESP headers.
- **icv_length** : The **NvIPsecIcv** enumeration value that specifies length of the ICV

```
typedef bit<8> nv_ipsec_icv_length_t;
enum nv_ipsec_icv_length_t NvIPsecIcv {
    LENGTH_64_BITS = 8w8,
    LENGTH_96_BITS = 8w12,
    LENGTH_128_BITS = 8w16,
}
```

- **spi** [in]: A 32-bit value that is used by a receiver to identify the Security Association (SA) to which an incoming packet is bound.

8.11.7.1.7.27 7.27. nv_encap_esp_v6_transport

Description:
Extern function that encapsulates a clear text IPv6 packet in IPsec transport mode with an ESP header. The resulting packet is ready for IPsec encryption.

Signature:

```
extern void nv_encap_esp_v6_transport<HEADERS>(
    inout HEADERS headers,
    NvIPsecIcv icv_length, // Must match the instance's icv_length
    in bit<32> spi
);
```

Parameters:

- **headers** [in/out]: The packet headers that are to be encapsulated with a ESP headers.
- **icv_length** : The **NvIPsecIcv** enumeration value that specifies length of the ICV.
- **spi** [in]: A 32-bit value that is used by a receiver to identify the Security Association (SA) to which an incoming packet is bound.

8.11.7.1.7.28 7.28. nv_decap_esp_transport

Description:
Extern function that decapsulates a clear text (decrypted) IP packet by removing the IPsec transport mode ESP header and trailer.

Signature:

```
extern void nv_decap_esp_transport<HEADERS>(
    inout HEADERS headers,
    NvIPsecIcv icv_length // Must match the instance's icv_length
);
```

Parameters:

- **headers** [in/out]: The packet headers that are to be decapsulated by removing the ESP headers.
- **icv_length** : The **NvIPsecIcv** enumeration value that specifies length of the ICV.

8.11.7.1.7.29 7.29. nv_encap_esp_v4_tunnel

Description:
Extern function that encapsulates a clear text IPv4 packet in IPsec tunnel mode with an ESP header. The resulting packet is ready for IPsec encryption.

Signature:

```
extern void nv_encap_esp_v4_tunnel<HEADERS>(
    inout HEADERS headers,
    NvIPsecIcv icv_length, // Must match the instance's icv_length
    in nv_mac_addr_t dst_mac,
    in nv_mac_addr_t src_mac,
    in nv_ipv4_addr_t sip,
    in nv_ipv4_addr_t dip,
    in bit<32> spi,
    in NvIPEncapProtocol next_header,
    @optional in nv_vlan_id_t vid
);
```

Parameters:

- **headers** [in/out]: The packet headers that are to be encapsulated with the ESP headers.
- **dst_mac** [in]: Destination MAC address.
- **src_mac** [in]: Source MAC address.
- **sip** [in]: Source IPv4 address.
- **dip** [in]: Destination IPv4 address.
- **spi** [in]: A 32-bit value that is used by a receiver to identify the Security Association (SA) to which an incoming packet is bound
- **next_header** [in]: IP protocol number, identifying the ESP next header. NvIPEncapProtocolType.IPV4 or NvIPEncapProtocolType.IPV6

```
enum nv_inner_protocol_type_t NvIPEncapProtocolType {
  IPV4 = NV_TYPE_IPV4,
  IPV6 = NV_TYPE_IPV6
}
```

- **vid** [in]: An optional VLAN ID to be included in the underlay headers.

8.11.7.1.7.30 7.30. nv_encap_esp_v6_tunnel

Description:
Extern function that encapsulates a clear text IPv6 packet in IPsec tunnel mode with an ESP header. The resulting packet is ready for IPsec encryption.

Signature:

```
extern void nv_encap_esp_v6_tunnel<HEADERS>(
    inout HEADERS headers,
    NvIPsecIcv icv_length, // Must match the instance's icv_length
    in nv_mac_addr_t dst_mac,
    in nv_mac_addr_t src_mac,
    in nv_ipv6_addr_t sip,
    in nv_ipv6_addr_t dip,
    in bit<32> spi,
    in NvIPEncapProtocol next_header,
    @optional in nv_vlan_id_t vid
);
```

Parameters:

- **headers** [in/out]: The packet headers that are to be encapsulated with a ESP headers.
- **dst_mac** [in]: Destination MAC address.
- **src_mac** [in]: Source MAC address.
- **sip** [in]: Source IPv6 address.
- **dip** [in]: Destination IPv6 address.
- **spi** [in]: A 32-bit value that is used by a receiver to identify the Security Association (SA) to which an incoming packet is bound.
- **next_header** [in]: IP protocol number, identifying the ESP next header. NvIPEncapProtocol.IPV4 or NvIPEncapProtocol.IPv6.
- **vid** [in]: An optional VLAN ID to be included in the underlay headers.

8.11.7.1.7.31 7.31. nv_decap_esp_tunnel

Description:
Extern function that decapsulates a clear text (decrypted) IP packet by removing the IPsec tunnel mode ESP header and trailer.

Signature:

```
enum nv_inner_protocol_type_t NvEspInnerProtocolType {
  IPV4 = NV_TYPE_IPV4,
  IPV6 = NV_TYPE_IPV6
}

extern void nv_decap_esp_tunnel<HEADERS>(
    inout HEADERS headers,
    NvIPsecIcv icv_length, // Must match the instance's icv_length
    in nv_mac_addr_t dst_mac,
    in nv_mac_addr_t src_mac,
    in NvEspInnerProtocolType l3_ether_type,
    @optional in nv_vlan_id_t vid
);
```

Parameters:

- **headers** [in/out]: The packet headers that are to be decapsulated by removing the ESP headers.
- **icv_length** : The **NvIPsecIcv** enumeration value that specifies length of the ICV.
- **dst_mac** [in]: Destination MAC address.
- **src_mac** [in]: Source MAC address.
- **l3_ether_type** [in]: The layer 3 ethertype protocol.

```
enum nv_inner_protocol_type_t NvEspInnerProtocolType {
  IPV4 = NV_TYPE_IPV4,
  IPV6 = NV_TYPE_IPV6
}
```

- **vid** [in]: An optional VLAN ID to be included in the underlay headers.

The following limitations apply to the IPsec feature:

- IPsec encapsulation does not support IPv4 with options or IPv6 with extension headers.
- IPsec UDP tunnel mode encapsulation/decapsulation is not currently implemented.
- Extended Sequence Numbers (ESN) are not currently implemented.
- SA provisioning via P4Runtime must be performed by the controller application prior to configuration of the IPsec objects.
- The hardware assumes the presence of an explicit IV that precedes the ESP payload.

8.11.7.1.7.32 7.32. nv_encap_psp_transport

Description:
Extern function that encapsulates a clear text packet with PSP transport mode headers, with an optional virtualization cookie. The resulting packet is ready for PSP encryption.

Signature:

```
void nv_encap_psp_transport<HEADERS>(
    inout HEADERS headers,
    // PSP modify header pattern
    in bit<16> udp_src_port,
    in bit<6> crypt_offset,
    in bit<1> needs_sampling,
    in bit<1> drop,
    in NvPSPVersion version,
    in bit<32> spi,
    in bit<64> init_vector,
    @optional in bit<64> virt_cookie   // optional
)
```

Parameters:

- **headers** [in/out]: The packet headers that are to be encapsulated with a PSP headers.
- **udp_src_port** [in]: UDP source port of the PSP transport. Set to zero for hardware to assign.
- **crypt_offset** [in]: The offset from the end of the Initialization Vector to the start of the encrypted portion of the payload, measured in 4-octet units
- **needs_sampling** [in]: Sample at Receiver
- **drop** [in]: Drop after Sampling
- **version** [in]: PSP header version. NvPSPVersion.V0_AES_GCM_128 = 4w0 or NvPSPVersion.V1_AES_GCM_256 = 4w1
- **spi** [in]: A 32-bit value that is used by a receiver to identify the Security Association (SA) to which an incoming packet is bound
- **init_vector** [in]: A unique value for each packet sent over a Security Association
- **virt_cookie** [in]: An optional field, it may contain a Virtual Network Identifier (VNI) or other data.

The UDP destination port is set to 1000 for PSP.

8.11.7.1.7.33 7.33. nv_decap_psp_transport

Description:
Extern function that decapsulates a clear text packet with PSP transport mode headers. This function can only be called on a decrypted PSP packet.

Signature:

```
void nv_decap_psp_transport<HEADERS>(inout HEADERS headers)
```

Parameters:

- **headers** [in/out]: The packet headers that are to be decapsulated and the PSP headers removed.

8.11.7.1.7.34 7.34. nv_encap_psp_v4_tunnel

Description:
Extern function that encapsulates a clear text packet with PSP L3 tunnel mode headers. A new underlay is provided, with an optional underlay VLAN and PSP virtualization cookie. The resulting packet is ready for PSP encryption.

Signature:

```
void nv_encap_psp_v4_tunnel<HEADERS>(
    inout HEADERS headers,
    // encap
    in nv_mac_addr_t dst_mac,
    in nv_mac_addr_t src_mac,
    in nv_ipv4_addr_t sip,
    in nv_ipv4_addr_t dip,
    in bit<16> udp_src_port,
    // PSP
    in NvIPEncapProtocol next_header,
    in bit<6> crypt_offset,
    in bit<1> needs_sampling,
    in bit<1> drop,
    in NvPSPVersion version,
    in bit<32> spi,
    in bit<64> init_vector,
    @optional in nv_vlan_id_t vid,
    @optional in bit<64> virt_cookie   // optional
)
```

Parameters:

- **headers** [in/out]: The packet headers that are to be encapsulated with a PSP headers.
- **dst_mac** [in]: Destination MAC address.
- **src_mac** [in]: Source MAC address.
- **src_ip** [in]: Source IPv4 address.
- **dst_ip** [in]: Destination IPv4 address.
- **udp_src_port** [in]: UDP source port of the PSP transport. Set to zero for hardware to assign.
- **next_header** [in]: IP protocol number, identifying the type of the next header. NvIPEncapProtocol.IPV4 or NvIPEncapProtocol.IPv6
- **crypt_offset** [in]: The offset from the end of the Initialization Vector to the start of the encrypted portion of the payload, measured in 4-octet units
- **needs_sampling** [in]: Sample at Receiver
- **drop** [in]: Drop after Sampling
- **version** [in]: PSP header version. NvPSPVersion.V0_AES_GCM_128 = 4w0 or NvPSPVersion.V1_AES_GCM_256 = 4w1
- **spi** [in]: A 32-bit value that is used by a receiver to identify the Security Association (SA) to which an incoming packet is bound
- **init_vector** [in]: A unique value for each packet sent over a Security Association
- vid [in]: An optional VLAN ID to be included in the underlay headers.
- **virt_cookie** [in]: An optional field, it may contain a Virtual Network Identifier (VNI) or other data.

The UDP destination port is set to 1000 for PSP.

8.11.7.1.7.35 7.35. nv_encap_psp_v6_tunnel

Description:
Extern function that encapsulates a clear text packet with PSP L3 tunnel mode headers. A new underlay is provided, with an optional underlay VLAN and PSP virtualization cookie. The resulting packet is ready for PSP encryption.

Signature:

```
void nv_encap_psp_v6_tunnel<HEADERS>(
    inout HEADERS headers,
    // encap
    in nv_mac_addr_t dst_mac,
    in nv_mac_addr_t src_mac,
```

```
    in nv_ipv6_addr_t sip,
    in nv_ipv6_addr_t dip,
    in bit<16> udp_src_port,
    // PSP
    in NvIPEncapProtocol next_header,
    in bit<6> crypt_offset,
    in bit<1> needs_sampling,
    in bit<1> drop,
    in NvPSPVersion version,
    in bit<32> spi,
    in bit<64> init_vector,
    @optional in nv_vlan_id_t vid,
    @optional in bit<64> virt_cookie   // optional
)
```

Parameters:

- **headers** [in/out]: The packet headers that are to be encapsulated with a PSP headers.
- **dst_mac** [in]: Destination MAC address.
- **src_mac** [in]: Source MAC address.
- **src_ip** [in]: Source IPv6 address.
- **dst_ip** [in]: Destination IPv6 address.
- **udp_src_port** [in]: UDP source port of the PSP transport. Set to zero for hardware to assign.
- **next_header** [in]: IP protocol number, identifying the type of the next header. NvIPEncapProtocol.IPV4 or NvIPEncapProtocol.IPv6
- **crypt_offset** [in]: The offset from the end of the Initialization Vector to the start of the encrypted portion of the payload, measured in 4-octet units
- **needs_sampling** [in]: Sample at Receiver
- **drop** [in]: Drop after Sampling
- **version** [in]: PSP header version. NvPSPVersion.V0_AES_GCM_128 = 4w0 or NvPSPVersion.V1_AES_GCM_256 = 4w1
- **spi** [in]: A 32-bit value that is used by a receiver to identify the Security Association (SA) to which an incoming packet is bound
- **init_vector** [in]: A unique value for each packet sent over a Security Association
- **vid** [in]: An optional VLAN ID to be included in the underlay headers.
- **virt_cookie** [in]: An optional field, it may contain a Virtual Network Identifier (VNI) or other data.

The UDP destination port is set to 1000 for PSP.

8.11.7.1.7.36 7.36. nv_decap_psp_tunnel

Description:
Extern function that decapsulates a clear text packet with PSP tunnel mode headers. This function can only be called on a decrypted PSP packet.

Signature:

```
void nv_decap_psp_tunnel<HEADERS>(
    inout HEADERS headers,
    // decap headers
    in nv_mac_addr_t dst_mac,
    in nv_mac_addr_t src_mac,
    in NvPspInnerProtocolType l3_ether_type,
    @optional in nv_vlan_id_t vid
)
```

Parameters:

- **headers** [in/out]: The packet headers that are to be decapsulated and the PSP headers removed.
- **dst_mac** [in]: Destination MAC address.
- **src_mac** [in]: Source MAC address.
- **l3_ether_type** [in]: The ethertype of the inner PSP header/payload. NvPspInnerProtocolType.IPV4 or NvPspInnerProtocolType.IPV6
- **vid** [in]: An optional VLAN ID to be included in the overlay header.

8.11.7.1.7.37 7.37. nv_mirror

Description:
Extern function that duplicates the packet, sending each to a different port. Hence this extern is a terminal action.

Signature:

```
void nv_mirror(in nv_logical_port_t vport, in nv_logical_port_t mirror_port)
```

Parameters:

- **vport** [in]: Destination for the first copy of the packet
- **mirror_port** [in]: Destination for the second copy of the packet

8.11.7.1.7.38 7.38. nv_mirror_vxlan_v4_to_remote

Description:
Extern function that duplicates the packet, sending each to a different port, with the copy being sent to the mirror port encapsulated with a VXLAN IPV4 underlay

Signature:

```
void nv_mirror_vxlan_v4_to_remote(in nv_logical_port_t vport, in nv_logical_port_t mirror_port, in nv_mac_addr_t
dst_mac, in nv_mac_addr_t src_mac, in nv_ipv4_addr_t sip, in nv_ipv4_addr_t dip, in nv_vxlan_id_t vni, @optional in
nv_vlan_id_t vid)
```

Parameters:

- **vport** [in]: Destination for the unmodified copy of the packet
- **mirror_port** [in]: The destination for the encapsulated copy of the packet.
- **dst_mac** [in]: Destination MAC address.
- **src_mac** [in]: Source MAC address.
- **sip** [in]: Source IPv4 address.
- **dip** [in]: Destination IPv4 address.
- **vni** [in]: VXLAN Network Identifier.
- **vid** [in]: Optional VLAN ID.

8.11.7.1.7.39 7.39. nv_mirror_vxlan_v6_to_remote

Description:
Extern function that duplicates the packet, sending each to a different port, with the copy being sent to the mirror port encapsulated with a VXLAN IPv6 underlay.

Signature:

```
void nv_mirror_vxlan_v6_to_remote(in nv_logical_port_t vport, in nv_logical_port_t mirror_port, in nv_mac_addr_t
dst_mac, in nv_mac_addr_t src_mac, in nv_ipv6_addr_t sip, in nv_ipv6_addr_t dip, in nv_vxlan_id_t vni, @optional in
nv_vlan_id_t vid)
```

Parameters:

- **vport** [in]: Destination for the unmodified copy of the packet.
- **mirror_port** [in]: Destination for the encapsulated copy of the packet.
- **dst_mac** [in]: Destination MAC address.
- **src_mac** [in]: Source MAC address.
- **sip** [in]: Source IPv6 address.
- **dip** [in]: Destination IPv6 address.
- **vni** [in]: VXLAN Network Identifier.

8.11.7.1.7.40 7.40. nv_mirror_gre_v4_to_remote

Description:
Extern function that duplicates the packet, sending each to a different port, with the copy being sent to the mirror port encapsulated with a GRE IPv4 underlay, including optional VLAN tagging and GRE key.

Signature:

```
void nv_mirror_gre_v4_to_remote(in nv_logical_port_t vport, in nv_logical_port_t mirror_port, in nv_mac_addr_t
src_mac, in nv_mac_addr_t dst_mac, in nv_ipv4_addr_t src_ip, in nv_ipv4_addr_t dst_ip, in NvInnerProtocolType
proto_type, @optional in nv_vlan_id_t vid, @optional in bit<32> key)
```

Parameters:

- **vport** [in]: Destination for the unmodified copy of the packet.
- **mirror_port** [in]: Destination for the encapsulated copy of the packet.
- **src_mac** [in]: Source MAC address.
- **dst_mac** [in]: Destination MAC address.
- **src_ip** [in]: Source IPv4 address.
- **dst_ip** [in]: Destination IPv4 address.
- **proto_type** [in]: Protocol type for the inner payload.
- **vid** [in]: Optional VLAN ID.
- **key** [in]: Optional GRE key.

8.11.7.1.7.41 7.41. nv_mirror_gre_v6_to_remote

Description:
Extern function that duplicates the packet, sending each to a different port, with the copy being

sent to the mirror port encapsulated with a GRE IPv6 underlay, including optional VLAN tagging and GRE key.

Signature:

```
void nv_mirror_gre_v6_to_remote(in nv_logical_port_t vport, in nv_logical_port_t mirror_port, in nv_mac_addr_t
src_mac, in nv_mac_addr_t dst_mac,, in nv_ipv6_addr_t sip, in nv_ipv6_addr_t dip, in NvInnerProtocolType
proto_type, @optional in nv_vlan_id_t vid, @optional in bit<32> key)
```

Parameters:

- **vport** [in]: Destination for the unmodified copy of the packet.
- **mirror_port** [in]: Destination for the encapsulated copy of the packet.
- **src_mac** [in]: Source MAC address.
- **dst_mac** [in]: Destination MAC address.
- **sip** [in]: Source IPv6 address.
- **dip** [in]: Destination IPv6 address.
- **proto_type** [in]: Protocol type for the inner payload.
- **vid** [in]: Optional VLAN ID.
- **key** [in]: Optional GRE key.

8.11.7.1.7.42 7.42. nv_mirror_geneve_v4_to_remote

Description:
Extern function that duplicates the packet, sending each to a different port, with the copy being sent to the mirror port encapsulated with a Geneve IPv4 underlay, including optional VLAN tagging and Geneve options.

Signature:

```
void nv_mirror_geneve_v4_to_remote(in nv_logical_port_t vport, in nv_logical_port_t mirror_port, in nv_mac_addr_t
dst_mac, in nv_mac_addr_t src_mac, in nv_ipv4_addr_t src_ip, in nv_ipv4_addr_t dst_ip, in bit<6> opt_fields_len, in
bit<1> oam, in bit<1> critical, in NvInnerProtocolType proto_type, in bit<24> vni, @optional in nv_vlan_id_t vid,
@optional in bit<16> opt_class, @optional in bit<8> opt_type, @optional in bit<5> opt_len, @optional in bit<32>
opt_data)
```

Parameters:

- **vport** [in]: Destination for the unmodified copy of the packet.
- **mirror_port** [in]: Destination for the encapsulated copy of the packet.
- **dst_mac** [in]: Destination MAC address.
- **src_mac** [in]: Source MAC address.
- **src_ip** [in]: Source IPv4 address.
- **dst_ip** [in]: Destination IPv4 address.
- **opt_fields_len** [in]: Length of the options field in 4-byte units (must be set to 0 if no options are set, otherwise must be set to 2)
- **oam** [in]: OAM flag.
- **critical** [in]: Critical options flag.
- **proto_type** [in]: Protocol type for the inner payload.
- **vni** [in]: Virtual Network Identifier.
- **vid** [in]: Optional VLAN ID.

- **opt_class** [in]: Optional GENEVE option class.
- **opt_type** [in]: Optional GENEVE option type.
- **opt_len** [in]: Optional GENEVE option length in 4-byte units (must be 1 if set).
- **opt_data** [in]: Optional GENEVE option data.

8.11.7.1.7.43 7.43. nv_mirror_geneve_v6_to_remote

Description:
Extern function that duplicates the packet, sending each to a different port, with the copy being sent to the mirror port encapsulated with a Geneve IPv6 underlay, including optional VLAN tagging and Geneve options.

Signature:

```
void nv_mirror_geneve_v6_to_remote(in nv_logical_port_t vport, in nv_logical_port_t mirror_port, in nv_mac_addr_t dst_mac, in nv_mac_addr_t src_mac, in nv_ipv6_addr_t src_ip, in nv_ipv6_addr_t dst_ip, in bit<6> opt_fields_len, in bit<1> oam, in bit<1> critical, in NvInnerProtocolType proto_type, in bit<24> vni, @optional in nv_vlan_id_t vid, @optional in bit<16> opt_class, @optional in bit<8> opt_type, @optional in bit<5> opt_len, @optional in bit<32> opt_data)
```

Parameters:

- **vport** [in]: Destination for the unmodified copy of the packet.
- **mirror_port** [in]: Destination for the encapsulated copy of the packet.
- **dst_mac** [in]: Destination MAC address.
- **src_mac** [in]: Source MAC address.
- **src_ip** [in]: Source IPv6 address.
- **dst_ip** [in]: Destination IPv6 address.
- **opt_fields_len** [in]: Length of the options field in 4-byte units (must be set to 0 if has_option is false, otherwise must be set to 2)
- **oam** [in]: OAM flag.
- **critical** [in]: Critical options flag.
- **proto_type** [in]: Protocol type for the inner payload.
- **vni** [in]: Virtual Network Identifier.
- **vid** [in]: Optional VLAN ID.
- **opt_class** [in]: Optional GENEVE option class.
- **opt_type** [in]: Optional GENEVE option type.
- **opt_len** [in]: Optional GENEVE option length in 4-byte units (must be 1 if set).
- **opt_data** [in]: Optional GENEVE option data.

8.11.7.1.7.44 7.44. nv_send_debug_pkt

Description:
Extern function that adds pipeline metadata and packet state to the packet, then sends it to the DPL Debugger. The packet is forwarded to the DPL debugger and will be reinjected back to the pipeline, at a point immediately following the invocation of **nv_send_debug_pkt**. This action may be invoked within a P4 action block or a control's apply block.

Signature:

```
void nv_send_debug_pkt(in nv_debug_cookie_t cookie = 8w0x0)
```

Parameters:

- **cookie** [in]: An optional cookie value can be provided by the user, that will show up in the DPL Debugger with the debug packet

Automatic Debug Points (**--debug-mode auto**)

When the compiler is invoked with **--debug-mode auto** , the DPL compiler now inserts automatic debug points at the end of each P4 action, in addition to the existing debug points before each statement in control **apply** blocks.

This behavior is implemented by inserting an internal **DebugInstruction** (equivalent to **nv_send_debug_pkt(cookie = 0)**) immediately before the terminal instruction of each action.

Placement Rules

For control body, a debug point is inserted prior to each statement. For each action body:

- If the last instruction in the action is effectively a RETURN or EXIT (for example, **nv_send_to_port** , **nv_drop**):
  - A debug point is inserted immediately before that terminal instruction.
  - For actions that ultimately call a terminal extern (e.g., **nv_send_to_port**), the debug point observes packet state after all field updates in the action, but before the packet leaves the pipeline.
- If the action is empty or its last instruction is not terminal, no automatic debug point is added.

Interaction with Manual **nv_send_debug_pkt**

If you explicitly call **nv_send_debug_pkt()** in an action:

- When a manual debug call is placed immediately before the terminal instruction, the compiler does not insert another automatic debug point at that same location, avoiding duplicate debug packets. A warning may be issued to indicate that the automatic point was suppressed.
- Manual debug calls elsewhere in the action are left unchanged. The automatic debug point is still inserted at the end unless it would be adjacent to an existing manual call, as described above.

Debugger Behavior

From the debugger's perspective:

- Each action execution yields a debug point at a consistent location:
  - After the action has finished updating headers and metadata.
  - Before any terminating extern that drops or sends the packet.
- This makes it easier to correlate:
  - Table hits → selected actions → final per-action state, and
  - Multiple debug points along the packet path, including across repeated invocations of the same action.

Limitations

You must enable debug mode by compiling your DPL program with the `-g/--debug-mode` flag in addition to calling `nv_send_debug_pkt`. Debugging is not supported for P4 tables larger than 8M in size. To debug packets from both the primary wire port P0 and the second wire port P1 requires configuring the DPU in multi-port eswitch mode. When using the automatic debug point feature, a `nv_send_debug_pkt` will override the automatic debug point, with the specified cookie value.

See DOCA Pipeline Language Developer Tool for more details.

8.11.7.1.7.45 7.45. nv_add_entry

Description:
Extern function that inserts a table entry via a data plane insertion mechanism. After invoking `nv_add_entry`, a status code is returned and the packet continues on the pipeline.

Signature:

```
NvAddEntryErrorStatus_t nv_add_entry(<K, A>(
    string table_name,
    in K keys,
    string action_name,
    in A action_params,
    in bit<32> default_timeout_ms = 0
);
```

Parameters:

- `table_name` [in]: The name of the table to which the entry will be added. The table and the invocation of the extern must be in the scope of the same control.
- `keys` [in]: A list of keys, consisting of the supported packet header fields. Only the following list of packet fields are permitted to be used as values for either the key list or the action parameter value list. Runtime variables are also allowed:

Header	Field
ethernet	dst_addr
ethernet	src_addr
std_meta	last_l2_ether_type
mpls	label
mpls	tc
mpls	bos
mpls	ttl
ipv4	src_addr
ipv4	dst_addr
ipv4	ttl
ipv4	diffserv
ipv6	hop_limit

Header	Field
ipv6	diffserv
ipv6	src_addr
ipv6	dst_addr
tcp	ecn
tcp	flags
tcp	src_port
tcp	dst_port
udp	src_port
udp	dst_port

- **action_name** [in]: The name of the action that should be applied if the entry matches. The action must be in the action list declared for the table.
- **action_params** [in]: A list of action parameter values. The number and order of values must match the parameters defined for the action. Values may be compile time constants, a runtime variable or a supported packet header field. The total width of all action parameters is limited to 11 bytes. Parameters that are not byte aligned are padded up to the next byte boundary.
- **default_timeout_ms** [in]: The default timeout for the entry, in milliseconds. Default value is zero, meaning no timeout will be applied.

Returns:

- A bit<1> **NvAddEntryErrorStatus_t** value is returned to the application:
  - ADD_ENTRY_SUCCESS = 0
  - ADD_ENTRY_FAILURE = 1

The table attribute **nv_support_add_entry** must be set to **true** for each table to which this extern applies.

If a non zero timeout is specified, then the table attribute **nv_support_timeout** must be set to true.

Supported usage:

nv_add_entry can only be called from the control's apply block. In particular, it cannot be called from within a P4 action body. For example:

```
apply {
    if (dst_table.apply().miss) {
        user_meta.ret = nv_add_entry("dst_table",
            { hdr.ethernet.dst_addr, hdr.ipv6.dst_addr },
            "forward",
            { 4 },
            5000 // 5 seconds
        );
        if (user_meta.ret == 0) {
            nv_send_to_port(4);
        }
    }
    // continue with rest of pipeline processing...
```

```
}
```

Limitations:

> **nv_add_entry** feature is a beta level feature in the current release. As such, there are number of restrictions on its usage and runtime behavior. The listed limitations reflect this and it is expected that this feature will be expanded in future releases. Not all combinations of the packet fields listed above are supported due to resource limitations. Since this feature is dependent on runtime resources, the compiler may not always issue an error. Instead, the application blob may fail to load with a DPL RT daemon error log message.

8.11.7.1.7.46 7.46. nv_get_random_value

Extern function that returns a 16 bit random value from the hardware.

Signature:

```
extern bit<16> nv_get_random_value();
```

Parameters:

- **bit<16>** : a random value between 0 and 65535

8.11.7.1.7.47 7.47. nv_get_ut_clock

Extern function that returns a UTC timestamp from the device's Real-Time clock. The MSB of the 64-bit timestamp the time in seconds. The LSB of the 64-bit timestamp holds the time in nanoseconds.

Signature:

```
extern bit<64> nv_get_ut_clock();
```

Parameters:

- **bit<64>** : The UTC time stamp.
  - Seconds in the upper 32 bits
  - Nano-seconds in the lower 32 bits

8.11.7.1.7.48 7.48. nv_get_fr_clock

Extern function that returns the free running clock from the device's internal timer.

Signature:

```
extern bit<64> nv_get_fr_clock();
```

Returns:

- **bit<64>** : The free running clock, in units of 1/device frequency.

⚠ The following MFT tool can report the device's clock frequency:

`sudo mlxuptime -d /dev/mst/<device_name>_pciconf`

8.11.7.1.8 8. DOCA Extern Objects

8.11.7.1.8.1 8.1. Counters

BlueField supports per entry counters, shared and direct. A (shared) counter is an extern object that allows the user to access N independent counters via an index. Whereas a direct counter is an extern object that is directly associated with a P4 table where the index is implied by the entry slot. For both types, the counter value is updated when an action calls the count() method.

8.1.1. NvCounter

Instantiates an indirect counter of the specified width and type.

Constructor:

```
extern NvCounter(bit<32> size, NvCounterType type)
```

Parameters:

- **size[in]** : Number of counter indices. The size must be a power of 2, and no greater than 16M.
- **type[in]** : Enum indicating counter type. Currently only **NvCounterType.PACKETS_AND_BYTES** is supported.

Methods:

count

Increments the counter at the specified index

```
void count(in bit<32> index)
```

Parameters:

- **index[in]** : Index of the counter to increment

8.1.2. NvDirectCounter

Instantiates a direct counter that can be assigned to a table. Direct counters do not have a separate size, they are as wide as the table. Each action invoked by an entry hit should call the count() method to increment the entry count.

Constructor:

```
extern NvDirectCounter(bit<32> size, NvCounterType type)
```

Parameters:

- **type[in]** : Enum indicating counter type. Currently only **NvCounterType.PACKETS_AND_BYTES** is supported.

Methods:

count

Increments the counter associated the the entry that triggered the action execution

void count()

> ⚠ Methods of a NvDirectCounter object may only be called by the P4 Table that owns it via the "direct_counter" property bound to the P4 Table. It is an error to attempt to call the count() method from a P4 Table that does not own the direct counter. While the system supports up to 16M total counters, a small number of counters may be reserved by the kernel.

> ⚠ If there are no side effects, it is most efficient to call the count method as the first primitive action in a P4 action.

8.11.7.1.8.2 8.2. Meters

BlueField supports per-flow metering, following RFC2697, RFC2698, and RFC4115. Similar to counters, both direct meters and shared meters are supported in the DPL target architecture. The packets for a given flow can be colored by the meter into three "colors": red, yellow and green. The meter burst parameters can be specified with units of bytes or packets.

```
typedef bit<8> nv_meter_color_t;

enum nv_meter_color_t NvMeterColor {
    RED = 8w0,
    YELLOW = 8w1,
    GREEN = 8w2,
}

enum NvMeterUnits {
  BYTES,
  PACKETS
}
```

The meter color indicators are:

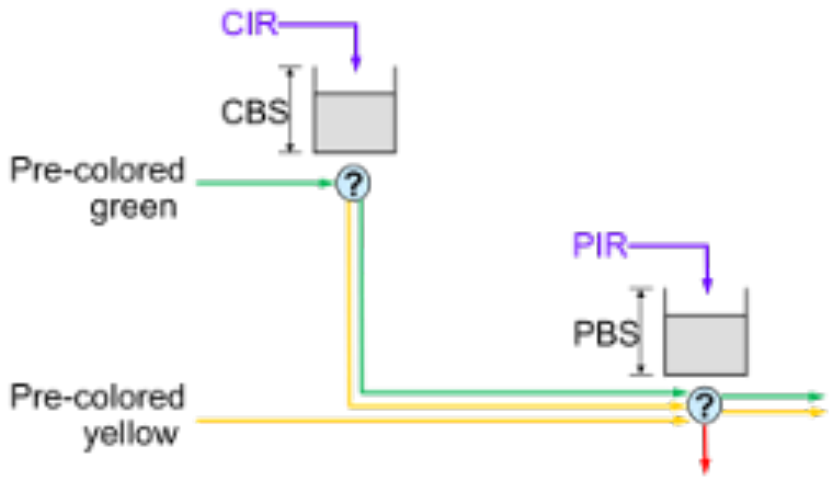
- **GREEN** : The packet conforms to the Committed Information Rate (CIR) and fits within the Committed Burst Size (CBS). It is fully within the agreed traffic profile.
- **YELLOW** : The packet exceeds the CIR but is within the Excess/Peak Information Rate and fits within the Excess Burst Size.
- **RED** : The packet exceeds the information rate or burst sizes and is considered out-of-profile, typically indication the packet should be dropping or set to a lower priority.

The meter units are defined as:

- **BYTES** : packet bytes as indicated by IP.length
- **PACKETS** : number of packets (128 bytes per packet)

8.2.1. NvMeterPeakTrTCM

The **NvMeterPeakTrTCM** extern object Implements RFC 2698.



Constructor:

Instantiates a Two Rate Three Color Marker shared meter object of the specified size, units and burst parameters.

```
extern NvMeterPeakTrTCM(bit<32> size, NvMeterUnits units, bit<64> cir, bit<64> cbs, bit<64> pir, bit<64> pbs)
```

Parameters:

- **size[in]** : Number of meter indices. The size must be a power of 2, and no greater than 16M.
- **units[in]** : Enum indicating the units used by the meter. Metering by **NvMeterUnits.BYTES** or by **NvMeterUnits.PACKETS** is supported.
- **cir[in]** : Committed Information Rate
- **cbs[in]** : Committed Burst Size
- **pir[in]** : Peak Information Rate
- **pbs[in]** : Peak Burst Size

Constructor:

Instantiates a Two Rate Three Color Marker shared meter object of the specified size with a default initial state of GREEN. The bucket parameters can be updated per entry at runtime.

```
extern NvMeterPeakTrTCM(bit<32> size, NvMeterUnits units)
```

Parameters:

- **size[in]** : Number of meter indices. The size must be a power of 2, and no greater than 16M.
- **units[in]** : Enum indicating the units used by the meter. Metering by **NvMeterUnits.BYTES** or by **NvMeterUnits.PACKETS** is supported.

Methods:

meter

Executes the metering of the flow entry at the specified index.

```
nv_meter_color_t meter(in bit<32> index, nv_meter_color_t initial_color = NvMeterColor.GREEN)
```

Parameters:

- **index[in]** : Index of the meter to execute
- **initial_color[in]** : The initial color to assign to the flow. Coloring of **NvMeterColor.RED** , **NvMeterColor.YELLOW** , or **NvMeterColor.GREEN** . The default is **NvMeterColor.GREEN** .

Returns:

- The **nv_meter_color_t** of the flow entry at the specified index.

8.2.2. NvDirectMeterPeakTrTCM

The **NvDirectMeterPeakTrTCM** extern object implements RFC 2698. A direct meter must be assigned to a table and be the same size as the table. Each action invoked by an entry hit should call the **meter()** method to perform the policing.

Constructor:

Instantiates a Two Rate Three Color Marker direct meter object of the specified size, units and burst parameters.

```
extern NvDirectMeterPeakTrTCM(NvMeterUnits units, bit<64> cir, bit<64> cbs,  bit<64> pir, bit<64> pbs)
```

Parameters:

- **units[in]** : Enum indicating the units used by the meter. Metering by **NvMeterUnits.BYTES** or by **NvMeterUnits.PACKETS** is supported.
- **cir[in]** : Committed Information Rate
- **cbs[in]** : Committed Burst Size
- **pir[in]** : Peak Information Rate
- **pbs[in]** : Peak Burst Size

Constructor:

Instantiates a Two Rate Three Color Marker direct meter object of the specified size with a default initial state of GREEN. The bucket parameters can be updated per entry at runtime.

```
extern NvDirectMeterPeakTrTCM(NvMeterUnits units)
```

Parameters:

- **units[in]** : Enum indicating the units used by the meter. Metering by **NvMeterUnits.BYTES** or by **NvMeterUnits.PACKETS** is supported.

Methods:

meter

Executes the metering of the flow entry at the specified index.

```
nv_meter_color_t meter(nv_meter_color_t initial_color = NvMeterColor.GREEN)
```

Parameters:

- **initial_color[in]** : The initial color to assign to the flow. Coloring of **NvMeterColor.RED** , **NvMeterColor.YELLOW** , or **NvMeterColor.GREEN** . The default is **NvMeterColor.GREEN** .

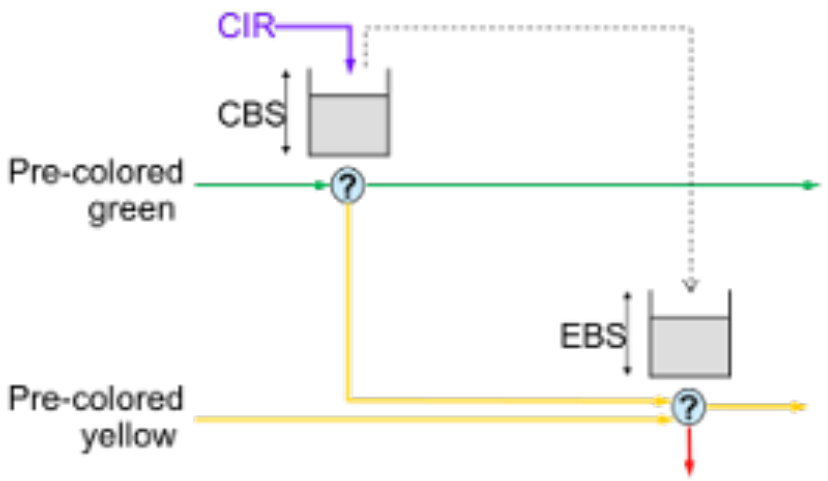
Returns:

- The **nv_meter_color_t** of the flow entry at the specified index.

> **Rate and Size Max Values**
>
> Each configuration parameter of **NvMeterPeakTrTCM** and **NvDirectMeterPeakTrTCM** is restricted to certain max values.
>
> When metering by BYTES, max values are the following:
> - cir: 255000000000
> - cbs: 2147483647
> - pir: 510000000000
> - pbs: 4294967295
>
> When metering by PACKETS, max values are the following:
> - cir: 1992187500
> - cbs: 16777215
> - pir: 3984375000
> - pbs: 33554431

8.2.3. NvMeterSrTCM

The **NvMeterSrTCM** extern object implements RFC 2697.



Constructor:

Instantiates a Single Rate Three Color Marker shared meter object of the specified size, units and burst parameters.

```
extern NvMeterSrTCM(bit<32> size, NvMeterUnits units, bit<64> cir, bit<64> cbs, bit<64> ebs)
```

Parameters:

- **size[in]** : Number of meter indices. The size must be a power of 2, and no greater than 16M.
- **units[in]** : Enum indicating the units used by the meter. Metering by **NvMeterUnits.BYTES** or by **NvMeterUnits.PACKETS** is supported.
- **cir[in]** : Committed Information Rate
- **cbs[in]** : Committed Burst Size
- **ebs[in]** : Excess Burst Size

Constructor:

Instantiates a Single Rate Three Color Marker shared meter object of the specified size with a default initial state of GREEN. The bucket parameters can be updated per entry at runtime.

```
extern NvMeterSrTCM(bit<32> size, NvMeterUnits units)
```

Parameters:

- **size[in]** : Number of meter indices. The size must be a power of 2, and no greater than 16M.
- **units[in]** : Enum indicating the units used by the meter. Metering by **NvMeterUnits.BYTES** or by **NvMeterUnits.PACKETS** is supported.

Methods:

meter

Executes the metering of the flow entry at the specified index

```
nv_meter_color_t meter(in bit<32> index, nv_meter_color_t initial_color = NvMeterColor.GREEN)
```

Parameters:

- **index[in]** : Index of the meter to execute
- **initial_color[in]** : The initial color to assign to the flow. Coloring of **NvMeterColor.RED** , **NvMeterColor.YELLLOW** , or **NvMeterColor.GREEN** . The default is **NvMeterColor.GREEN** .

Returns:

- The **nv_meter_color_t** of the flow entry at the specified index.

8.2.4. NvDirectMeterSrTCM

The **NvDirectMeterSrTCM** extern object implements RFC 2697. A direct meter must be assigned to a table and be the same size as the table. Each action invoked by an entry hit should call the **meter()** method to perform the policing.

Constructor:

Instantiates a Single Rate Three Color Marker direct meter object of the specified size, units and burst parameters.

```
extern NvDirectMeterSrTCM(NvMeterUnits units, bit<64> cir, bit<64> cbs, bit<64> ebs)
```

Parameters:

- **units** : Enum indicating the units used by the meter. Metering by **NvMeterUnits.BYTES** or by **NvMeterUnits.PACKETS** is supported.
- **cir** : Committed Information Rate
- **cbs** : Committed Burst Size
- **ebs** : Excess Burst Size

Constructor:

Instantiates a Single Rate Three Color Marker direct meter object of the specified size with a default initial state of GREEN. The bucket parameters can be updated per entry at runtime.

```
extern NvDirectMeterSrTCM(NvMeterUnits units)
```

Parameters:

- **units** : Enum indicating the units used by the meter. Metering by **NvMeterUnits.BYTES** or by **NvMeterUnits.PACKETS** is supported.

Methods:

meter

Executes the metering of the flow entry at the specified index.

```
nv_meter_color_t meter(in bit<32> index, in nv_meter_color_t initial_color = NvMeterColor.GREEN)
```

Parameters:

- **index[in]** : Index of the meter to execute
- **initial_color[in]** : The initial color to assign to the flow. Coloring of **NvMeterColor.RED** , **NvMeterColor.YELLLOW** , or **NvMeterColor.GREEN** . The default is **NvMeterColor.GREEN** .

Returns:

- The **nv_meter_color_t** of the flow entry at the specified index.

⚠ Rate and Size Max Values

Each configuration parameter of NvMeterSrTCM and NvDirectMeterSrTCM is restricted to certain max values.

When metering by BYTES, max values are the following:

- cir: 255000000000
- cbs, ebs: 2147483647

When metering by PACKETS, max values are the following:

- cir: 1992187500
- cbs, ebs: 16777215

⚠ Methods of a **NvDirectMeterPeakTrTCM** and a **NvDirectMeterSrTCM** object may only be called by the P4 Table that owns it via the **direct_meter** property bound to the P4 Table. It is an error to attempt to call the **meter()** method from a P4 Table that does not own the direct meter. A meter may not be owned by multiple P4 Tables. The system supports up to 16M total meters.

8.11.7.1.8.3 8.3. Options Parser

DPL supports TLV style parsing for protocols with options.

8.3.1. NvOptionsParser

The NvOptionsParser is an extern object used to define the behavior of the TLV (Type-Length-Value) options parser.

```
extern NvOptionParser<VALUE, OPTION> {
    NvOptionParser(
        string options_length_field = "",
        bit<32> options_length_shift = 32w0,
        bit<32> options_length_add = 32w0,
        string option_layout_header_type = "",
        string option_length_field = "",
        bit<32> option_length_shift = 32w0,
        bit<32> option_length_add = 32w0,
        string option_type_field = "",
        list<tuple<VALUE, OPTION>> options
    );

    void parseOptions<H>(packet_in p, inout H headers);
}
```

Parameters:

- **options_length_field** [in]: Optional; omit if *total* options length is fixed, otherwise provide the field name that contains the header length value
- **options_length_shift** : Optional; omit if the field can be used without shifting
- **options_length_add** : Optional; omit if the field can be used without adding
- **option_layout_header_type** : Mandatory **header_type_name**
- **option_length_field** : Optional; omit if each option is a constant size, otherwise provide the field name that contains the header length value
- **option_length_shift** : Optional; omit if the field can be used without shifting
- **option_length_add** : Optional; omit if the field can be used without adding
- **option_type_field** : Mandatory **field_name**

- **options** : Mandatory; variable-length (expression-)list of (type, header field)

Methods:

parseOptions

Parses the TLV

Signature: **void parseOptions<H>(packet_in packet, inout H headers);**

Parameters:

- packet[in]: input packet
- headers[in/out]: packet headers struct

Examples:

See Geneve TLV Parsing Example.

8.11.7.1.8.4 8.4. Custom Tunnel Encapsulation and Header Insertion

DPL supports the creation of L2 and L3 tunnels using custom tunnel headers, as well as inserting custom headers at specific anchor points in the header stack.

8.4.1. NvHeaderDataTemplate

Instantiates a template object used to inserting custom data to the header stack (e.g., tunnel encapsulations or pushing new headers). This object is used for controlling the behavior of the actions that insert data into the header at predefined locations.

```
extern NvHeaderDataTemplate<HEADERS_DATA_TYPE> {
    NvHeaderDataTemplate();
}
```

Type Variables:

- **HEADERS_DATA_TYPE[in]** : Must be the typename of a user-declared **struct** whose only fields must be of **header** type. The entire set of underlay headers should be defined as a struct. For example:

```
header my_tunnel_h {
    bit<32> field_1;
    bit<32> field_2;
    bit<32> field_3;
}

struct tunnel_headers_t {
    nv_ethernet_h ethernet;
    my_tunnel_h   custom_tunnel;
}
```

Annotation:

NvHeaderDataTemplate works slightly differently than other extern objects in that the object has no methods; instead, the object can be passed into the extern functions:

- **nv_set_l2tunnel_underlay**
- **nv_set_l3tunnel_underlay**
- **nv_push_headers**

Declaring a **NvHeaderDataTemplate** object does not use any resources in hardware, as the template construct is provided for simplifying the process of specifying header field values for custom tunnels. When declaring a **NvHeaderDataTemplate** object, the annotation **@nv_header_data_fields** must be present. This annotation contains a key-value entry for each header in the specified struct type. For each header, the header fields must be specified in the order that they appear in the header definition. The value assigned to a field must be one of the following:

- a non-negative integer
- the string "variable"
- the string "ignore" (only for the tunnel externs)

```
@nv_header_data_fields(
    ethernet = {
        dst_addr = "variable",
        src_addr = "variable",
        ether_type = 0xABCD
    },
    custom_tunnel = {
        field_1 = 1,
        field_2 = "ignore",
        field_3 = "variable"
    }
)
NvHeaderDataTemplate<tunnel_headers_t>() my_tunnel;
```

When using a list expression in an action that uses a **NvHeaderDataTemplate** the following restrictions apply:

- The length of the list must match the number of header fields marked "variable" in the **NvTunnelTemplate** object's annotation
- The order of the header fields determines the order of the values in the list expression (e.g., 3rd header field marked "variable" will be set to the 3rd value in the list)
- Each value is required to be runtime constant, meaning it may either be an integer value or an action parameter value
- Unlike the list expressions used in the @nv_header_data_fields annotation, named parameters (e.g. *field_3 = "variable"*) are not supported in the list expression used as an action parameters.

For a complete example, see GTP Tunnel Encapsulation Example.

Header fields overwritten by Encapsulation

Certain header fields will automatically be recalculated by the hardware after encapsulation. These fields cannot be assigned by the user. It is good practice to mark them as "ignore".

Header Field	Value
ipv4.ecn	0
ipv4.identification	0

Header Field	Value
ipv4.hdr_checksum	Hardware calculated
ipv4.total_len	Hardware calculated
ipv6.traffic_class (ECN bits only)	0
ipv6.payload_length	Hardware calculated
udp.src_port	Entropy hash
udp.length	Hardware calculated
udp.checksum	0
tcp.checksum	Hardware calculated
gre.key (8 LSB only)	Entropy hash

8.4.2. NvTunnelTemplate

This extern object is deprecated and may be removed in a future release. Please use NvHeaderDataTemplate instead.

8.11.7.1.8.5 8.5. Custom Hash

The DOCA Target Architecture allows the user to define a set of key fields to calculate a hash function on and use the resulting hash value in subsequent match/action tables.

8.5.1. NvHashAlgorithm

The supported native hash functions are listed in the following enumeration:

```
enum NvHashAlgorithm {
    CRC32
}
```

8.5.2. NvHash

The NvHash object is initialized by selecting the hash type. The method can invoked by supplying the fields to calculate the hash value, with the result stored in a variable.

Constructor:

```
NvHash(NvHashAlgorithm algo)
```

Parameters:

- **algo** : The hash algorithm to use.

Methods:

get_hash

Computes the hash value from the specified key fields

```
bit<32> get_hash<DATA>(in bit<32> size, in DATA data)
```

Parameters:

- **size[in]** : Number of hash buckets. The hash value is divided by size to get modulo. The size must be a power of 2, and no greater than 16M.
- **data[in]** : The data over which to calculate the hash. Provided as a list of key fields.

Returns:

- **bit<32>** of the hash value modulo size.

Limitations:

The use of NvHash is limited to the apply block of a P4 control. This extern cannot be invoked from a P4 action body.

8.11.7.1.8.6 8.6. TCP State

The DOCA Target Architecture provides a stateful object used to track TCP sequence, acknowledgement, and direction data for a given TCP flow. A flow is defined by a standard 5-tuple consisting of:

- IP source address
- IP destination address
- IP Protocol (IPv4) or Next Header (IPv6)
- TCP source port
- TCP destination port

TCP flows are connection-oriented and bi-directional (established between two applications with distinct IP addresses and ports). To manage this, two hash tuples (lookup keys) are maintained for every connection:

1. Original direction: Packets originating from the endpoint that initiated the connection.
2. Reply direction: Reply packets sent toward the endpoint that initiated the connection.

The TCP connection tracking state is fully maintained in hardware. The packet processing workflow is as follows:

1. Search: Upon receiving a packet with a TCP 5-tuple, the dataplane searches for a matching connection entry.
2. Retrieve: If a connection entry is found, the TCP State object is retrieved.
3. Update: The relevant contexts (sequence numbers, window size, etc.) are updated.
4. Pipeline: The packet returns to the steering pipeline for final processing.

The resulting TCP connection states may be queried by the control plane at any time.

The hardware determines which header to use for stateful tracking based on the flow type:

- Non-tunneled flows: The outer TCP header is used.

- UDP tunneled flows: The inner TCP header is used (the hardware parses past the tunnel encapsulation).

8.6.1. NvTcpFlowDirection

TCP flows are by nature bi-directional. **ORIGINAL** denotes the side that initiates the connection via a TCP SYN packet. Reply denotes the responding side (i.e., the side that sends the SYN-ACK):

```
typedef bit<1> nv_tcp_flow_direction_t;

enum nv_tcp_flow_direction_t NvTcpFlowDirection {
    ORIGINAL = 1w0,
    REPLY = 1w1,
}
```

8.6.2. NvTcpState

The **NvTcpState** object is initialized by selecting a size. The method can then be called in either apply block scope or P4 action scope.

Constructor:

```
NvTcpState(bit<32> size)
```

Parameters:

- **size** : The number of indices to allocate for TCP state tracking. The same index should be used to track both directions of the given flow. The size must be a power of 2, and no greater than 16M.

Methods:

store

Store the TCP state of the packet for the given direction.

```
nv_tcp_state_error_t store(in bit<32> index, in NvTcpFlowDirection direction)
```

Parameters:

- **index[in]** : The identifier for a specific TCP state store in the object.
- **direction[in]** : The sender side versus the receiver side of the bi-directional TCP flow.

Returns:

- The **nv_tcp_state_error_t** of the flow entry at the specified index. The error code is a **bit<8>** with the following definition:

Bits	Description
0	Bad Packet
1	State Changed
2..5	reserved

Bits	Description
6..7	0 = valid 1 = invalid 2 = trap

Typically the NvTcpState object would be used as follows:

1. Upon receiving a SYN packet in the pipeline, send the packet to the controller application.
2. Controller adds the TCP 5 tuple entry, with an action that calls the store method with an index and direction of ORIGINAL.
3. Upon receiving the SYN-ACK packet in the pipeline for the same flow, send the packet to the controller application.
4. Controller adds the reverse TCP 5 tuple entry, with an action that calls the store method with the same index as the ORIGINAL and direction of REPLY.
5. Upon receiving the FIN or RST packet in the pipeline for the same flow, send the packet to the controller application.
6. Remove both entries, and the index will be reset and can be reused.

While the TCP ACK/SEQ/direction values are being tracked, the controller can call the query API asynchronously to retrieve the last processed values.

The BlueField hardware expects to see one TCP header in a packet, therefore HW Connection Tracking (CT) can work on the TCP header whether it is present in the outer or inner header. State tracking is based on the first the first TCP header that is parsed.

8.11.7.1.8.7 8.7. IPsec

IPsec is a standards-based encryption and authentication protocol suite used to secure IPv4/IPv6 traffic in both tunnel and transport modes. On BlueField-3/ConnectX-9, IPsec is implemented as a fully offloaded ESP datapath that integrates tightly with the DPL pipeline and crypto engines, enabling line-rate protection for data-center north–south and east–west traffic.

The hardware IPsec engine supports:

- **ESP** as the primary payload protection protocol (AH is considered legacy and not implemented).
- **AES-GCM-128/256** ESP ciphers (AEAD), providing combined encryption and authentication.
- **ICV lengths** of **64, 96, or 128 bits**.
- **Transport mode** (end-to-end protection of the L4 payload) and **tunnel mode** (encapsulation of an entire IP packet).
- **ESP over IP** and **ESP over UDP (NAT-T, dst port 4500)** encapsulations.

The ESP header is parsed natively by the hardware parser. For ESP, the parser extracts at least the **SPI**, **sequence number**, and **next_header** fields; optional IV and ICV fields are not directly matchable and are handled implicitly by the crypto engine. Transitions to ESP are supported from:

- IPv4 / IPv6 with protocol **50 (ESP)**.

- UDP with destination port 4500 for UDP-encapsulated ESP (NAT-T).

As with PSP, IPsec key exchange and policy management are out of scope for DPL. These functions are implemented by the control plane (e.g., IKE daemons such as StrongSwan/Openswan) and Linux xfrm or a custom SPD/SADB implementation. The DPL pipeline assumes that:

- SAs and keys have already been negotiated (SPI, cipher suite, salt, IV policy, lifetimes, replay window, ESN).
- The controller populates the IPsec Offload and DEK objects and programs steering rules via P4Runtime vendor externs before protected traffic is sent through the pipeline.

At the packet level, DPL exposes IPsec via two layers of functionality:

1. ESP encapsulation / decapsulation externs
    - Transport mode: **nv_encap_esp_v4_transport**, **nv_encap_esp_v6_transport**, **nv_encap_udp_esp_v4_transport**, **nv_encap_udp_esp_v6_transport**.
    - Tunnel mode: **nv_encap_esp_v4_tunnel**, **nv_encap_esp_v6_tunnel**, **nv_encap_udp_esp_v4_tunnel**, **nv_encap_udp_esp_v6_tunnel**.
    - Decapsulation: **nv_decap_esp_transport**, **nv_decap_udp_esp_transport**, **nv_decap_esp_tunnel**.
2. Stateful crypto and replay tracking
    - Exposed via the NvIPsec extern and the ESP/IPsec status metadata fields.
    - Not part of the DPL dataplane, but is managed by the controller via the P4Runtime interface.

8.7.1. NvIPsec

The **NvIPsec** extern object provides indexed access to the IPsec offload engine, managing ESP encryption, decryption, and optional stateful features such as sequence-number offload, ESN handling, replay protection, and lifetime enforcement. Internally, each NvIPsec instance is backed by:

- A range of IPSEC_OFFLOAD contexts (one per SA index).
- A parallel range of DEK objects carrying the AES-GCM key material.
- An embedded IPsec ASO state region and a pair of C-registers used to return per-packet status and sequence number.

Constructor:

```
NvIPsec(
       bit<32>             size,
       NvIPsecCipherType   algo,
       NvIPsecIcv          icv_length = NvIPsecIcv.LENGTH_128_BITS,
       bit<64>             implicit_iv = 0,
       bit<32>             salt        = 0
   );
```

Parameters:

- **size**: Number of IPsec SA indices managed by this NvIPsecinstance. A different index should be used for each direction of a given flow. Must be a power of 2, and no greater than 16M.

- **algo** : Cipher algorithm used by all SAs in this NvIPsec instance. Currently AES_GCM_128 and AES_GCM_256 are supported.
- **icv_length** : Default ICV length (64/96/128 bits) applied to all SAs at creation; can later be overridden per SA by the control plane.
- **implicit_iv** : Default implicit IV seed (used together with the per-packet sequence number to derive the AES-GCM nonce).
- **salt** : Default 32-bit salt value derived from IKE KEYMAT; combined with implicit IV and sequence number to form the ESP nonce.

A newly created **NvIPsec** instance is operational but minimal: all optional features such as hardware replay window, ESN, and lifetimes are disabled until explicitly configured via P4Runtime (**NvIPsecEntry**).

Methods:

8.7.1.1. update_state

Updates or queries the stateful IPsec context for the given SA index.

- On RX (decrypt path): advances the replay window, updates ESN high/low bits, and may signal HARD_LIFETIME when thresholds are reached.
- On TX (encrypt path): reads the current ESP sequence number, increments and writes it back, and returns the updated value.

```
typedef bit<64> nv_ipsec_state_t;

nv_ipsec_state_t update_state<HEADERS>(
    inout HEADERS headers,
    bit<32>       sa_index
);
```

Parameters:

- **headers** [in/out]: The packet headers that are updated as a result of IPsec state.
- **sa_index** : The index to a specific IPsec security association. The index points to a IPsec encryption key obtained during key exchange, and is stored at a given slot in the bucket of keys in NIC memory. These slots can be overwritten as needed by the controller.

Returns:

- NvIPsecResult in the 8 bits (nv_ipsec_state_t[39:32])

```
enum nv_ipsec_result_t NvIPsecResult {
    ASO_OK = 8w0,
    BAD_REPLAY = 8w1,
    HARD_LIFETIME = 8w2,
}
```

- - **ASO_OK** - Operation was successful
  - **BAD_REPLAY** - Replay window violation
  - **HARD_LIFETIME** - Hard lifetime exceeded
- Sequence number (for encrypt) or other state (for decrypt) in the remaining bits. (nv_ipsec_state_t[31:0])

8.7.1.2. encrypt

Performs ESP encryption + authentication for a packet that has already been encapsulated using one of the IPsec encap externs and parsed as an ESP packet.

- Must be invoked only on FDB-Tx flows (host → network).
- Uses the configured DEK, salt, implicit IV, ICV length, and mode (e.g., INCREMENT_SN) for the given SA.
- Writes ESP ICV and any required padding/trailer bytes; the ESP header/trailer layout must have been prepared by the corresponding encap extern and IPsec trailer action.

See the extern functions **nv_encap_esp_transport**, **nv_encap_esp_v4_tunnel**, and **nv_encap_esp_v6_tunnel**.

```
nv_ipsec_state_t encrypt<HEADERS>(
    inout HEADERS headers,
    bit<32>       sa_index
);
```

Parameters:

- **headers[in/out]**: packet headers struct
- **sa_index**: The index to a specific IPsec security association. The index points to a IPsec encryption key obtained during key exchange, and is stored at a given slot in the bucket of keys in NIC memory. These slots can be overwritten as needed by the controller.

Returns:

The returned **nv_ipsec_state_t** provides the updated sequence number and ASO result (e.g., **ASO_OK** or **HARD_LIFETIME**).

8.7.1.3. decrypt

Performs ESP decryption + authentication for an inbound ESP packet associated with the given SA index.

- Must be invoked only on FDB-Rx flows (network → host).
- Consumes ESP header, trailer, and ICV according to the configured ICV length and crypto parameters.
- Sets the packet-level **ipsec_syndrome** and ASO return status, which can be matched in subsequent tables before decapsulation.

After a successful decrypt (no error syndromes), the program typically:

1. Matches on **std_meta.ipsec_syndrome == IPSEC_OK** and ASO result **ASO_OK**.
2. Optionally calls **update_state()** to advance replay/lifetime state.
3. Invokes the appropriate decap extern (**nv_decap_esp_transport**, **nv_decap_udp_esp_transport**, or **nv_decap_esp_tunnel**) to remove ESP (and any UDP tunnel header) and restore the original L3/L4 headers.

```
nv_ipsec_state_t decrypt<HEADERS>(
    inout HEADERS headers,
    bit<32>       sa_index
);
```

Parameters:

- **headers[in/out]** : packet headers struct
- **sa_index** : The index to a specific IPsec security association. The index points to a IPsec encryption key obtained during key exchange, and is stored at a given slot in the bucket of keys in NIC memory. These slots can be overwritten as needed by the controller.

Returns:

The returned **nv_ipsec_state_t** provides the updated sequence number and ASO result (e.g., **ASO_OK** or **HARD_LIFETIME**).

8.7.1.4. Syndrome

The IPsec syndrome is set after a decrypt operation is completed. The possible values are in the following enum:

```
typedef bit<8> nv_ipsec_status_t;
enum nv_ipsec_status_t NvIPsecStatus {
    IPSEC_OK = 8w0,
    ICV_FAIL = 8w1,
    BAD_TRAILER = 8w2,
}
```

It is recommended that the DPL user checks the **NvIPsecStatus** after decrypt by matching on the metadata field **nv_standard_metadata_t.ipsec_syndrome** . If the status is not **IPSEC_OK** , then the application can drop the packet or send it to the controller for further error handling.

8.7.2. Limitations on IPsec Functionality

The following restrictions apply to IPsec capabilities on BlueField-3 / ConnectX-9 in the DPL Target Architecture.

8.7.2.1. Version & Encapsulation

- ESP over IP is fully supported for both tunnel and transport modes.
- UDP+ESP encapsulation is currently not implemented in DPL (but supported in HW).
- AH is considered legacy and is not implemented.
- AES-GCM-128 and AES-GCM-256 are supported for encryption/decryption.
- Other cipher suites (e.g., AES-CCM, ChaCha20-Poly1305) are defined at the control-plane level but may not be enabled in the initial DPL release.

8.7.2.2. Hardware Parsing Requirements

IPsec ESP is a native header; the packet must be parsed as ESP (or UDP-encapsulated ESP) before any NV_IPSEC crypto operation.

- For Encryption: The cleartext packet must be parsed through L3/L4. The appropriate IPsec encap extern must be called first to insert ESP, after which **encrypt()** can be called.
- For Decryption: The packet must arrive with a valid ESP header and trailer (crypto-engine requirements) and be steered to the correct SA by matching {outer IP tuple, protocol = 50, SPI}. The decryption step must complete successfully (no IPsec or ASO syndrome errors) before any decap extern is invoked.

8.7.2.3. State Modes, Replay, and Lifetime

- IPsec supports hardware-assisted Extended Sequence Numbers (ESN), replay windows, and soft/hard lifetimes, but these are per-SA options that must be configured via P4Runtime (NvIPsecEntry); by default they are disabled.
- Modes:
  - **MODE_NONE** : no additional hardware state beyond basic ESP offload.
  - **MODE_REPLAY_PROTECTION** (RX): hardware maintains a replay window bitmap and high/low bounds.
  - **MODE_INCREMENT_SN** (TX): hardware maintains and increments the ESP sequence number.
- Anti-replay is conceptually shared between hardware and software:
  - Hardware can enforce a window (drop or mark **BAD_REPLAY**), but sophisticated policies or rekey logic are expected to be managed by the control plane using ASO state and syndromes.
- Lifetimes:
  - Per-SA soft and hard lifetimes are available; expiry can trigger controller-driven rekey and/or flow removal.

8.7.2.4. Functional & Topology Constraints

- eSwitch only: IPsec full-offload is supported only in the eSwitch (FDB) domain; VF/SF support is firmware-dependent and may require disabling VF IPsec when full offload on the admin eSwitch is enabled.
- Direction-specific crypto:
  - ESP encrypt operations are supported only in the FDB-Tx sub-domain.
  - ESP decrypt operations are supported only in the FDB-Rx sub-domain.
    This applies to NvIPsec.encrypt/decrypt just as it does to PSP; see *Execution Domains and Directional Constraints* for details.
- Single offload per packet:
  - An IPsec full-offload operation (encrypt or decrypt) can be applied at most once per packet.
- Fragmentation and exceptions:
  - IP-fragmented packets are not supported for full IPsec offload; such flows must be handled in software and should be detected and steered by the DPL program (e.g., via fragment offset and MTU range checks).

8.7.2.5. Pipeline Configuration Workflow

To successfully implement IPsec in a DPL-based solution, the application and controller must follow the sequence below.

1. System bring-up and blob load
   - Compile the DPL program with:
     - An **NvIPsec** extern instance (size, cipher, ICV length, return register selection).
     - ESP transport and tunnel encap/decap extern usage.
     - IPsec trailer actions and any required IPsec-specific schema resources.
   - Load the DPL blob into the DPL Runtime Service.
   - The DPL agent:
     - Allocates IPSEC_OFFLOAD contexts and DEK resources for the configured NvIPsec instance(s).

        - Pre-allocates the C-register pairs used for **nv_ipsec_state_t**.
        - Creates packet-reformat contexts and trailer actions for ESP transport/tunnel encap/decap.
2. Policy and IKE key exchange
    - Define IPsec policies (SPD) via Linux xfrm or a custom policy database:
        - Selectors (src/dst IP, protocol, ports), tunnel vs transport, direction, QoS/DSCP, etc.
    - Run an IKE daemon (e.g., StrongSwan, Openswan) to:
        - Establish IKE SAs (Phase 1).
        - Negotiate IPsec child SAs (Phase 2) including SPI, cipher, salt, IV strategy, lifetimes, replay window, ESN flags.
    - For each SA and direction, pass the resulting SADB entry (keys + parameters) to the DPL Runtime Service.
3. SA provisioning via P4Runtime
    - For each SA index:
        - Program the associated DEK via an NvDEKConfig (key type = IPSEC, key size, key bytes).
        - Program the IPSEC_OFFLOAD and IPsec ASO parameters via NvIPsecEntry:
            - Crypto params: **icv_length**, **implicit_iv**, **salt**.
            - Mode: **MODE_NONE**, **MODE_REPLAY_PROTECTION**, or **MODE_INCREMENT_SN**.
            - ESN: **extended_seq_num_enable**, **overlap**, **extended_seq_num_msb**, **esn_event_arm**.
            - Replay window and bitmap.
            - Lifetime and flow-removal thresholds (soft/hard lifetime, packet count/time).
4. Per-flow rule installation
    - For each negotiated SA (per direction), translate SPD policy into DPL steering rules:
        - Decrypt path (RX / full offload):
            - In FDB-Rx: match on {outer IP tuple, protocol = 50, ESP.SPI}.
            - Action: run **NvIPsec.decrypt()**, verify IPsec / ASO syndromes, then call the appropriate ESP decap extern and forward the cleartext packet to the target vport or pipe.
        - Encrypt path (TX / full offload):
            - In FDB-Tx: match on inner cleartext selectors (src/dst IP, protocol, ports).
            - Action: call the proper ESP encap extern (transport or tunnel, IP or UDP), then **NvIPsec.encrypt()**, and finally forward toward the uplink.
5. Exception and software dataplane handling
    - Configure additional rules to detect unsupported or exceptional packets (fragments, MTU violations, crypto syndromes other than **IPSEC_OK** or **ASO_OK**) and steer them to:
        - A VF/SF or host queue for a software IPsec dataplane (e.g., DPDK application).
        - A packet-in path to the controller for diagnostics.

8.7.2.6. Execution Domains and Directional Constraints

The DPL pipeline executes within the administrative eSwitch, also known as the Forwarding Database (FDB) domain. Within this domain, the hardware strictly distinguishes between two directional sub-domains:

- FDB-Rx (Receive): Traffic moving from the network to the host.
- FDB-Tx (Transmit): Traffic moving from the host to the network.

These sub-domains reflect hardware-specific differences in processing capabilities. Consequently, specific cryptographic functions are direction-dependent:

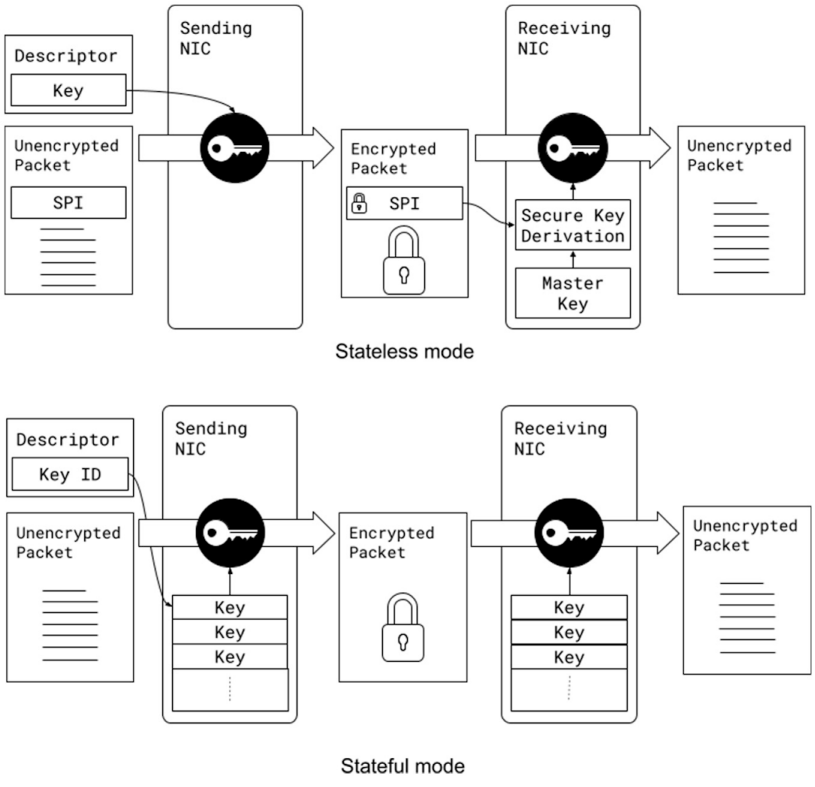
Function	Supported Path	Direction
Encryption	FDB-Tx Only	Host → Network
Decryption	FDB-Rx Only	Network → Host

8.11.7.1.8.8 8.8. PSP

PSP is a transport-independent encryption protocol designed for efficient, large-scale data center and inter-data center network traffic security.

On the Bluefield target, PSP is fully offloaded, meaning cryptographic operations are performed entirely in hardware. This allows for a massive scale of parallel encrypted connections without requiring software state tracking. PSP uses UDP encapsulation with custom headers and standard cryptographic algorithms (AES-GCM) for confidentiality and integrity.

Unlike traditional security protocols (e.g., IPsec or TLS), PSP enables per-connection encryption, supports stateless operation for hardware scalability, and does not mandate a specific key exchange mechanism.



⚠ PSP Key Exchange is out of scope for DPL (Data Plane Logic) and must occur prior to pipeline processing. The host application is responsible for:

1. Performing the asynchronous key exchange (e.g., using **doca_flow_crypto_psp_spi_key_bulk_generate**).
2. Supplying the DPL runtime tables with the appropriate keys via P4Runtime.
3. Ensuring encapsulation entries use the SPI values supplied by the key generation step.

8.8.1. NvPSP

The **NvPSP** object is initialized by selecting a size, representing the number of unique SA indices. The method can then be called in either apply block scope or P4 action scope.

Constructor:

```
NvPSP(bit<32> size)
```

Parameters:

- **size**: The number of indices to allocate for PSP crypto object. A different index should be used for each direction of a given flow. The size must be a power of 2, and no greater than 16M.

Methods:

encrypt

Encrypts a clear text PSP packet. See the extern functions **nv_encap_psp_transport**, **nv_encap_psp_v4_tunnel**, and **nv_encap_psp_v6_tunnel**.

```
void encrypt<HEADERS>(inout HEADERS headers, bit<32> sa_index)
```

Parameters:

- **headers[in/out]**: packet headers struct
- **sa_index**: The index to a specific PSP security association data. The index points to a PSP encryption key obtained during key exchange, and is stored at a given slot in the bucket of keys in NIC memory. These slots can be overwritten as needed by the controller.

decrypt

Decrypts an encrypted PSP packet. After decryption, the PSP headers can be removed using the extern functions **nv_decap_psp_tunnel** or **nv_decap_psp_transport**.

```
void decrypt<HEADERS>(inout HEADERS headers)
```

The decrypt function operates in a stateless mode, where the SPI from the packet is used in conjunction with a master key store on the receiving NIC in order to generate the decryption key.

Syndrome

The PSP syndrome is set after a decrypt operation is completed. The possible values are in the following enum:

```
enum nv_psp_status_t NvPSPStatus {
    PSP_OK = 8w0,
    ICV_FAIL = 8w1,
    BAD_TRAILER = 8w2,
    BAD_CONTEXT = 8w3,
    GENERAL_ERROR = 8w4,
}
```

It is recommended that the DPL user checks the **NvPSPStatus** after decrypt by matching on the metadata field **nv_standard_metadata_t.psp_syndrome**. If the status is not **PSP_OK**, then the application can drop the packet or send it to the controller for further error handling.

8.8.2. Limitations on PSP Functionality

The following restrictions apply to PSP capabilities on the BlueField-3 architecture:

8.8.2.1. Version & Encapsulation

- Only PSP Version 0 and Version 1 are supported.
- UDP and UDP over Tunnel are the only supported transport encapsulations.
- The packet must contain a valid L4 header; otherwise, the encapsulation behavior is undefined.

8.8.2.2. Hardware Parsing Requirements

The packet must be properly parsed by the native hardware parser (**hdrs.psp.isValid() == true**) to ensure operation success:

- For Encryption: The header must be valid before the encryption action is attempted.
- For Decryption: The header must be valid after decryption (and free of error syndromes) before decapsulation can occur.

8.8.2.3. State Modes

- PSP Sender: Supports stateful mode.
- PSP Receiver: Supports stateless mode.

8.8.2.4. Pipeline Configuration Workflow

To successfully implement PSP, the user application must follow this specific configuration order:

1. Load Program: The DPL program is loaded using the P4Runtime API. This step allocates the necessary PSP hardware resources.
2. Key Exchange: The application performs key exchange (see DOCA API References for **doca_flow_crypto_psp_spi_***).
3. Update Receiver: The receiver application updates the PSP resource with the PSP keys using the P4Runtime API.
4. Update Sender: The sender application updates the hardware with the generated SPIs using the P4Runtime API.

8.8.2.5. Execution Domains and Directional Constraints

The DPL pipeline executes within the administrative eSwitch, also known as the Forwarding Database (FDB) domain. Within this domain, the hardware strictly distinguishes between two directional sub-domains:

- FDB-Rx (Receive): Traffic moving from the network to the host.
- FDB-Tx (Transmit): Traffic moving from the host to the network.

These sub-domains reflect hardware-specific differences in processing capabilities. Consequently, specific cryptographic functions are direction-dependent:

Function	Supported Path	Direction
Encryption	FDB-Tx Only	Host --> Network
Decryption	FDB-Rx Only	Network --> Host

8.11.7.1.8.9 8.9. Memory Context

The Memory Context feature provides a stateful, per-index memory resource that can be accessed atomically from the DPL data plane. It is available on ConnectX-9 (and future targets) and is intended for use cases such as:

- Per-flow or per-session state that does not fit in regular metadata
- Stateful counters or bitmaps that must survive across packets
- Saving and restoring header fields across encapsulation/decapsulation

The ConnectX-9 hardware implements the Memory Context as URISC memory objects with 64-byte cache lines, addressable in 8-byte (64-bit) lines per index. Each DPL access is translated into a URISC instruction and executed atomically with respect to packet processing.

Two extern types are exposed:

- **NvMemory** – indirect, indexed by an explicit **index** parameter
- **NvDirectMemory** – direct, bound 1:1 to a P4 table entry

These externs are only available on **CX9** and newer targets; they are gated in the target architecture headers and will fail compilation on BlueField-3.

8.9.1.1. NvMemory Extern

The **NvMemory** extern models a bank of indexed memory contexts. Each index corresponds to one 64-byte cache line, accessed as 64-bit values. The **NvMemory** object is initialized by selecting a size, representing the number of 64-bit values. The methods can then be called in either apply block scope or P4 action scope.

Constructor:

```
NvMemory(bit<32> size)
```

Parameters:

- **size** : The number of indices to allocate for Memory Context object. The size must be a power of 2, and no greater than 16M.

Methods:

read

Returns the 64 bit value stored at the given index.

```
bit<64> read(bit<32> index)
```

Parameters:

- **index** : The index to a specific Memory Context entry.

Returns:

- The 64 bit value stored for the specified index.

write

Stores a 64 bit value at the given index.

```
bit<64> read(bit<32> index, bit<64> value)
```

Parameters:

- **index** : The index to a specific Memory Context entry.
- **value** : The value to be stored

Returns:

- The previous value stored at the specified index.

increment

Increments the value stored at the given index by a specified value with wraparound.

```
bit<64> increment(bit<32> index, bit<64> value)
```

Parameters:

- **index** : The index to a specific Memory Context entry.
- **value** : The value to be added to the stored value

Returns:

- The original value stored at the specified index.

xor

Bitwise XOR with the value stored at the given index.

```
bit<64> xor(bit<32> index, bit<64> value)
```

Parameters:

- **index** : The index to a specific Memory Context entry.
- **value** : The value to XOR with the stored value

Returns:

- The original value stored at the specified index.

or

Bitwise OR with the value at the given index.

```
bit<64> or(bit<32> index, bit<64> value)
```

Parameters:

- **index** : The index to a specific Memory Context entry.
- **value** : The value to OR with the stored value

Returns:

- The original value stored at the specified index.

and

Bitwise AND with the value at the given index.

```
bit<64> and(bit<32> index, bit<64> value)
```

Parameters:

- **index** : The index to a specific Memory Context entry.
- **value** : The value to AND with the stored value.

Returns:

- The original value stored at the specified index.

reset

Clears the value at the given index to zero.

```
bit<64> reset(bit<32> index)
```

Parameters:

- **index** : The index to a specific Memory Context entry.

Returns:

- Zero.

find_first_set_offset_msb, find_first_set_offset_lsb

Finds the bit offset of the most/least significant set bit of the 64 bit value at the given index.

```
bit<64> find_first_set_offset_msb(bit<32> index, bool clear = false);
```

```
bit<64> find_first_set_offset_lsb(bit<32> index, bool clear = false);
```

Parameters:

- **index** : The index to a specific Memory Context entry.
- **clear** : Default, clears the most/least significant set bit in the stored value.

Returns:

Two values are stored in the 64 bit return variable:

- The bit offset/index of the most/least significant set bit. For a return variable named **result** , then the index is a 6 bit value that can be accessed as **result[63:58]**
- A flag to indicate if the memory value was zero (and hence no bit offset was found). The flag can be accessed as a 1 bit value **result[0:0]**

It is up to the user to check the flag if necessary before accessing the index. If the flag is set, then the offset value is undefined.

find_first_set_bit_msb, find_first_set_bit_lsb

Finds the most/least significant set bit of the 64 bit value at the given index.

```
bit<64> find_first_set_bit_msb(bit<32> index, bool clear = false);
bit<64> find_first_set_bit_lsb(bit<32> index, bool clear = false);
```

Parameters:

- **index** : The index to a specific Memory Context entry.
- **clear** : Default, clears the most/least significant set bit in the stored value.

Returns:

- The value with only the most/least set bit retained.

8.9.1.2. NvDirectMemory Extern

The **NvDirectMemory** extern models a bank of memory contexts that is associated with each entry of a P4 table. Each entry contains a 64-bit memory context that persists across table lookups. The **NvDirectMemory** object is initialized by the associated P4 table so the methods can only be called in the P4 action scope.

Constructor:

```
NvDirectMemory()
```

Restrictions: The size is set according to the size of the table, and must be a power of 2, and no greater than 16M.

Usage:

The **NvDirectMemory** object should be declared in the control. Then it is attached to a table using a new table attribute **nv_direct_memory** . For example:

```
NvDirectMemory() my_memory;

table session_table {
    key = {
        user_meta.session_id : exact;
    }
    actions = {
        session_hit;
        session_miss;
    }
    default_action = session_miss();
    direct_counter   = direct_session_counter;
    nv_direct_memory = my_memory;
    size             = P4_SESSION_TABLE_SIZE;
}
```

For each entry in **session_table**, the bound **NvDirectMemory** object references a per-entry memory slot (index == entry index). Actions may call **read()** / **write()** / **increment()** etc. to manipulate that per-entry state atomically.

Methods:

read

Returns the 64 bit value stored for a given entry.

```
bit<64> read()
```

Returns:

- The 64 bit value stored for the specified entry

write

Stores a 64 bit value for a given entry.

```
bit<64> read(bit<64> value)
```

Parameters:

- **value**: The value to be stored

Returns:

- The previous value stored at the specified entry

increment

Increments the value stored for a given entry by the specified value, with wraparound.

```
bit<64> increment(bit<64> value)
```

Parameters:

- **value**: The value to be added to the stored value

Returns:

- The original value stored for a given entry

xor

Bitwise XOR with the value stored for a given entry.

```
bit<64> xor(bit<64> value)
```

Parameters:

- **value** : The value to XOR with the stored value

Returns:

- The original value stored at the specified entry

or

Bitwise OR with the value for a given entry.

```
bit<64> or(bit<64> value)
```

Parameters:

- **value** : The value to OR with the stored value

Returns:

- The original value stored at the specified entry

and

Bitwise AND with the value for a given entry.

```
bit<64> and(bit<64> value)
```

Parameters:

- **value** : The value to AND with the stored value.

Returns:

- The original value stored at the specified entry

reset

Clears the value for a given entry to zero.

```
bit<64> reset()
```

Returns:

- Zero.

find_first_set_offset_msb, find_first_set_offset_lsb

Finds the bit offset of the most/least significant set bit of the 64 bit value for a given entry.

```
bit<64> find_first_set_offset_msb(bool clear = false);
bit<64> find_first_set_offset_lsb(bool clear = false);
```

Parameters:

- **clear** : Default, clears the most/least significant set bit in the stored value.

Returns:

Two values are stored in the 64 bit return variable:

- The bit offset/index of the most/least significant set bit. For a return variable named **result** , then the index is a 6 bit value that can be accessed as **result[63:58]**
- A flag to indicate if the memory value was zero (and hence no bit offset was found). The flag can be accessed as a 1 bit value **result[0:0]**

> ⚠ It is up to the user to check the flag if necessary before accessing the index. If the flag is set, then the offset value is undefined.

find_first_set_bit_msb, find_first_set_bit_lsb

Finds the most/least significant set bit of the 64 bit value for a given entry.

```
bit<64> find_first_set_bit_msb(bool clear = false);
bit<64> find_first_set_bit_lsb(bool clear = false);
```

Parameters:

- **clear** : Default, clears the most/least significant set bit in the stored value.

Returns:

- The value with only the most/least set bit retained.

8.11.7.1.9 9. DPL Expressions

Both the DPL Parser and the DPL standard metadata are part of the DOCA Target Architecture. The hardware has built in header protocols and key field extraction, along with capabilities to set, copy and perform add/subtract on certain fields. The following tables describe the compiler support for expressions using the built in header fields and standard metadata as L-values and R-values. Note: this is separate of header fields that can be used as match keys.

> ⚠ Alias of header fields
>
> In the default hardware parser, some fields that are mutually exclusive are extracted to the same buffer location (referred to in the table as an alias). Assignments to and copy from these fields can use either of the aliased field names.

8.11.7.1.9.1 9.1. Header Field Support

Fixed Header Fields	Assign	Copy	Add/Sub	Notes
headers.ethernet.dst_addr	✔	✔	✘	
headers.ethernet.src_addr	✔	✔	✘	
headers.ethernet.ether_type	✘	✘	✘	Last extracted outer **etherType** value

Fixed Header Fields	Assign	Copy	Add/Sub	Notes
`headers.vlan.vlan_pcp`	✘	✘	✘	
`headers.vlan.vlan_dei`	✘	✘	✘	
`headers.vlan.vlan_id`	✔	✔	✘	
`headers.vlan.vlan_ether_type`	✘	✘	✘	Last extracted outer `etherType` value
`headers.inner_ethernet.dst_addr`	✘	✔	✘	
`headers.inner_ethernet.src_addr`	✘	✔	✘	
`headers.inner_ethernet.ether_type`	✘	✘	✘	Last extracted inner `etherType` value
`headers.inner_vlan.vlan_pcp`	✘	✘	✘	
`headers.inner_vlan.vlan_dei`	✘	✘	✘	
`headers.inner_vlan.vlan_id`	✘	✘	✘	
`headers.inner_vlan.vlan_ether_type`	✘	✘	✘	Last extracted inner `etherType` value
`headers.ipv4.version` Alias with headers.ipv6.version	✘	✘	✘	
`headers.ipv4.ihl`	✔	✔	✔	
`headers.ipv4.diffserv` Alias with `headers.ipv6.diffserv`	✘	✔	✔	Can be set using `nv_set_ip_dscp` extern
`headers.ipv4.ecn` Alias with `headers.ipv6.ecn`	✘	✔	✘	Can be set using `nv_set_ip_ecn` extern
`headers.ipv4.total_len`	✔	✔	✔	May be updated also by hardware
`headers.ipv4.identification`	✘	✘	✘	
`headers.ipv4.flags`	✘	✘	✘	
`headers.ipv4.frag_offset`	✘	✘	✘	
`headers.ipv4.ttl`	✔	✔	✔	
`headers.ipv4.protocol` Alias with `headers.ipv6.next_header`	✘	✔	✘	Can be set using `nv_set_ip_protocol` extern
`headers.ipv4.hdr_checksum`	✘	✘	✘	Value is write only by hardware
`headers.ipv4.src_addr`	✔	✔	✘	
`headers.ipv4.dst_addr`	✔	✔	✘	
`headers.inner_ipv4.version` Alias with `headers.inner_ipv6.version`	✘	✘	✘	
`headers.inner_ipv4.ihl`	✘	✘	✘	

Fixed Header Fields	Assign	Copy	Add/Sub	Notes
`headers.inner_ipv4.diffserv` Alias with `headers.inner_ipv6.diffserv`	✘	✔	✘	
`headers.inner_ipv4.ecn` Alias with `headers.inner_ipv6.ecn`	✘	✘	✘	
`headers.inner_ipv4.total_len`	✘	✘	✘	
`headers.inner_ipv4.identification`	✘	✘	✘	
`headers.inner_ipv4.flags`	✘	✘	✘	
`headers.inner_ipv4.frag_offset`	✘	✘	✘	
`headers.inner_ipv4.ttl`	✘	✔	✘	
`headers.inner_ipv4.protocol` Alias with `headers.inner_ipv6.protocol`	✘	✘	✘	
`headers.inner_ipv4.hdr_checksum`	✘	✘	✘	
`headers.inner_ipv4.src_addr`	✘	✔	✘	
`headers.inner_ipv4.dst_addr`	✘	✔	✘	
`headers.ipv6.flow_label`	✔	✔	✔	
`headers.ipv6.hop_limit`	✔	✔	✔	
`headers.ipv6.payload_length`	✔	✔	✔	
`headers.ipv6.src_addr`	✔	✔	✘	
`headers.ipv6.dst_addr`	✔	✔	✘	
`headers.inner_ipv6.flow_label`	✘	✘	✘	
`headers.inner_ipv6.hop_limit`	✘	✔	✘	
`headers.inner_ipv6.payload_length`	✘	✘	✘	
`headers.inner_ipv6.src_addr`	✘	✔	✘	
`headers.inner_ipv6.dst_addr`	✘	✔	✘	
`headers.mpls.label`	✔	✔	✘	
`headers.mpls.tc`	✔	✔	✘	
`headers.mpls.bos`	✔	✔	✘	
`headers.mpls.ttl`	✔	✔	✘	
`headers.inner_mpls.label`	✘	✔	✘	
`headers.inner_mpls.tc`	✘	✔	✘	
`headers.inner_mpls.bos`	✘	✔	✘	

Fixed Header Fields	Assign	Copy	Add/Sub	Notes
`headers.inner_mpls.ttl`	✘	✔	✘	
`headers.icmp.type`	✘	✘	✘	
`headers.icmp.code`	✘	✘	✘	
`headers.icmp.checksum`	✘	✘	✘	
`headers.icmp.identifier`	✘	✘	✘	
`headers.icmp.sequence_number`	✘	✘	✘	
`headers.inner_icmp.type`	✘	✘	✘	
`headers.inner_icmp.code`	✘	✘	✘	
`headers.inner_icmp.checksum`	✘	✘	✘	
`headers.inner_icmp.identifier`	✘	✘	✘	
`headers.inner_icmp.sequence_number`	✘	✘	✘	
`headers.icmpv6.type`	✘	✘	✘	
`headers.icmpv6.code`	✘	✘	✘	
`headers.icmpv6.checksum`	✘	✘	✘	
`headers.icmpv6.payload_1`	✘	✘	✘	
`headers.icmpv6.payload_2`	✘	✘	✘	
`headers.inner_icmpv6.type`	✘	✘	✘	
`headers.inner_icmpv6.code`	✘	✘	✘	
`headers.inner_icmpv6.checksum`	✘	✘	✘	
`headers.inner_icmpv6.payload_1`	✘	✘	✘	
`headers.inner_icmpv6.payload_2`	✘	✘	✘	
`headers.tcp.src_port` Alias with `headers.udp.src_port`	✘	✔	✘	Can be set using `nv_set_l4_src_port` extern
`headers.tcp.dst_port` Alias with `headers.udp.dst_port`	✘	✔	✘	Can be set using `nv_set_l4_dst_port` extern
`headers.tcp.seq_no`	✔	✔	✔	
`headers.tcp.ack_no`	✔	✔	✔	
`headers.tcp.data_offset`	✔	✔	✔	
`headers.tcp.res`	✘	✘	✘	
`headers.tcp.nonce_sum`	✘	✘	✘	
`headers.tcp.ecn`	✔	✔	✘	

Fixed Header Fields	Assign	Copy	Add/Sub	Notes
headers.tcp.flags	✔	✔	✘	
headers.tcp.window	✘	✘	✘	
headers.tcp.checksum	✘	✘	✘	
headers.tcp.urgent_ptr	✘	✘	✘	
headers.inner_tcp.src_port Alias with headers.inner_udp.src_port	✘	✔	✘	
headers.inner_tcp.dst_port Alias with headers.inner_udp.dst_port	✘	✔	✘	
headers.inner_tcp.seq_no	✘	✘	✘	
headers.inner_tcp.ack_no	✘	✘	✘	
headers.inner_tcp.data_offset	✘	✘	✘	
headers.inner_tcp.res	✘	✘	✘	
headers.inner_tcp.nonce_sum	✘	✘	✘	
headers.inner_tcp.ecn	✘	✔	✘	
headers.inner_tcp.flags	✘	✔	✘	
headers.inner_tcp.window	✘	✘	✘	
headers.inner_tcp.checksum	✘	✘	✘	
headers.inner_tcp.urgent_ptr	✘	✘	✘	
headers.udp.length	✘	✘	✘	
headers.udp.checksum	✘	✘	✘	
headers.inner_udp.length	✘	✘	✘	
headers.inner_udp.checksum	✘	✘	✘	
headers.gre.checksum_present	✘	✘	✘	
headers.gre.reserved1	✘	✘	✘	
headers.gre.key_present	✘	✘	✘	
headers.gre.sequence_present	✘	✘	✘	
headers.gre.reserved2	✘	✘	✘	
headers.gre.reserved3	✘	✘	✘	
headers.gre.version	✘	✘	✘	
headers.gre.protocol	✘	✘	✘	
headers.nvgre_vsid.vsid	✘	✘	✘	

Fixed Header Fields	Assign	Copy	Add/Sub	Notes
headers.nvgre_vsid.flow_id	✘	✘	✘	
headers.esp.security_parameters_index	✔	✔	✘	
headers.esp.sequence_number	✔	✔	✘	
headers.esp.next_header	✔	✔	✘	Value set by hardware after decryption
headers.psp.next_header	✔	✔	✘	Value set by hardware after decryption
headers.psp.hdr_ext_len	✔	✔	✘	
headers.psp.crypt_offset	✔	✔	✘	
headers.psp.needs_sampling	✔	✔	✘	
headers.psp.drop	✔	✔	✘	
headers.psp.version	✔	✔	✘	
headers.psp.has_virtualization_key	✔	✔	✘	
headers.psp.one_1	✔	✔	✘	
headers.psp.security_parameters_index	✔	✔	✘	
headers.psp.initialization_vector	✔	✔	✘	
headers.psp.virtualization_key_high	✔	✔	✘	
headers.psp.virtualization_key_low	✔	✔	✘	
headers.psp.payload_1	✔	✔	✘	
headers.psp.payload_2	✔	✔	✘	
headers.psp.payload_3	✔	✔	✘	
headers.psp.payload_4	✔	✔	✘	
headers.vxlan.reserved1	✘	✘	✘	
headers.vxlan.vni_valid	✘	✘	✘	
headers.vxlan.reserved2	✘	✘	✘	
headers.vxlan.reserved3	✘	✘	✘	
headers.vxlan.vni	✔	✔	✘	
headers.vxlan.reserved4	✔	✔	✘	
headers.vxlan_gpe.reserved1	✘	✘	✘	
headers.vxlan_gpe.vni_valid	✘	✘	✘	
headers.vxlan_gpe.reserved2	✘	✘	✘	

Fixed Header Fields	Assign	Copy	Add/Sub	Notes
headers.vxlan_gpe.reserved3	✘	✘	✘	
headers.vxlan_gpe.next_proto	✘	✘	✘	
headers.vxlan_gpe.vni	✔	✔	✘	
headers.vxlan_gpe.reserved4	✔	✔	✘	
headers.geneve.ver	✘	✘	✘	
headers.geneve.opt_len	✘	✘	✘	
headers.geneve.o	✘	✘	✘	
headers.geneve.c	✘	✘	✘	
headers.geneve.reserved1	✘	✘	✘	
headers.geneve.protocol_type	✘	✘	✘	
headers.geneve.vni	✔	✔	✘	
headers.geneve.reserved2	✔	✔	✘	

8.11.7.1.9.2 9.2. Standard Metadata Support

All the fields of DPL standard metadata are read only. The following table outlines the current support for using a standard metadata field as an R-value in an expression.

Standard Metadata Fields	Copyable	Notes
ingress_port	✘	
eth_to_fcs_packet_len	✘	
is_l2_ok	✘	
l2_type	✘	
last_l2_ether_type	✔	Last extracted value of etherType within ethernet header or VLAN tags
vlan_type	✘	
is_l3_ok	✘	
l3_type	✘	
is_ip_fragmented	✘	
is_ipv4_checksum_ok	✘	
is_l4_ok	✘	
l4_type	✘	

Standard Metadata Fields	Copyable	Notes
l4_type_ext	✘	
is_l4_checksum_ok	✘	
l4_src_port	✘	Copy instead the **headers.tcp** / **udp.src_port** field
l4_dst_port	✘	Copy instead the **headers.tcp** / **udp.dst_port** field
first_tcp_seq_num_icmp_dw0	✘	First extracted value of TCP sequence number or ICMP first dword
first_tcp_ack_num_icmp_dw1	✘	First extracted value of TCP acknowledge number or ICMP second dword
first_tcp_window_urg_ptr_icmp_dw2	✘	First extracted value of TCP window/urgent pointer or ICMP third dword
encap_type	✘	ROCE not currently supported
ipsec_layer	✘	
ipsec_syndrome	✘	Valid only after hardware crypto decrypt
psp_syndrome	✘	Valid only after hardware crypto decrypt
is_inner_l2_ok	✘	
inner_l2_type	✘	
inner_last_l2_ether_type	✔	Last extracted value of etherType within inner ethernet header or VLAN tags
inner_vlan_type	✘	
is_inner_l3_ok	✘	
inner_l3_type	✘	
is_inner_ipv4_checksum_ok	✘	
is_inner_l4_ok	✘	
inner_l4_type	✘	
inner_l4_type_ext	✘	
source_qp	✘	
random_value	✔	*Deprecated*. Use the extern method **nv_get_random_value()** .
ut_clock	✔	*Deprecated*. Use the extern method **nv_get_ut_clock()** .
fr_clock	✔	*Deprecated*. Use the extern method **nv_get_fr_clock()** .

8.11.7.1.10 10. Tables

8.11.7.1.10.1 10.1. Match Kinds

The DPL compiler supports following match types with the restrictions below:

- Exact match
  - Support for at least 9 keys of DWORD width and 8 keys of byte width. Note that the actual limit is dependent on HW resources and may fail only at load time.
  - See Exact tables in the PSA spec for general guidance and usage from a programmer perspective.
- LPM match
  - Not supported for bool-type keys.
  - Not supported for standard metadata fields.
  - See LPM tables in the PSA spec for general guidance and usage from a controller perspective.
- Ternary match
  - Not supported for bool-type keys.
  - Not supported for standard metadata fields.
  - See Ternary tables in the PSA spec for general guidance and usage from a programmer perspective.
- Range match
  - Support up to a maximum of 4 range match keys per P4 table .
  - Support for slices of header fields and std_meta fields as keys.
  - Not supported for slices of user-declared variables and user metadata fields as keys.
  - Not supported for keys larger than 32 bits (unless sliced down to 32 bits or less, with some key-specific slice alignment restrictions).
  - Range match keys in the same P4 Table with Ternary match keys or LPM keys are currently not supported.
  - When using multiple range keys in a single P4 table, some combinations of keys may not be supported. Note that this may fail only at load time.
  - See Range tables in the PSA spec for general guidance and usage from a programmer perspective.

8.11.7.1.10.2 10.2. DPL Key Support

IsValid method call, local variables excluding packet-in struct instance, fixed or flex header fields, and standard metadata can be used as a table key.

However, there are following restrictions:

- A key cannot be used multiple times within one P4 table; this will actively fail during compilation. When IsValid method call is used as a table key, the compiler allocates a register for the key. Thus, using multiple isValid method calls as keys within a table causes undefined behavior.
- Complex expressions must use `@name` annotation.

The following table lists the match key support for fixed header fields.

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`headers.ethernet.dst_addr`	48	✔	✔	✔	✔	
`headers.ethernet.src_addr`	48	✔	✔	✔	✔	
`headers.ethernet.ether_type`	16	✘	✘	✘	✘	Last extracted l2 etherType can be matched through `std_meta.last_l2_ether_type`
`headers.vlan.vlan_pcp`	3	✔	✔	✔	✔	
`headers.vlan.vlan_dei`	1	✔	✔	✔	✔	
`headers.vlan.vlan_id`	12	✔	✔	✔	✔	
`headers.vlan.vlan_ether_type`	16	✘	✘	✘	✘	Last extracted l2 etherType can be matched through `std_meta.last_l2_ether_type`
`headers.inner_ethernet.dst_addr`	48	✔	✔	✔	✔	
`headers.inner_ethernet.src_addr`	48	✔	✔	✔	✔	
`headers.inner_ethernet.ether_type`	16	✘	✘	✘	✘	Last extracted inner l2 etherType can be matched through `std_meta.inner_last_l2_ether_type`
`headers.inner_vlan.vlan_pcp`	3	✔	✔	✔	✔	
`headers.inner_vlan.vlan_dei`	1	✔	✔	✔	✔	
`headers.inner_vlan.vlan_id`	12	✔	✔	✔	✔	
`headers.inner_vlan.vlan_ether_type`	16	✘	✘	✘	✘	Last extracted inner l2 etherType can be matched through `std_meta.inner_last_l2_ether_type`
`headers.ipv4.version` Alias with `headers.ipv6.version`	4	✔	✔	✔	✔	
`headers.ipv4.ihl`	4	✔	✔	✔	✔	
`headers.ipv4.diffserv` Alias with `headers.ipv6.diffserv`	6	✔	✔	✔	✔	

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`headers.ipv4.ecn` Alias with `headers.ipv6.ecn`	2	✔	✔	✔	✔	
`headers.ipv4.total_len`	16	✔	✔	✔	✔	
`headers.ipv4.identification`	16	✔	✔	✔	✔	
`headers.ipv4.flags`	3	✔	✔	✔	✔	
`headers.ipv4.frag_offset`	13	✔	✔	✔	✔	
`headers.ipv4.ttl`	8	✔	✔	✔	✔	
`headers.ipv4.protocol` Alias with `headers.ipv6.next_header`	8	✔	✔	✔	✔	
`headers.ipv4.hdr_checksum`	16	✔	✔	✔	✔	
`headers.ipv4.src_addr`	32	✔	✔	✔	✔	
`headers.ipv4.dst_addr`	32	✔	✔	✔	✔	
`headers.inner_ipv4.version` Alias with `headers.inner_ipv6.version`	4	✔	✔	✔	✔	
`headers.inner_ipv4.ihl`	4	✔	✔	✔	✔	
`headers.inner_ipv4.diffserv` Alias with `headers.inner_ipv6.diffserv`	6	✔	✔	✔	✔	
`headers.inner_ipv4.ecn` Alias with `headers.inner_ipv6.ecn`	2	✔	✔	✔	✔	
`headers.inner_ipv4.total_len`	16	✔	✔	✔	✔	
`headers.inner_ipv4.identification`	16	✔	✔	✔	✔	

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`headers.inner_ipv4.flags`	3	✔	✔	✔	✔	
`headers.inner_ipv4.frag_offset`	13	✔	✔	✔	✔	
`headers.inner_ipv4.ttl`	8	✔	✔	✔	✔	
`headers.inner_ipv4.protocol` Alias with `headers.inner_ipv6.next_header`	8	✔	✔	✔	✔	
`headers.inner_ipv4.hdr_checksum`	16	✔	✔	✔	✔	
`headers.inner_ipv4.src_addr`	32	✔	✔	✔	✔	
`headers.inner_ipv4.dst_addr`	32	✔	✔	✔	✔	
`headers.ipv6.flow_label`	20	✔	✔	✔	✔	
`headers.ipv6.payload_length`	16	✔	✔	✔	✔	
`headers.ipv6.hop_limit`	8	✔	✔	✔	✔	
`headers.ipv6.src_addr`	128	✔	✔	✔	✔	
`headers.ipv6.dst_addr`	128	✔	✔	✔	✔	
`headers.inner_ipv6.flow_label`	20	✔	✔	✔	✔	
`headers.inner_ipv6.payload_length`	16	✔	✔	✔	✔	
`headers.inner_ipv6.hop_limit`	8	✔	✔	✔	✔	
`headers.inner_ipv6.src_addr`	128	✔	✔	✔	✔	
`headers.inner_ipv6.dst_addr`	128	✔	✔	✔	✔	

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`headers.mpls.label`	20	✔	✔	✔	✔	
`headers.mpls.tc`	3	✔	✔	✔	✔	
`headers.mpls.bos`	1	✔	✔	✔	✔	
`headers.mpls.ttl`	8	✔	✔	✔	✔	
`headers.inner_mpls.label`	20	✔	✔	✔	✔	
`headers.inner_mpls.tc`	3	✔	✔	✔	✔	
`headers.inner_mpls.bos`	1	✔	✔	✔	✔	
`headers.inner_mpls.ttl`	8	✔	✔	✔	✔	
`headers.icmp.type`	8	✘	✘	✘	✘	First parsed value of ICMP type can be matched through `std_meta.first_tcp_seq_num_icmp_dw0`
`headers.icmp.code`	8	✘	✘	✘	✘	First parsed value of ICMP code can be matched through `std_meta.first_tcp_seq_num_icmp_dw0`
`headers.icmp.checksum`	16	✘	✘	✘	✘	First parsed value of ICMP checksum can be matched through `std_meta.first_tcp_seq_num_icmp_dw0`
`headers.icmp.identifier`	16	✘	✘	✘	✘	First parsed value of ICMP identifier can be matched through `std_meta.first_tcp_seq_num_icmp_dw1`
`headers.icmp.sequence_number`	16	✘	✘	✘	✘	First parsedv alue of ICMP sequence number can be matched through `std_meta.first_tcp_seq_num_icmp_dw1`
`headers.inner_icmp.type`	8	✘	✘	✘	✘	First parsed value of ICMP type can be matched through `std_meta.first_tcp_seq_num_icmp_dw0`
`headers.inner_icmp.code`	8	✘	✘	✘	✘	First parsed value of ICMP code can be matched through `std_meta.first_tcp_seq_num_icmp_dw0`
`headers.inner_icmp.checksum`	16	✘	✘	✘	✘	First parsed value of ICMP checksum can be matched through `std_meta.first_tcp_seq_num_icmp_dw0`
`headers.inner_icmp.identifier`	16	✘	✘	✘	✘	First parsedvalue of ICMP identifier can be matched through `std_meta.first_tcp_seq_num_icmp_dw1`
`headers.inner_icmp.sequence_number`	16	✘	✘	✘	✘	First parsed value of ICMP sequence number can be matched through `std_meta.first_tcp_seq_num_icmp_dw1`
`headers.icmpv6.type`	8	✘	✘	✘	✘	First parsed value of ICMPv6 type can be matched through `std_meta.first_tcp_seq_num_icmp_dw0`

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`headers.icmpv6.code`	8	✘	✘	✘	✘	First parsed value of ICMPv6 code can be matched through `std_meta.first_tcp_seq_num_icmp_dw0`
`headers.icmpv6.checksum`	16	✘	✘	✘	✘	First parsed value of ICMPv6 checksum can be matched through `std_meta.first_tcp_seq_num_icmp_dw0`
`headers.icmpv6.payload_1`	32	✘	✘	✘	✘	First parsed value of ICMPv6 payload_1 can be matched through `std_meta.first_tcp_seq_num_icmp_dw1`
`headers.icmpv6.payload_2`	32	✘	✘	✘	✘	First parsed value of ICMPv6 payload 2can be matched through `std_meta.first_tcp_seq_num_icmp_dw2`
`headers.inner_icmpv6.type`	8	✘	✘	✘	✘	First parsed value of ICMPv6 type can be matched through `std_meta.first_tcp_seq_num_icmp_dw0`
`headers.inner_icmpv6.code`	8	✘	✘	✘	✘	First parsed value of ICMPv6 code can be matched through `std_meta.first_tcp_seq_num_icmp_dw0`
`headers.inner_icmpv6.checksum`	16	✘	✘	✘	✘	First parsed value of ICMPv6 checksum can be matched through `std_meta.first_tcp_seq_num_icmp_dw0`
`headers.inner_icmpv6.payload_1`	32	✘	✘	✘	✘	First parsed value of ICMPv6 payload_1 can be matched through `std_meta.first_tcp_seq_num_icmp_dw1`
`headers.inner_icmpv6.payload_2`	32	✘	✘	✘	✘	First parsed value of ICMPv6 payload 2can be matched through `std_meta.first_tcp_seq_num_icmp_dw2`
`headers.tcp.src_port` Alias with `headers.udp.src_port`	16	✔	✔	✔	✔	
`headers.tcp.dst_port` Alias with `headers.udp.dst_port`	16	✔	✔	✔	✔	
`headers.tcp.seq_no`	32	✘	✘	✘	✘	First parsed value of TCP sequence number can be matched through `std_meta.first_tcp_seq_num_icmp_dw0`
`headers.tcp.ack_no`	32	✘	✘	✘	✘	First parsed value of TCP acknowledge number can be matched through `std_meta.first_tcp_seq_num_icmp_dw1`
`headers.tcp.data_offset`	4	✔	✔	✔	✔	
`headers.tcp.res`	3	✘	✘	✘	✘	

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`headers.tcp.nonce_sum`	1	✔	✔	✔	✔	
`headers.tcp.ecn`	2	✔	✔	✔	✔	
`headers.tcp.flags`	6	✔	✔	✔	✔	
`headers.tcp.window`	16	✘	✘	✘	✘	First parsed value of TCP window can be matched through `std_meta.first_tcp_seq_num_icmp_dw2`
`headers.tcp.checksum`	16	✔	✔	✔	✔	
`headers.tcp.urgent_ptr`	16	✘	✘	✘	✘	First parsed value of TCP urgent pointer can be matched through `std_meta.first_tcp_seq_num_icmp_dw2`
`headers.inner_tcp.src_port` Alias with `headers.inner_udp.src_port`	16	✔	✔	✔	✔	
`headers.inner_tcp.dst_port` Alias with `headers.inner_udp.dst_port`	16	✔	✔	✔	✔	
`headers.inner_tcp.seq_no`	32	✘	✘	✘	✘	First parsed value of TCP sequence number can be matched through `std_meta.first_tcp_seq_num_icmp_dw0`
`headers.inner_tcp.ack_no`	32	✘	✘	✘	✘	First parsed value of TCP acknowledge number can be matched through `std_meta.first_tcp_seq_num_icmp_dw1`
`headers.inner_tcp.data_offset`	4	✔	✔	✔	✔	
`headers.inner_tcp.res`	3	✘	✘	✘	✘	
`headers.inner_tcp.nonce_sum`	1	✔	✔	✔	✔	
`headers.inner_tcp.ecn`	2	✔	✔	✔	✔	
`headers.inner_tcp.flags`	6	✔	✔	✔	✔	
`headers.inner_tcp.window`	16	✘	✘	✘	✘	First parsed value of TCP window can be matched through `std_meta.first_tcp_seq_num_icmp_dw2`

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`headers.inner_tcp.checksum`	16	✔	✔	✔	✔	
`headers.inner_tcp.urgent_ptr`	16	✘	✘	✘	✘	First parsed value of TCP urgent pointer can be matched through `std_meta.first_tcp_seq_num_icmp_dw2`
`headers.udp.length`	16	✔	✔	✔	✔	
`headers.udp.checksum`	16	✔	✔	✔	✔	
`headers.inner_udp.length`	16	✔	✔	✔	✔	
`headers.inner_udp.checksum`	16	✔	✔	✔	✔	
`headers.gre.checksum_present`	1	✔	✔	✔	✔	
`headers.gre.reserved1`	1	✔	✔	✔	✔	
`headers.gre.key_present`	1	✔	✔	✔	✔	
`headers.gre.sequence_present`	1	✔	✔	✔	✔	
`headers.gre.reserved2`	4	✔	✔	✔	✔	
`headers.gre.reserved3`	5	✔	✔	✔	✔	
`headers.gre.version`	3	✔	✔	✔	✔	
`headers.gre.protocol`	16	✔	✔	✔	✔	
`headers.nvgre_vsid.vsid`	24	✔	✔	✔	✔	
`headers.nvgre_vsid.flow_id`	8	✔	✔	✔	✔	
`headers.esp.security_parameters.index`	32	✔	✔	✔	✔	
`headers.esp.sequence_number`	32	✔	✔	✔	✔	
`headers.esp.next_header`	8	✔	✔	✔	✔	

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
headers.psp.next_header	8	✔	✔	✔	✔	
headers.psp.hdr_ext_len	8	✔	✔	✔	✔	
headers.psp.crypt_offset	8	✔	✔	✔	✔	
headers.psp.needs_sampling	1	✔	✔	✔	✔	
headers.psp.drop	1	✔	✔	✔	✔	
headers.psp.version	1	✔	✔	✔	✔	
headers.psp.has_virtualization_key	1	✔	✔	✔	✔	
headers.psp.one_1	1	✔	✔	✔	✔	
headers.psp.security_parameters_index	32	✔	✔	✔	✔	
headers.psp.initialization_vector	64	✔	✔	✔	✔	
headers.psp.virtualization_key_high	32	✔	✔	✔	✔	
headers.psp.virtualization_key_lower	32	✔	✔	✔	✔	
headers.psp.payload_1	32	✔	✔	✔	✔	
headers.psp.payload_2	32	✔	✔	✔	✔	
headers.psp.payload_3	32	✔	✔	✔	✔	
headers.psp.payload_4	32	✔	✔	✔	✔	
headers.vxlan.reserved1	4	✔	✔	✔	✔	
headers.vxlan.vni_valid	1	✔	✔	✔	✔	
headers.vxlan.reserved2	3	✔	✔	✔	✔	
headers.vxlan.reserved3	24	✔	✔	✔	✔	

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`headers.vxlan.vni`	24	✔	✔	✔	✔	
`headers.vxlan.reserved4`	8	✔	✔	✔	✔	
`headers.vxlan_gpe.reserved1`	4	✔	✔	✔	✔	
`headers.vxlan_gpe.vni_valid`	1	✔	✔	✔	✔	
`headers.vxlan_gpe.reserved2`	3	✔	✔	✔	✔	
`headers.vxlan_gpe.reserved3`	16	✔	✔	✔	✔	
`headers.vxlan_gpe.next_proto`	8	✔	✔	✔	✔	
`headers.vxlan_gpe.vni`	24	✔	✔	✔	✔	
`headers.vxlan_gpe.reserved4`	8	✔	✔	✔	✔	
`headers.geneve.ver`	2	✔	✔	✔	✔	
`headers.geneve.opt_len`	6	✔	✔	✔	✔	
`headers.geneve.o`	1	✔	✔	✔	✔	
`headers.geneve.c`	1	✔	✔	✔	✔	
`headers.geneve.reserved1`	6	✔	✔	✔	✔	
`headers.geneve.protocol_type`	16	✔	✔	✔	✔	
`headers.geneve.vni`	24	✔	✔	✔	✔	
`headers.geneve.reserved2`	8	✔	✔	✔	✔	

We don't have a way to export this macro.

The following table lists the support for `isValid()` for fixed headers:

Header IsValid	Notes
ethernet	
inner_ethernet	
vlan	VLAN tag is CVLAN
inner_vlan	Inner VLAN tag is CVLAN
svlan	VLAN tag is SVLAN
inner_svlan	Inner VLAN tag is SVLAN
ipv4	L3 header protocol is IPv4
inner_ipv4	Inner l3 header protocol is IPv4
ipv6	L3 header protocol is IPv6
inner_ipv6	Inner l3 header protocol is IPv6
mpls	MPLS layer 0 is valid after passing all HW checks
inner_mpls	Inner MPLS layer 0 is valid after passing all HW checks
mpls1	MPLS layer 1 is valid after passing all HW checks
inner_mpls1	Inner MPLS layer 1 is valid after passing all HW checks
mpls2	MPLS layer 2 is valid after passing all HW checks
inner_mpls2	Inner MPLS layer 2 is valid after passing all HW checks
mpl3	MPLS layer 3 is valid after passing all HW checks
inner_mpls3	Inner MPLS layer 3 is valid after passing all HW checks
mpls_4	MPLS layer 4 is valid after passing all HW checks
inner_mpls4	Inner MPLS layer 4 is valid after passing all HW checks
tcp	L4 type is TCP
inner_tcp	Inner l4 type is TCP
udp	L4 type is UDP
inner_udp	Inner l4 type is UDP
gre	• L3 layer is valid after passing all HW checks • IPv4 protocol field value is 47
nvgre	• L3 is OK • IPv4 protocol is 47 • GRE key is present • GRE protocol type is 25944
esp	L4 type is IPSEC
icmp	L4 type is ICMP
inner_icmp	Inner l4 type is ICMP

Header IsValid	Notes
icmpv6	L4 type is ICMP
inner_icmpv6	Inner l4 type is ICMP
vxlan	• L4 type is UDP • headers.udp.dst_port is 4789
vxlan_gpe	• L4 type is UDP • headers.udp.dst_port is 4790
geneve	• L4 type is UDP • headers.udp.dst_port is 6081
psp	• L4 type is UDP • headers.udp.dst_port is 1000

The following table lists the match key support for standard metadata fields:

Standard Metadata Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
ingress_port	32	✔	✘	✘	✘	The P4 port ID that the packet ingressed the pipeline
eth_to_fcs_packet_len	16	✔	✘	✘	✔	Length of the packet from L2 start to FCS, in bytes
is_l2_ok	1	✔	✘	✘	✔	L2 layer is valid after passing all HW checks
l2_type	2	✔	✘	✘	✔	• 0: L2_TYPE_UNICAST • 1: L2_TYPE_MULTICAST • 2: L2_TYPE_BROADCAST
last_l2_ether_type	16	✔	✔	✔	✔	Last extracted value of etherType within ethernet header or VLAN tags
vlan_type	2	✔	✘	✘	✔	• 0: VLAN_TYPE_NONE • 1: VLAN_TYPE_SVLAN • 2: VLAN_TYPE_CVLAN
is_l3_ok	1	✔	✘	✘	✔	L3 layer is valid after passing all hardware checks
l3_type	2	✔	✘	✘	✔	• 0: L3_TYPE_NONE • 1: L3_TYPE_IPV4 • 2: L3_TYPE_IPv6
is_ip_fragmented	1	✔	✘	✘	✔	
is_ipv4_checksum_ok	1	✔	✘	✘	✔	IPv4 layer checksum is valid
is_l4_ok	1	✔	✘	✘	✔	L4 layer (TCP/UDP) is valid after passing all hardware checks
l4_type	2	✔	✘	✘	✔	• 0: L4_TYPE_NONE • 1: L4_TYPE_TCP • 2: L4_TYPE_UDP • 3: L4_TYPE_IPSEC

Standard Metadata Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
l4_type_ext	4	✔	✘	✘	✔	• 0: L4_TYPE_EXT_NONE • 1: L4_TYPE_EXT_TCP • 2: L4_TYPE_EXT_UDP • 3: L4_TYPE_EXT_IPSEC
is_l4_checksum_ok	1	✔	✘	✘	✔	L4 layer (TCP/UDP) checksum is valid
l4_src_port	16	✔	✔	✔	✔	
l4_dst_port	16	✔	✔	✔	✔	
first_tcp_seq_num_icmp_dw0	32	✔	✔	✔	✔	First parsed TCP sequence number. Or the last extracted ICMP DWORD 0
first_tcp_ack_num_icmp_dw1	32	✔	✔	✔	✔	First parsed TCP acknowledge number. Or the last extracted ICMP DWORD 1
first_tcp_window_urg_ptr_icmp_dw2	32	✔	✔	✔	✔	First parsed TCP window/urgent pointer DWORD. Or the last extracted ICMP DWORD 2
encap_type	2	✔	✘	✘	✔	• 0: ENCAP_TYPE_NONE • 1: ENCAP_TYPE_L2_TUNNEL • 2: ENCAP_TYPE_L3_TUNNEL • 3: ENCAP_TYPE_ROCE
ipsec_layer	2	✔	✘	✘	✔	• 0: IPSEC_TYPE_NONE • 1: IPSEC_TYPE_OVER_IP • 2: IPSEC_TYPE_OVER_UDP
ipsec_syndrome	8	✔	✘	✘	✔	0: CRYPTO_OK
psp_syndrome	8	✔	✘	✘	✔	0: CRYPTO_OK
is_inner_l2_ok	1	✔	✘	✘	✔	Inner L2 layer is valid after passing all HW checks
inner_l2_type	2	✔	✘	✘	✔	• 0: L2_TYPE_UNICAST • 1: L2_TYPE_MULTICAST • 2: L2_TYPE_BROADCAST
inner_last_l2_ether_type	16	✔	✔	✔	✔	Last extracted value of etherType within inner ethernet header and VLAN tags
inner_vlan_type	2	✔	✔	✔	✔	• 0: VLAN_TYPE_NONE • 1: VLAN_TYPE_SVLAN • 2: VLAN_TYPE_CVLAN
is_inner_l3_ok	1	✔	✘	✘	✔	Inner l3 layer (IPv4/IPv6) is valid after passing all HW checks
inner_l3_type	2	✔	✘	✘	✔	• 0: L3_TYPE_NONE • 1: L3_TYPE_IPV4 • 2: L3_TYPE_IPv6
is_inner_ipv4_checksum_ok	1	✔	✔	✔	✔	

Standard Metadata Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
is_inner_l4_ok	1	✔	✘	✘	✔	Inner L4 layer (TCP/UDP) is valid after passing all HW checks
inner_l4_type	2	✔	✘	✘	✔	• 0: L4_TYPE_NONE • 1: L4_TYPE_TCP • 2: L4_TYPE_UDP • 3: L4_TYPE_IPSEC
inner_l4_type_ext	4	✔	✘	✘	✔	• 0: L4_TYPE_EXT_NONE • 1: L4_TYPE_EXT_TCP • 2: L4_TYPE_EXT_UDP • 3: L4_TYPE_EXT_IPSEC
is_inner_l4_checksum_ok	1	✔	✘	✘	✔	
inner_l4_src_port	16	✔	✔	✔	✔	
inner_l4_dst_port	16	✔	✔	✔	✔	
source_qp	24	✔	✘	✘	✔	
random_value	16	✔	✔	✔	✔	*Deprecated*. Match on the return value from calling the extern method **nv_get_random_value()** .
ut_clock	64	✔	✔	✔	✔	*Deprecated*. Match on the return value from calling the extern method **nv_get_ut_clock()** .
fr_clock	64	✔	✔	✔	✔	*Deprecated*. Match on the return value from calling the extern method **nv_get_fr_clock()** .

⚠ The maximum number of keys in any P4 table is known only at load time, due to the dynamic nature of the key packing algorithm. The key width limit in BlueField-3 is approximately 40 bytes. If more keys are required, the user should split the keys into two or more tables.

8.11.7.1.10.3 10.3. Match Key Annotations

The following annotations for individual P4 table match keys is supported:

- @name
  - The @name annotation lets a P4 programmer override the compiler-chosen string name for a P4 object as it is emitted into P4Info (and therefore what is visible to P4Runtime clients), instead of using the default generated name.
  - Because it can change the emitted names, @name can introduce conflicting (duplicate) names for distinct objects if not used with caution.
- @nv_high_entropy_bits
  - The @nv_high_entropy_bits annotation is used to indicate which bits of the key to prioritize when configuring the table's hash function implementation. This

information is used to influence the hash computed to try to reduce the number of hash collisions, which on average increases performance.
- See DPL Performance section for usage, details and restrictions

8.11.7.1.10.4 10.4. Size

Each P4 table in the DOCA Target Architecture allows a **size** attribute. This represents the maximum number of entries, excluding the default miss entry. If the **size** attribute is omitted, a default value of 128 is assigned.

10.4.1. Sizing Constraints

A single P4 table supports a maximum size of 20M entries on BlueField3, and 32M entries on CX-9. The specific requirements for the **size** parameter depend on the scale of the table:

Table Scale	Size Range	Constraint
Standard	1 to 8,388,608 (8M)	Must be a power of two.
Large (BlueField3)	8,388,609 to 20,971,520 (20M)	Must be specified in 1M (1,048,576) increments.
Large (CX-9)	8,388,609 to 33,554,432 (32M)	Must be specified in 1M (1,048,576) increments.

10.4.2. Large Table Support (>8M Entries)

When a table is configured with a size greater than 8M, the DPL compiler automatically generates a set of sharded tables. This process is transparent to the user but introduces specific limitations compared to standard tables.

10.4.2.1. Architecture & Memory Limits

- Total System Limit: The total memory the DPL runtime can allocate for *all* tables combined is 20M entries on Bluefield3. On CX-9 the limit depends upon the amount of Host memory that is available.
- Hardware Dependency: This limit is constrained by the 32GB DRAM on the BlueField-3 Arm subsystem. The actual limit may be lower if other resource-intensive processes are running on the Arm cores.

10.4.2.2. Functional Limitations

The following features are not supported for Large Tables:

- Match Type: Only Exact Match keys are supported.
- Direct Objects: Direct resources (e.g., **direct_counter**, **direct_meter**) cannot be attached due to resource constraints. Solution: Use the indexed (indirect) versions of these objects instead.
- Idle Timeout: Entry aging/timeout is not supported.
- Debug Actions: The **nv_send_debug_pkt** action cannot be used within these tables.
- On CX9, to support multiple tables with a large size, the **hugepages** may need to be resized to from the default of 4 MB to 16 MB.

The actual number of supported entries in Large Tables is best-effort and subject to hash collisions. For optimal performance and deterministic behavior, it is recommended to keep table sizes to 8M or less whenever possible.

8.11.7.1.10.5 10.5. Default Action

The table can contain default action. When there are no matching entries within this table, it will execute the default action. When the default action is not provided, the missing entry will execute NoAction action (continue to the next logical step in the pipeline).

```
table ipv4_fowarding_table {
    key = {
        headers.ipv4.src_addr : exact;
    }
    actions = {
        forward;
        NoAction;
    }
    default_action = forward(1);
}
```

8.11.7.1.10.6 10.6. Keyless Tables

A table with no keys is supported, by either omitting the **key** property or specifying an empty key set. Such tables are deemed "always miss", and all packets processed by the table will go to the default action. It is not permitted to insert or remove entries on a keyless table. However, if the default action is not marked **const**, then it is permitted to modify the default action at runtime. While it is a matter of coding style, an alternative is to invoke such P4 actions directly in the apply block logic of the appropriate control.

Instead of using keyless tables, using the apply block for generic logic that applies to all packets (i.e. no key lookup) allows the compiler more opportunities to optimize the DPL program.

8.11.7.1.10.7 10.7. Const Entries

Const entries are an optional property of a P4 table, which can be used to insert entries when loading the blob to BlueField. For LPM or ternary matching, the unmasked bits must be zero. As defined in the P4 Language specification, the presence of const entries means the P4 table is not modifiable, and entries cannot be inserted or removed at runtime.

```
table ipv4_fowarding_table {
    key = {
        headers.ipv4.src_addr : lpm;
    }
    actions = {
        forward;
        NoAction;
    }
    default_action = forward(1);
    const entries = {
        (32w0x11111100 &&& 32w0xFFFFFF00) : forward(2);
        (32w0x22220000 &&& 36w0xFFFF0000) : forward(3);
    }
}
```

8.11.7.1.10.8 10.8. Entry Timeout

The DOCA Target Architecture supports the idle timeout and notification mechanisms specified in the P4Runtime specification. This feature is disabled by default and must be explicitly enabled on a per-table basis.

To enable this feature, add the attribute **nv_support_timeout = true** to the table definition.

```
table ipv4_forwarding_table {
    key = {
        headers.ipv4.src_addr : lpm;
    }
    actions = {
        forward;
        NoAction;
    }
    default_action = forward(1);

    /* Required for Timeout Support */
    direct_counter = fwd_counter;
    nv_support_timeout = true;
}
```

10.8.1. Restrictions and Prerequisites

Enabling entry timeout imposes the following constraints on the table:

- Counters Required: The table *must* have the **direct_counter** property set.
- Action Logic: Every action in the table's action list must explicitly call the **count()** method on the associated direct counter.
- Mutability: The table cannot contain **const** entries.
- Size Limit: The table size must be 8M entries or less. (Large Tables >8M do not support timeout).

10.8.2. Timing Precision and Latency

The timeout notification mechanism is subject to the polling frequency of the hardware.

10.8.2.1. Notification Delay

Notifications may be delivered slightly later than the exact timeout expiration. This delay is governed by the **idle_timeout_polling_interval** parameter in the DPL Runtime Service configuration.

The maximum potential delay is calculated as:

We don't have a way to export this macro.

10.8.2.2. Timestamp Accuracy

The P4Runtime field **time_since_last_hit.elapsed_ns** reflects the polling cycle in which the hit was detected, rather than the exact nanosecond timestamp of the packet hit itself.

> ⚠ For very large tables with high churn (frequent additions/deletions), the time required to read all counters becomes a factor. Under heavy load, the system guarantees that the notification will eventually be delivered, but strictly never earlier than the defined timeout duration.

8.11.7.1.10.9 10.9. Delayed Counter Statistics

The DOCA Target Architecture supports delayed counter statistics, which allows reducing the performance impacts of frequent queries by collecting statistics periodically or on demand. This feature is by default off and must be enabled on a per-table basis.

Under the table definition, add the attribute **nv_delayed_counter_stats = true**. For example:

```
table ipv4_fowarding_table {
    key = {
        headers.ipv4.src_addr : lpm;
    }
    actions = {
        forward;
        NoAction;
    }
    default_action = forward(1);
    direct_counter = fwd_counter;
    nv_delayed_counter_stats = true;
 }
```

Restriction when delayed counter statistics is enabled:

- The table must have the **direct_counter** property set

> The counter values are updated at time intervals that are controlled by the **idle_timeout_polling_interval** setting in the configuration of the DPL Runtime Service.

8.11.7.1.10.10 10.10. High Update Rate Tables

P4Runtime provides a gRPC-based API to insert and remove entries from P4 tables. While flexible, platform-independent, and capable of operating remotely within the network, such controllers are inherently limited in terms of performance and scalability. To address this limitation, DPL introduces support for High Update Rate (HUR) tables.

Under the table definition, add the attribute **nv_high_update_rate = true**. For example:

```
    table acl_table {
        key = {
            headers.inner_ipv6.src_addr : exact;
            headers.inner_ipv6.dst_addr : exact;
            headers.inner_ipv6.next_header : exact;
            std_meta.inner_l4_src_port : exact;
            std_meta.inner_l4_dst_port : exact;
        }
        actions = {
            deny;
            allow;
            NoAction;
        }
        default_action = NoAction;
        nv_high_update_rate = true;
        size = 4194304;
    }
```

The following restrictions apply when high update rate (HUR) table is enabled:

- All keys must be of match kind exact
- HUR tables are not compatible with **nv_add_entry** tables
- HUR tables cannot have const entries
- To use the HUR counter API in conjunction with the HUR insertion API, **nv_support_timeout** must be enabled
- Table size must not be greater than 8M

The HUR table is designed to support table entry insertion and deletion at rates significantly higher than a P4Runtime controller / server model can support. Instead, the HUR model replaces gRPC/protobuf messaging with a compiler generated packed C structure, a DPL Runtime library SDK and a shared memory buffer/pool that a client application can enqueues the table entry for insertion/deletion. See DPL Runtime Controller SDK for details on the programming model.

10.10.1. C API Generation

When a table is marked with **nv_high_update_rate**, the compiler will generate a set of accompanying C header and implementation files in the output directory. For **program.p4**, the following files are generated:

program_dpl_p4rt.h

- A C header file with the P4Runtime IDs for the tables and actions used in the program.
- C structs for each table keyset and action parameters.
- P4Runtime APIs for inserting, removing and retrieving counters

program_dpl_p4rt.cpp

- An implementation of the P4Runtime APIs in program_dpl_p4rt.h.

program_dpl_shm_id.h

- A hash value for **program.p4**, which much match the hash for the program that is loaded into the device.

program_dpl_shm.h

- A C header file with the shared memory linear IDs for the tables and actions used in the program.
- Packed and aligned C structs for each table keyset and action parameters.
- Shared Memory APIs for inserting, removing and retrieving counters

program_dpl_shm.cpp

- an implementation of the Shared Memory APIs in program_dpl_p4rt.h.

10.10.2. Example and Usage

Consider the below example **hello_world.p4**

hello_world.p4

```
control main_flow(
    inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) my_shm_table_counter;

    action shm_forward(nv_logical_port_t port) {
        my_shm_table_counter.count();
        nv_send_to_port(port);
    }

    action shm_drop() {
        my_shm_table_counter.count();
        nv_drop();
    }

    action grpc_forward(nv_logical_port_t port) {
        nv_send_to_port(port);
    }

    action grpc_drop() {
```

```
        nv_drop();
    }

    table my_grpc_table {
        key = {
            headers.ipv4.protocol : exact;
            headers.tcp.src_port : exact;
            headers.tcp.dst_port : exact;
        }
        actions = {
            grpc_forward;
            grpc_drop;
        }
        default_action = grpc_drop;
        size = 128;
    }

    table my_shm_table {
        key = {
            headers.ipv4.src_addr : exact;
            headers.ipv4.dst_addr : exact;
        }
        actions = {
            shm_forward;
            shm_drop;
            NoAction;
        }
        default_action = NoAction;
        direct_counter = my_shm_table_counter;
        nv_high_update_rate = true;
        size = 128;
    }

    apply {
        if (!my_shm_table.apply().hit)
            my_grpc_table.apply();
    }
}
```

This example has P4 tables where one has the **nv_high_update_rate** set to **true** and the other does not use the high update rate feature.

10.10.2.1. P4Runtime APIs for Non-high Update Rate Tables

- P4Runtime IDs for the tables and actions used in **hello_world** program for the **my_grpc_table**.

  P4Runtime IDs

  ```
  /**
   * Table definitions for main_flow_my_grpc_table
   */
  #define DPL_P4RT_HELLO_WORLD_MAIN_FLOW_MY_GRPC_TABLE_TABLE_ID 36982097
  /**
   * P4RT IDs for main_flow_my_grpc_table's keys
   */
  #define DPL_P4RT_HELLO_WORLD_MAIN_FLOW_MY_GRPC_TABLE_HEADERS_IPV4_PROTOCOL_KEY_ID 1
  #define DPL_P4RT_HELLO_WORLD_MAIN_FLOW_MY_GRPC_TABLE_HEADERS_TCP_SRC_PORT_KEY_ID 2
  #define DPL_P4RT_HELLO_WORLD_MAIN_FLOW_MY_GRPC_TABLE_HEADERS_TCP_DST_PORT_KEY_ID 3

  #define DPL_P4RT_HELLO_WORLD_MAIN_FLOW_MY_GRPC_TABLE_GRPC_DROP_ACTION_ID 23410936

  #define DPL_P4RT_HELLO_WORLD_MAIN_FLOW_MY_GRPC_TABLE_GRPC_FORWARD_ACTION_ID 25120277

  #define DPL_P4RT_HELLO_WORLD_MAIN_FLOW_MY_GRPC_TABLE_GRPC_FORWARD_PORT_PARAM_ID 1
  ```

  All the auto-generated values follow the naming convention of **DPL_P4RT_{application_name}_{control_name}_{table_name}_{attribute_name}**

- C structs for each table keyset and action parameters.

  C structs for table keys and actions

  ```
  /* Keys Structure */
  struct dpl_p4rt_hello_world_main_flow_my_grpc_table_keys {
      uint8_t headers_ipv4_protocol;
      uint16_t headers_tcp_src_port;
      uint16_t headers_tcp_dst_port;
  };
  /* Parameters Structure */
  struct dpl_p4rt_hello_world_main_flow_my_grpc_table_grpc_drop_params {
  };
  struct dpl_p4rt_hello_world_main_flow_my_grpc_table_grpc_forward_params {
  ```

```
    uint32_t port;
};
/**
 * @brief
 * Table main_flow_my_grpc_table's action parameter's union
 */
union dpl_p4rt_hello_world_main_flow_my_grpc_table_action_params
{
    dpl_p4rt_hello_world_main_flow_my_grpc_table_grpc_drop_params_t grpc_drop;
    dpl_p4rt_hello_world_main_flow_my_grpc_table_grpc_forward_params_t grpc_forward;
};

/**
 * @brief
 * Table main_flow_my_grpc_table's table entry structure
 */
struct dpl_p4rt_hello_world_main_flow_my_grpc_table_entry {
    dpl_p4rt_hello_world_main_flow_my_grpc_table_keys_t keys;
    uint32_t action_id;
    union dpl_p4rt_hello_world_main_flow_my_grpc_table_action_params action_params;
};
```

- P4Runtime APIs for inserting entries, deleting entries and retrieving counters information
  - Inserting entries into the table

Insert entries

```
// Create and populate the table entry
dpl_p4rt_hello_world_my_grpc_table_entry_t entry;

// Set the key fields - match TCP traffic from port 8080 to port 443
entry.key.headers_ipv4_protocol = 6;           // TCP protocol
entry.key.headers_tcp_src_port = htons(8080);  // Source port 8080
entry.key.headers_tcp_dst_port = htons(443);   // Destination port 443

// Set the action - forward to port 1
entry.action_id = DPL_P4RT_HELLO_WORLD_MY_GRPC_TABLE_GRPC_FORWARD_ACTION_ID;
entry.action_params.grpc_forward.port = 1;

// Set up attributes if required
struct dpl_rt_controller_add_entry_attr attr;

// Insert the entry
result = dpl_p4rt_hello_world_my_grpc_table_insert_entry(
    device, &entry, &attr, &dpl_entry);

if (result != DOCA_SUCCESS) {
    DOCA_LOG_ERR("Failed to insert P4RT entry: %s", doca_error_get_descr(result));
    return -1;
}
```

  - Deleting entries from the table

Delete entries

```
/* Deletes an entry by calling the function whose argument is the handle to entry returned by the
insert entry function*/
doca_error_t result = dpl_p4rt_hello_world_delete_entry(*dpl_entry);
```

  - Retrieving counter information

Retreive counters

```
dpl_rt_controller_counter_data counter_data;
/* The counter_data structure will have the data after the API is called */
doca_error_t result = dpl_p4rt_hello_world_default_entry_counter(device,
    DPL_P4RT_HELLO_WORLD_MAIN_FLOW_MY_GRPC_TABLE_TABLE_ID
    &counter_data);
```

10.10.2.2. SHM APIs for High Update Rate Tables

- SHM linear IDs for the tables and actions used in **hello_world** program for the **my_shm_table**.

Linear IDs

```
/**
 * P4Table definitions for main_flow_my_shm_table
 */
#define DPL_SHM_HELLO_WORLD_MAIN_FLOW_MY_SHM_TABLE_TABLE_ID 0

#define DPL_SHM_HELLO_WORLD_MAIN_FLOW_MY_SHM_TABLE_NOACTION_ACTION_ID  0
#define DPL_SHM_HELLO_WORLD_MAIN_FLOW_MY_SHM_TABLE_SHM_FORWARD_ACTION_ID  1
#define DPL_SHM_HELLO_WORLD_MAIN_FLOW_MY_SHM_TABLE_SHM_DROP_ACTION_ID  2
```

- C structs for each table keyset and action parameters.

C structs for table keys and action

```
/**
 * @brief
 * P4Table main_flow_my_shm_table's keys structure
 */
struct dpl_shm_hello_world_main_flow_my_shm_table_keys {
    uint32_t headers_ipv4_src_addr;
    uint32_t headers_ipv4_dst_addr;
} __attribute__((__packed__));

/**
 * @brief
 * Action NoAction's parameter structure
 */
struct dpl_shm_hello_world_main_flow_my_shm_table_noaction_params {
} __attribute__((__packed__));

/**
 * @brief
 * Action main_flow.shm_forward's parameter structure
 */
struct dpl_shm_hello_world_main_flow_my_shm_table_shm_forward_params {
    uint32_t port;
} __attribute__((__packed__));

/**
 * @brief
 * Action main_flow.shm_drop's parameter structure
 */
struct dpl_shm_hello_world_main_flow_my_shm_table_shm_drop_params {
} __attribute__((__packed__));

/**
 * @brief
 * P4Table main_flow_my_shm_table's action parameter's union
 */
union dpl_shm_hello_world_main_flow_my_shm_table_action_params
{
    dpl_shm_hello_world_main_flow_my_shm_table_noaction_params_t noaction;
    dpl_shm_hello_world_main_flow_my_shm_table_shm_forward_params_t shm_forward;
    dpl_shm_hello_world_main_flow_my_shm_table_shm_drop_params_t shm_drop;
    uint8_t padding_1[8];
};

/**
 * @brief
 * P4Table main_flow_my_shm_table's table entry structure
 */
struct dpl_shm_hello_world_main_flow_my_shm_table_entry {
    dpl_shm_hello_world_main_flow_my_shm_table_keys_t keys;
    uint32_t action_id;
    uint8_t padding_1[4];
    union dpl_shm_hello_world_main_flow_my_shm_table_action_params action_params;
} __attribute__((__packed__));
```

- SHM APIs for inserting entries, deleting entries and retrieving counters information
  - Inserting entries into the High Update Rate Table

Insert entries

```
    // Allocate shared memory for the entry
    result = dpl_shm_hello_world_my_shm_table_alloc_entry_mem(
        device, dpl_entry, &shm_entry);

    if (result != DOCA_SUCCESS) {
        DOCA_LOG_ERR("Failed to allocate SHM entry: %s", doca_error_get_descr(result));
        return -1;
    }

    // Fill in the entry
```

```
    shm_entry->key.headers_ipv4_src_addr = htonl(src_ip);
    shm_entry->key.headers_ipv4_dst_addr = htonl(dst_ip);
    shm_entry->action_id = DPL_SHM_HELLO_WORLD_MY_SHM_TABLE_SHM_FORWARD_ACTION_ID;
    shm_entry->action_params.shm_forward.port = output_port;
    shm_entry->valid = 1;

    // Insert the entry
    result = dpl_shm_hello_world_my_shm_table_insert_entry(device, *dpl_entry, shm_entry);

    if (result != DOCA_SUCCESS) {
        DOCA_LOG_ERR("Failed to insert SHM entry: %s", doca_error_get_descr(result));
        return -1;
    }
```

- Deleting entries from the High Update Rate Table

Delete entries

```
/* Delete an entry */
doca_error_t result = dpl_shm_hello_world_delete_entry(*dpl_entry);
```

- Retrieving Counters information of High Update Rate Table

Retreive counters

```
dpl_rt_controller_counter_data counter_data;
/* The counter_data structure will have the data after the API is called */
doca_error_t result = dpl_shm_hello_world_default_entry_counter(device,
    DPL_SHM_HELLO_WORLD_MAIN_FLOW_MY_SHM_TABLE_TABLE_ID,
    &counter_data);
```

8.11.7.1.10.11 10.11. Action Parameters

P4 action parameters provide an interface to the control plane for specifying per entry values to user defined actions. These values are sometimes referred to as control-plane constants (versus compile-time constants), where the following restrictions apply:

- Usage is limited in scope to the enclosed P4 action body
- They cannot be assigned to (LHS of an assignment).
- They can be the RHS of an assignment. See P4 Language Support in DPL heading "Operators" for a list of supported operators.

10.11.1. Modifiable Action Parameters

The **@nv_modifiable** annotation allows certain P4 action parameters to be updated atomically in place at runtime, without deleting and reinserting the table entry. This enables high-rate, low-disruption updates of selected header-rewrite fields (for example, NAT addresses and ports) for both gRPC/P4Runtime and SHM insertion modes. For example:

```
action change_tuple(
    @nv_modifiable nv_ipv4_addr_t new_sip,
                   nv_ipv4_addr_t new_dip,
    @nv_modifiable bit<16>       new_sport,
                   bit<16>       new_dport)
{
    headers.ipv4.src_addr = new_sip;
    headers.ipv4.dst_addr = new_dip;
    nv_set_l4_src_port(headers, new_sport);
    nv_set_l4_dst_port(headers, new_dport);
}
```

In this example, **new_sip** and **new_sport** are atomically modifiable; **new_dip** and **new_dport** are not, using either the gRPC/P4Runtime **MODIFY** message or SHM modify APIs.

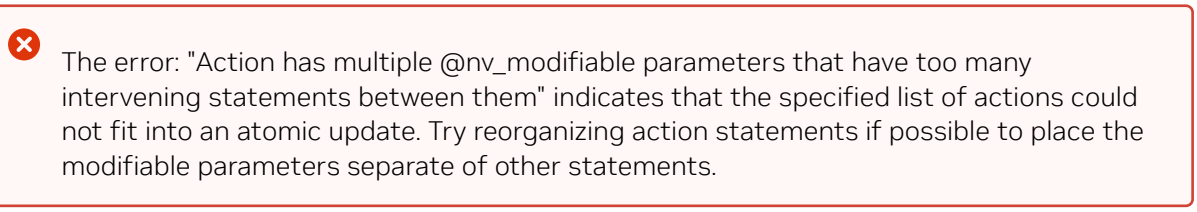
The following sematics apply for a table entry whose selected action has one or more `@nv_modifiable` parameters:

- The DPL runtime can perform an **atomic in-place update** of those parameters' values.
- The underlying hardware guarantees that no packet will see a partially updated set of values; the change is all-or-nothing at the level of the compiled modify-header resource.
- Match keys, action IDs, and non-annotated parameters remain **immutable** for the purpose of atomic modification. Changing them still requires delete + insert.

Compiler Validation and Restrictions

To ensure that a parameter can be updated atomically on the target hardware, the compiler enforces:

- All `@nv_modifiable` parameters of an action must map into a **single modify-header pattern/cache line** (64 bytes).
  - If a candidate parameter set would require touching multiple cache lines, compilation fails with an error.
- Currently, this support is limited to
  - **Modify-header** style assignment expressions (SET/ADD); header reformat (encapsulations), header insertion (`nv_push_headers`)
  - Other modifications such as header removal, packet mirroring and all other actions do not currently support atomic action parameter modifications.
- The annotation guarantees atomicity only if compilation succeeds; the compiler does not attempt to re-order modify-header instructions to make an unsupported pattern modifiable.



The error: "Action has multiple @nv_modifiable parameters that have too many intervening statements between them" indicates that the specified list of actions could not fit into an atomic update. Try reorganizing action statements if possible to place the modifiable parameters separate of other statements.

10.11.2. Modify Runtime APIs

The compiler generates helper APIs on top of the DPL runtime to simplify safe modification:

- For **P4Runtime** tables, a wrapper `*_modify_entry` helper is generated for the specific table and action. The controller API:
  - Supplies the full key and **only** the modifiable parameters.
  - The helper validates that all changes are limited to `@nv_modifiable` fields and uses a low-level "argument write" operation to atomically update the underlying modify-header resource.
- For **SHM insertion mode** tables, the compiler generates per-action helpers such as:

```
doca_error_t dpl_shm_main_ctrl_my_table_change_tuple_action_modify(
    struct dpl_rt_controller_entry *dpl_entry,
    uint32_t *new_sip,    // @nv_modifiable, optional
    uint16_t *new_sport   // @nv_modifiable, optional
);
```

These helpers:

- Validate that the entry belongs to the expected table/action.
- Read and back up the current parameter values from the opaque SHM entry.

- Patch only the modifiable fields.
- Issue a **dpl_rt_controller_table_entry_modify()** request.
- Roll back to the original values on failure.

Direct use of low-level SHM or P4Runtime APIs remains possible but may bypass these safety checks; atomicity is only guaranteed when the **@nv_modifiable** constraints are respected.

8.11.7.1.11 11. Packet IO

Controller Packet in (pipeline to controller) and Packet out (controller to pipeline) are supported in the DOCA Target Architecture. To send a packet to the controller with metadata:

- Create a struct with the annotation **@nv_controller_metadata("packet_in")**
- The struct must be exactly 32 bits in width (user must insert padding as needed)
- Use the extern action **extern void nv_send_to_controller<PACKET_META>(in PACKET_META pkt_in_meta);**

To check if a packet that has ingressed the pipeline from the controller:

- Add as a match key "**std_meta.ingress_port**" to a P4 table key set
- Add an entry that specifies the key value to be the **p4_controller_port_id** defined in the DPL Port ID Mapping step.

8.11.7.1.12 12. P4Runtime Extern Entities

DPL-specific extern entities are extended per the *P4Runtime Specification* (v1.4.1). These extensions encode vendor specific messages using the **google.protobuf.Any** type to facilitate custom resource configuration.

8.11.7.1.12.1 12.1. NvPSP

The following **p4info** Protobuf extension allows a controller to initialize the NvPSP resource and specify its size:

```
// SPDX-FileCopyrightText: Copyright (c) 2025 NVIDIA CORPORATION & AFFILIATES. All rights reserved.
// SPDX-License-Identifier: LicenseRef-NvidiaProprietary
//
// NVIDIA CORPORATION, its affiliates and licensors retain all intellectual
// property and proprietary rights in and to this material, related
// documentation and any modifications thereto. Any use, reproduction,
// disclosure or distribution of this material and related documentation
// without an express license agreement from NVIDIA CORPORATION or
// its affiliates is strictly prohibited.

syntax = "proto3";
package nvidia.dpl.config.v1;

// a copy of the base P4Ids message with the vendor values
message NvDPLIds {
  enum Prefix {
    UNSPECIFIED = 0;
    NV_PSP = 0x8b;
  }
}

message NvPSPInfo {
  NvPSP psp = 1;
}

message NvPSP {
  // number of entries in the psp array
  uint64 size = 1;
}
```

8.11.7.2 P4 Language Support in DPL

This section outlines the P4-16 language features supported by the DOCA Pipeline Language (DPL) and notes any known deviations or limitations. All references are based on the P4 Language Specification version v1.2.5.

8.11.7.2.1 Introduction

NVIDIA® BlueField® networking platforms (DPU or SuperNIC) are designed to accelerate data center workloads by offloading and isolating key infrastructure tasks, including networking, storage, and security. BlueField-3 combines ASIC performance with programmable flexibility, supporting up to 400G connectivity—ideal for AI and HPC environments.

To optimize for high-performance hardware, the P4 implementation on BlueField differs from generic software targets. As a result, DOCA Pipeline Language (DPL) selectively supports P4 features that align with its target architecture. This section details which P4 features are currently supported, restricted, or unsupported.

> A feature labeled as "unsupported" is either not implemented or has not been validated. The compiler may explicitly reject such usage or accept it only if resolved at compile time (e.g., via constant folding). However, acceptance does not mean the feature is supported; do not rely on accepted, unsupported features.

8.11.7.2.2 P4 Language Features

The DPL compiler implements a performance-oriented subset of P4-16, aligned with the capabilities of the BlueField hardware. Unsupported features are typically those that require runtime flexibility (e.g., dynamic parsing or general-purpose control logic) which are not efficient in ASIC-based systems. Developers should reference this section during development to ensure compatibility and portability of P4 programs targeting DPL.

8.11.7.2.2.1 Identifiers

- Identifiers starting with `__` are reserved for internal compiler use
- All other identifiers conform to P4 spec §6.4.1

8.11.7.2.2.2 Data Types

Type	Support Level	Notes
`bool`	Supported	Fully supported
Arbitrary-precision int	Limited	Only for literals (see spec §7.1.6.5)
`int` (signed integers)	Unsupported	
String literals	Unsupported	Accepted, but no operations or validity checks (see spec §6.4.3.3). Only supported in annotations.
Bit strings	Supported	Subject to hardware resource limits

Refer to section "Operators" for support of operations on values with these types.

8.11.7.2.2.3 Derived Types

Type	Support Level	Notes
`enum`	Supported	As per spec §7.2.1, with restrictions based on target-supported types
`header`	Supported	All field types except `varbit<>` (only supported via `NvOptionParser`). See spec §7.2.2.
`header_stack`	Unsupported	
`struct`	Supported	Must use only supported field types. See spec §7.2.5.
`union`	Unsupported	
`tuple` / `list`	Supported	As per spec §7.2.6
`extern` types	Limited	Only the externs provided with the DPL compiler are allowed; no custom externs. See spec §7.2.9.
Type specialization	Supported	As per spec §7.2.10

8.11.7.2.2.4 Statements and Expressions

Feature	Support Level	Notes
`assignment`	Limited	L-values are restricted. Cannot assign to method calls, `packet_out` metadata, flex-header fields, action parameters, or unsupported metadata fields. Only some fixed headers are valid L-values.
`op-assignment (+=, -=)`	Supported	See DLP Expressions, Header Field Support for details.
`if` / conditionals	Limited	Only within control `apply` blocks; must evaluate to a `bool`. Disallowed in `action` blocks.
`switch`	Limited	Only within control `apply` blocks; switch expression must be `bool`. Disallowed in `action` blocks. Fall-through, default, and empty `switch` cases are supported per spec §11.7.

8.11.7.2.2.5 Operators

The P4-16 language specification lists a wide variety of operations that the language accepts for the supported data types (see Section 8). The table below lists the operators that are officially supported by the NVIDIA P4 compiler:

Operator	Compile-time value	P4 action parameter value	Runtime value	Spec section
bool && bool	✔	✔	✔	8.5
bool || bool	✔	✔	✔	8.5
! bool	✔	✔	✔	8.5
bool == bool	✔	✘	✘	8.5
bool != bool	✔	✘	✘	8.5
bit<W> == bit<W>	✔	✔	✔	8.6
bit<W> != bit<W>	✔	✔	✔	8.6
bit<W> << integer	✔	✔	✔	8.6
bit<W> >> integer	✔	✔	✔	8.6
bit<W>[H:L]	✔	✔	✔	8.6
bit<W> > bit<W>	✔	✔	✔	8.6
bit<W> >= bit<W>	✔	✔	✔	8.6
bit<W> < bit<W>	✔	✔	✔	8.6
bit<W> <= bit<W>	✔	✔	✔	8.6
All explicit casts between supported types	✔	✔	✔	8.11.1
All implicit casts between supported types	✔	✔	✔	8.11.2
bit<W>..bit<W>	✔	✔	✘	8.15.4
Assignment to user struct fields	✔	✔	✔	8.16
Assignment to packet-in struct fields	✔	✔	✔	8.16
All operations on header fields	✔	✔	✔	8.17
Method calls	✔	✔	✘	8.20
Function calls with positional args	✔	✔	✔	8.20
Extern constructor invocations	✔	✘	✘	8.21
Parser constructor invocations	✔	✘	✘	8.21

Operator	Compile-time value	P4 action parameter value	Runtime value	Spec section
Control constructor invocations	✔	✘	✘	8.21
Package constructor invocations	✔	✘	✘	8.21
Decrement (-=)	✔	✔	✔	-
Increment (+=)	✔	✔	✔	-

We don't have a way to export this macro.

8.11.7.2.2.6 Variables

Variables are supported in accordance with the following spec items:

- Constants (*spec 11.1*)
  - "Compile-time known values" are evaluated on a best-effort basis. It is possible that a compile-time known value may not be recognized by the compiler as such.
- Variables (*spec 11.2*)
- Instantiations (*spec 11.3*)
  - Instantiations with abstract methods (*spec 11.3.1*) are allowed in BlueField Target Architecture
  - Named arguments are not supported

Variables may be declared in any of the locations described in (*spec 11.2*) and follow the scope rules described there.

The compiler will emit errors for uninitialized values. In some cases where a struct is partially initialized, only a warning may be produced. In some cases there may be no error emitted when an uninitialized struct field is accessed. The accessed field will then contain an undefined value.

⚠ On BlueField-3 the maximum amount of user metadata is 256 bits at any single point in the program. The remaining register space is utilized internally by the firmware.

8.11.7.2.2.7 Control Apply Block

The `apply` block within a control logic supports a specific subset of P4 statements to ensure deterministic execution and hardware compatibility.

Supported Statements

The following statements are supported within the control flow:

- `table.apply()` calls

- `if` / `else` statements
- `switch` statements
- Extern function and method calls
- Assignment statements (using supported operators)
- The empty statement (`;`)
- `return` statements

> ⚠ The `exit` statement is not supported in this target architecture.

Loop Constructs

The `for` loop is supported, provided it adheres to strict structural restrictions. These constraints are necessary to ensure the compiler can statically verify loop termination (preventing infinite loops in the hardware pipeline).

Both `break` and `continue` statements are supported within the loop body.

Loop Constraints

To ensure validity, the loop must follow this specific decrementing pattern:

1. Init Clause: Must be a single assignment defining the loop variable.
2. Condition Clause: Must test that the loop variable is not equal to zero (`!= 0`).
3. Update Clause: Must be a single operation-assignment that decrements the variable by 1 (`-= 1`).
4. Body Constraint: The loop body must not modify the loop control variable.

Example Implementation

for loop

```
bit<32> sum = 1;

/* Valid Decrementing Loop */
for (
    bit<8> i = headers.ipv4.ttl;  // Init statement
    i != 0;                       // Condition
    i -= 1                        // Update statement
) {
    if (i == 64) {
        continue;
    } else if (i == 1) {
        break;
    } else {
        sum += 1;
    }
}
```

Hardware Constraints: Steering Hop Limits

The BlueField hardware enforces a global limit on the absolute number of steering hops in the pipeline. This acts as a safety mechanism to prevent infinite loops in the data path.

- Scope: This "TTL" (Time To Live) or hop limit is managed by the firmware. It is an accumulation over all steering domains (not just the DPL program).
- Validation: Because this is a runtime hardware state dependent on the total system configuration, it is out of the scope of DPL to validate at compile time.

- Behavior: Once the limit is exceeded, the packet is immediately dropped.

Troubleshooting Steering Violations

Violations of this limit can be detected by monitoring specific steering failure counters exposed to the kernel.

Relevant counters:

- `generated_pkt_steering_fail`
- `handled_pkt_steering_fail`

These can be inspected using the `devlink` health reporter. See the command `devlink health diagnose` in the Linux Kernel Documentation.

8.11.7.2.2.8 Table Apply

Calling the apply method on a table produces a result object that can return specific attributes about the table lookup outcome:

- bool hit
- bool miss
- enum action_run

8.11.7.2.2.9 Actions

Actions support the same statements as controls except for the following:

- `table.apply()` calls are not supported
- Conditional statements - `if` and `switch` - are not allowed

Actions support the same expressions as controls except for the following:

- Boolean logical operators - `&&`, `||`, ternary operator are not supported
- Comparisons (`==`, `!=`, etc.) are not supported

8.11.7.2.3 P4Runtime Features

The DPL compiler generates a `p4info` file compliant with the P4Runtime Specification (v1.4.1) for all supported DPL features.

8.11.7.2.3.1 Supported Annotations

The compiler supports the following package annotations as outlined in the P4Runtime PkgInfo Message. These allow for better documentation and versioning of the P4 program.

- `@pkginfo` with the following fields:
  - `name`
  - `version`
  - `doc`
  - `arch`
- `@brief`

- `@description`

8.11.7.2.3.2 Limitations and Unsupported Features

The following P4Runtime features are not supported in this target architecture:

Unsupported P4Info Objects

- ValueSet
- Register
- Digest

Unsupported P4 Entity Messages

The following runtime entities cannot be manipulated via P4Runtime on this target:

- Packet Replication:
  - `PacketReplicationEngineEntry`
  - `MulticastGroupEntry`
  - `CloneSessionEntry`
- State and Metadata:
  - `ValueSetEntry`
  - `RegisterEntry`
  - `DigestEntry`

8.11.7.3 DPL Installation Guide

8.11.7.3.1 Introduction

The DPL Development Container (`dpl_dev`) bundles several tools for compiling, controlling, and debugging DPL programs.

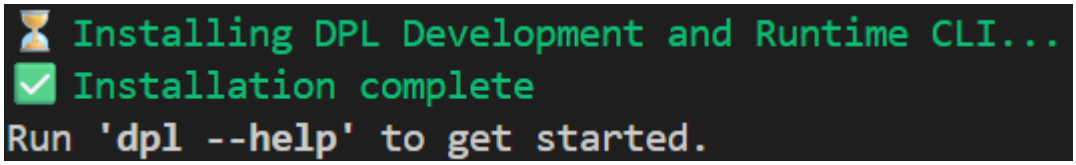
> First, select and pull the appropriate container tag that matches the DOCA version you intend to use. You can find available releases at NVIDIA DOCA DPL Container Tags.
>
> `docker pull nvcr.io/nvidia/doca/dpl_dev:X.X.X-docaX.X.X-host`

We'll use `nvcr.io/nvidia/doca/dpl_dev:1.4.0-doca3.3.0-host` for all examples in this page.

Running the container requires a set of base `docker run` options to work as intended.
Most use cases involve in extending the base options, for example adding mounts to files the tools interact with. We recommend to first understand the base options before you extend them with your own.

8.11.7.3.2 Installation

```
docker run -it -e HOME=${HOME} -v ${HOME}:${HOME} -u $(id -u):$(id -g) nvcr.io/nvidia/doca/dpl_dev:1.4.0-doca3.3.0-host install
```



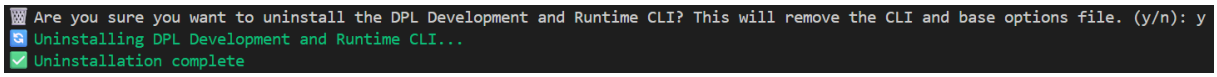
8.11.7.3.2.1 The options file

The installation creates an options file at **~/dpl.opts** which contains options to be passed to the **docker run** command running the container.

Removing any of the default options is not recommended.
However adding more options may be needed per your use case, the options file contain commented out examples of such use cases.

8.11.7.3.3 Uninstallation

```
docker run -it -e HOME=${HOME} -v ${HOME}:${HOME} -u $(id -u):$(id -g) nvcr.io/nvidia/doca/dpl_dev:1.4.0-doca3.3.0-
host uninstall
```



8.11.7.3.4 Working with dpl

> ⚠ Running **dpl** requires access to **docker** so either run it with user **root** or with non-root user that is part of the docker group.

Running the installed **dpl** executes a CLI with commands of the bundled developer tools.

```
dpl --help
```

```
dpl - DPL Development and Runtime CLI

Usage:
  dpl COMMAND
  dpl [COMMAND] --help | -h
  dpl --version | -v

Commands:
  compile        Runs the DPL NVP4 Compiler (alias command: nvp4c)
  admin          Runs the DPL Admin
  nspect         Runs the DPL Nspect
  debugger       Runs the DPL Debugger
  p4runtime_sh   Runs the P4Runtime Shell

Options:
  --help, -h
    Show this help

  --version, -v
    Show version number

Environment Variables:
```

```
DPL_COMPILE_DEFAULT_ARGS
  Default arguments for the DPL Compile command

DPL_ADMIN_DEFAULT_ARGS
  Default arguments for the DPL Admin command

DPL_NSPECT_DEFAULT_ARGS
  Default arguments for the DPL Nspect command

DPL_DEBUGGER_DEFAULT_ARGS
  Default arguments for the DPL Debugger command

DPL_P4RUNTIME_SHELL_DEFAULT_ARGS
  Default arguments for the P4Runtime Shell command
```

Using the CLI is self explanatory, use the **--help** flag for further help on each command. For example, **dpl compile --help** will show the compiler tool's usage and full help message.

The environment variables **DPL_..._DEFAULT_ARGS** should be defined in the options file, a commented out template already exists.

8.11.7.3.5 Next Steps

1. Compiling DPL Applications that you wish to examine using the developer tools.
2. Loading DPL Applications can now be done with the compilation outputs.
3. Use the DOCA Pipeline Language Developer Tools to further examine the DPL programs correct operation.

8.11.7.4 Compiling DPL Applications

This page describes how to use the NVIDIA DPL compiler to build DPL applications.

8.11.7.4.1 Introduction

DOCA Programming Language (DPL) sample applications serve as educational resources that demonstrate how to program on the NVIDIA BlueField networking platform (DPU or SuperNIC) using DPL.

The NVIDIA DPL compiler (**nvp4c**) is provided as an executable within a self-contained Docker image. This image is available to partners via the NVIDIA NGC platform. To access it, log in to NVIDIA NGC and download the latest DPL Docker containers.

8.11.7.4.2 Installation

- For system requirements, refer to DPL Runtime Service.
- For compiler installation instructions refer to the DPL Installation Guide.
- For information about the development environment, refer to the DOCA Pipeline Programming Model and the DPL System Overview.

8.11.7.4.3 Dependencies

- BlueField-3 DPU or CX-9 SuperNIC
- Ubuntu 22.04 hosts (x86) or greater
- A P4Runtime controller based on API version 1.4.0 or greater

8.11.7.4.4 Compiling Applications

The DPL compiler is provided with all its dependencies in the **dpl-dev** container. For details on how to obtain and install the Docker images refer to the DPL Installation Guide.

A script command, **dpl nvp4c**, is provided to execute the DPL compiler from the **dpl-dev** container:

```
dpl nvp4c --target [doca|doca-bf3|doca-cx9] nvp4c_args* program.p4
```

- **--target** – the DPL target architecture
- **dplp4c_args** – arguments to be passed to the compiler (e.g., **program.p4**)

The following example compiles **hello_packet** using DPL:

```
user@host:~/p4-samples/doca/hello_packet$ dpl nvp4c --target doca hello_packet.p4
Generating compiler output in "_out"
```

The following files in this example are produced in the **_out** directory:

File name	Description
compiler.log	Log of any compiler warnings or errors to the specified program
hello_packet.dplconfig	Binary blob, containing all the data needed for the DPL Runtime daemon to load the program
hello_packet.p4info.txt	P4Runtime protobuf file, in text format

Additional **nvp4c** specific arguments include:

```
--help                                  Print this help message
--version                               Print compiler version
-I path                                 Specify include path (passed to preprocessor)
-D arg=value                            Define macro (passed to preprocessor)
-U arg                                  Undefine macro (passed to preprocessor)
-E                                      Preprocess only, do not compile (prints program on stdout)
-M                                      Output `make` dependency rule only (passed to preprocessor)
-MD                                     Output `make` dependency rule to file as side effect (passed to
preprocessor)
-MF file                                With -M, specify output file for dependencies (passed to preprocessor)
-MG                                     with -M, suppress errors for missing headers (passed to preprocessor)
-MP                                     with -M, add phony target for each dependency (passed to preprocessor)
-MT target                              With -M, override target of the output rule (passed to preprocessor)
-MQ target                              Like -Mt, override target but quote special characters (passed to
preprocessor)
--nocpp                                 Skip preprocess, assume input file is already preprocessed.
--Wdisable[=diagnostic]                 Disable a compiler diagnostic, or disable all warnings if no diagnostic is
specified.
--Winfo[=diagnostic]                    Report an info message for a compiler diagnostic.
--Wwarn[=diagnostic]                    Report a warning for a compiler diagnostic, or treat all info messages as
warnings if no diagnostic is specified.
--Werror[=diagnostic]                   Report an error for a compiler diagnostic, or treat all warnings as errors
if no diagnostic is specified.
--maxErrorCount errorCount              Set the maximum number of errors to display before failing.
--target target                         Compile for the specified target device.
--metrics metric1[,metric2]             Select which code metrics will be collected.
                                        Valid options: all, loc, cyclomatic, halstead, unused-code, duplicit-code,
                                        nesting-depth, header-general, header-manipulation, header-modification,
                                        match-action, parser, inlined, extern.
--odir out_directory                    Write output to out directory
--enable feature[,feature]*|--help      Enable a feature, or comma-separated list of features
--disable feature[,feature]*|--help     Disable a feature, or comma-separated list of features
-g                                      Enable debugging via dplnspect. Equivalent to --debug-mode=auto
--debug-mode debug_mode                 Enable debugging via DPLnspect. Options: extern, auto
--log filename                          [Compiler debugging] Compiler log file name
--list-pass-params                      List known pass parameters and exit
--pass-param pass.key=value             Pass parameter (repeatable). Example: --pass-param
merge_adjacent_hwtables.max_merges=256
```

```
-v logLevel                          [Compiler debugging] Backend log level
-T fileLogSpec                       [Compiler debugging] Backend logging level per file
```

The binary `<sample_name>.` is created under `_out/<sample_name>.dplconfig`.

The P4Runtime file is created under `_out/<sample_name>.p4info.txt`.

8.11.7.4.5 Compiling Applications in Debug Mode

To enable packet debugging, you must ensure the following:

- Compile your DPL program with the `--debug-mode=extern` flag. This enables using the debug flow.
- Ensure you have at least one `nv_send_debug_pkt()` call in your DPL program. A debug packet will be emitted where `nv_send_debug_pkt()` is placed, and represents the program state at that point.
- Alternatively, compile your DPL program using `--debug-mode=auto` (equivalent to `-g`).
- `auto` automatically inserts debug points before each statement in each control's apply block, and at the end of each P4 action block. The user can still manually place calls to `nv_send_debug_pkt()` in a P4 action.

Packet debugging is supported for packets received from a wire port (RX direction) and for packets sent by the CPU (TX direction).

You may optionally include a cookie in your `nv_send_debug_pkt()` call (e.g., `nv_send_debug_pkt(8w0x42)`). This cookie can help distinguish between different debug extern calls and will appear in the debugging output.

If `nv_send_debug_pkt()` is placed in an apply block and `debug-mode=auto`, the user's specified debug point will override the automatic debug point.

8.11.7.5 Loading DPL Applications

NVIDIA DPL programs are deployed to the NVIDIA® BlueField® networking platform (DPU or SuperNIC) using the P4Runtime API. This allows platform-independent, standards-based integration with SDN controllers.

8.11.7.5.1 Introduction

The DPL compiler generates a pipeline binary optimized for execution on BlueField. Pipeline loading and control—such as installing the program and populating P4 tables—are handled via the P4Runtime API, an open and well-defined interface.

The P4Runtime server, running on the BlueField device, enables a P4 Controller to:

- Connect over gRPC
- Set the `ForwardingPipelineConfig` (i.e., install the compiled DPL binary and `p4info`)
- Query the device for its current pipeline config and table state

- Maintain runtime P4 tables as defined in the DPL source

This model enables integration with open-source, proprietary, or custom-built controllers in a standardized way.

8.11.7.5.2 Prerequisites

Before loading a DPL application, ensure the following services and components are properly set up:

- DPL Runtime Service is running and configured on the BlueField (Arm side). See the DPL Container Deployment page for setup instructions.
- DPL Development Container and the **p4runtime_sh.sh** launch script are installed on the host. See the DPL Installation Guide for more details.

8.11.7.5.3 Loading the Application

A DPL application can be loaded using:

- A custom P4Runtime controller
- The NVIDIA-supplied Python controller
- An open source controller

In the following example, we'll be using the P4Runtime controller bundled within the DPL Development Container.

8.11.7.5.3.1 Using p4runtime_sh Script

```
./p4runtime_sh.sh help
```

```
usage: dpl p4runtime_sh [--help] [--grpc-addr <dpl_rtd_ip:port>] [--device-id <device ID>] [--config
program.p4info.txt,program.dplconfig]
```

Arguments:

- **--help** - Show all the command line arguments.
- **--config** - A list of the DPL program compilation output for P4info and dplconfig (Compiling DPL Applications).
- **--grpc-addr** - Address of the DPL Runtime daemon and the P4Runtime port as specified in the DPL Runtime DPL Service Configuration.
- **--device-id** – ID of the target device

> To configure the **dpl** command see DPL Installation Guide (Configuring Launch Scripts). The docker image used can be changed by setting the environment variable $DPL_DEV_IMAGE.
>
> The examples below assumes that everything was pre-configured using the **dpl-dev** install procedure.

Example Command

```
dpl p4runtime_sh --grpc-addr 192.168.100.2:9559 --device-id 1000 --config
hello_packet.p4info.txt,hello_packet.dplconfig
```

8.11.7.5.3.2 p4runtime_sh python application

After successfully loading a program, the script launches an interactive Python shell (`p4runtime_sh` python application) connected to the P4Runtime server on the dpl-rt-service running on the DPU.
From here, you can inspect and manage tables entries using this tool. For more details about using the `p4runtime_sh` application see:

- Our P4 Runtime Controller documentation.
- Official p4runtime-shell project documentation.

8.11.7.5.4 P4Runtime Optimizations

DPL table entries are added via a P4Runtime controller, which may run remotely or locally on the device.

- The standard gRPC-based controller model supports ~50K rule insertions per second

8.11.7.5.5 Troubleshooting

Loading a program might fail when the dpl-rt-service processes the compiled program due to various reasons (mostly configuration issues), in such cases, you might see error message like:

Program load failure example

```
# dpl p4runtime_sh --device-id 1000 --config p.p4info.txt,p.dplconfig
CRITICAL:root:Error when setting config
CRITICAL:root:P4Runtime RPC error (INVALID_ARGUMENT): Error setting program: Failed creating P4 tables
```

Refer to the following for common troubleshooting issues:

Checkpoint	Guidance/Command
Inspect the `dpl-rt-service` log for more error details	There are two methods to get the `dpl-rt-service` log: 1. Using the dpl admin utility: `# dpl admin -a 192.168.100.2:9600 get-log-recent --number_of_lines 50` 2. Manually logging into the DPU and finding the log file at: `/var/log/doca/dpl_rt_service/dpl_rtd.log`. Look for the last info and error messages at the end of the log file.
Inspect the current `dpl-rt-service` configurations	Use the dpl admin utility: `# dpl admin -a 192.168.100.2:9600 config` Verify that all required ports are present with the expected IDs. The devices and ports correspond to the `/etc/dpl_rt_service/devices.d/*.conf` configuration file(s) on the DPU.

Checkpoint	Guidance/Command
Inspect the DPU non-volatile configurations (nvconfig)	Use the dpl admin utility: `# dpl admin -a 192.168.100.2:9600 nv-config --device_id 1000` Check if there are missing required firmware settings per DPL Runtime Service (Firmware Settings).
Inspect the hardware/ firmware capabilities configurations	Use the dpl admin utility: `# dpl admin -a 192.168.100.2:9600 hca-capabilities --device_id 1000` Check if your hardware/firmware combination supports the required features.
The **dpl-rt-service** log contains error message like: **[dpl_port_id_to_ib_port_num] Cannot find port ID (16)**	This indicates that the DPL program refers to a port ID that does not exist in the device configuration files. Fix the program to use the correct port ID(s), and/or update the **/etc/ dpl_rt_service/devices.d/*.conf** configuration file(s) on the DPU.
The **dpl-rt-service** log contains error message like: **NV_HWS[table_process_attr: 434]: Unified FDB not supported by current FW**	This indicates that the DPU does not have a Firmware version that meets the minimum required version. Upgrade the DOCA and/or the Firmware version on the DPU to the minimum required version as documented at DPL Runtime Service (Software Prerequisites).

8.11.7.6 Sample DPL Applications

This section provides NVIDIA DOCA Pipeline Language sample applications that demonstrate how to develop, compile, and deploy programs using the NVIDIA DOCA Pipeline Language (DPL). These examples are intended to help developers become familiar with DPL concepts and runtime behavior on both host and BlueField®-3 platforms.

8.11.7.6.1 Sample Prerequisites

DPL programs can be compiled on any host machine with the DPL compiler installed. Once compiled, the output binary can be loaded onto:

- The same host machine (for development/debugging purposes), or
- A BlueField-3 DPU where the DPL Runtime daemon is running

Refer to the following documents for setup and troubleshooting:

- DPL Runtime Service – For instructions on installing and configuring DPL on BlueField
- DPL Release Notes – For known issues and guidance on installation, compilation, and execution of DPL sample applications

8.11.7.6.2 Running the Sample

1. Compile the DPL program using the DPL compiler.
2. Locate the generated output files in the compiler output directory:

- `.p4info` – Describes the program's tables and actions
- `.dplconfig` – Contains the binary pipeline configuration

3. Load the program onto the BlueField target using a P4Runtime controller (e.g., `p4runtime_sh.sh` , NVIDIA's controller, or a custom solution)

Once loaded, you can use the controller to populate flow tables, query runtime state, and validate pipeline behavior.

8.11.7.6.3 Sample Applications

- Hello Packet Example
- GTP Parsing Example
- GTP Tunnel Encapsulation Example
- GTP Full Application
- Geneve TLV Parsing Example
- VXLAN Tunnel Gateway Example
- Connection Tracking Example
- TCP State tracking
- Host-based Networking Example
- Equal-Cost Multi-Path Routing
- Entropy in Hash-Based Load Balancing
- NAT64 Example
- Add on Miss
- PSP encryption

8.11.7.6.4 Hello Packet Example

This example demonstrates the most basic pipeline in the DOCA target architecture. It consists of:

- Match on L2 destination MAC address
- Action to either forward to a P4 port ID, drop, or no action

8.11.7.6.4.1 Sample Code

The following #includes are required for every DPL program. They define the DOCA target architecture. In particular, they contain things like the default DOCA parser, default headers struct, etc. Note that the symbol names are prefixed with "nv_" and are reserved for NVIDIA DOCA TA usage.

```
#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>
```

The DOCA TA features a single control, which requires a headers struct and standard metadata. User metadata and packet out metadata, defined by the user, are optional.

```
control hello_packet(
    inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
```

```
) {
    action drop() {
        nv_drop();
    }

    action forward(bit<32> port) {
        nv_send_to_port(port);
    }

    table forward_table {
        key = {
            headers.ethernet.dst_addr : exact;
        }
        actions = {
            drop;
            forward;
            NoAction;
        }
        default_action = forward(3);
        const entries = {
            (48w0x001111111111) : forward(1);
            (48w0x002222222222) : forward(2);
            (48w0x00dddddddddd) : drop();
            (48w0x00aaaaaaaaaa) : NoAction();
            (48w0x00bbbbbbbbbb) : NoAction();
        }
    }
    apply {
        forward_table.apply();
    }
}
```

Finally, the main package must be instantiated:

```
NvDocaPipeline(
    nv_fixed_parser(),
    hello_packet()
) main;
```

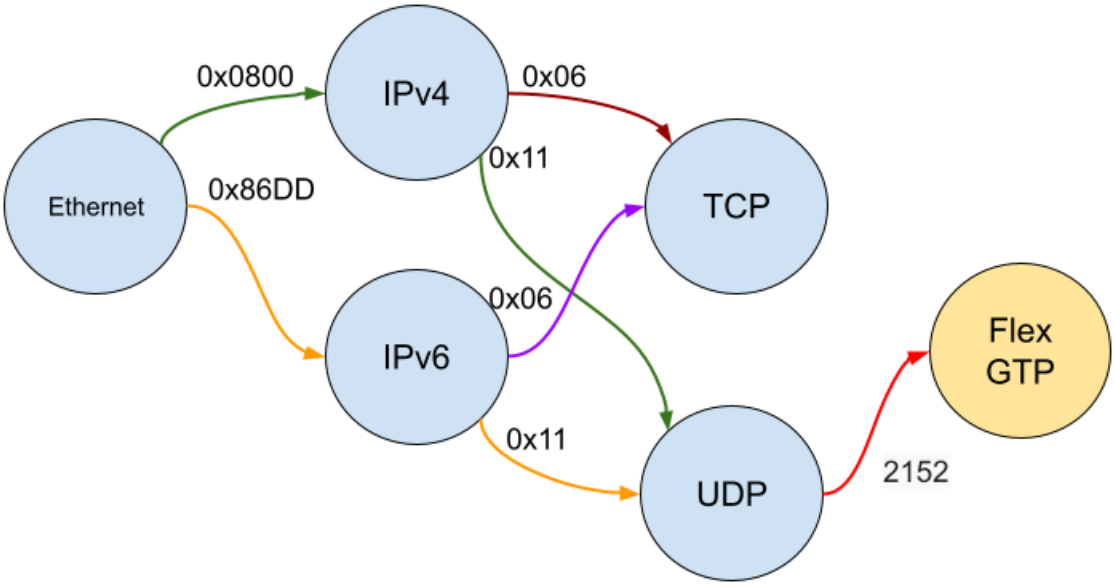
See below for the complete DPL example.

```
1   /*
2    * SPDX-FileCopyrightText: Copyright (c) 2024-2025 NVIDIA CORPORATION & AFFILIATES. All rights
    reserved.
3    * SPDX-License-Identifier: LicenseRef-NvidiaProprietary
4    *
5    * NVIDIA CORPORATION, its affiliates and licensors retain all intellectual
6    * property and proprietary rights in and to this material, related
7    * documentation and any modifications thereto. Any use, reproduction,
8    * disclosure or distribution of this material and related documentation
9    * without an express license agreement from NVIDIA CORPORATION or
10   * its affiliates is strictly prohibited.
11   */
12
13  #include <doca_model.p4>
14  #include <doca_headers.p4>
15  #include <doca_externs.p4>
16  #include <doca_parser.p4>
17
18  /*
19   * This basic application demonstrates a simple match/action pipeline using NVIDIA
20   * DOCA Target Architecture.
21   * - Match on destination MAC address
22   * - Forward or drop the packet
23   */
24
25  control hello_packet(
26      inout nv_headers_t headers,
27      in nv_standard_metadata_t std_meta,
28      inout nv_empty_metadata_t user_meta,
29      inout nv_empty_metadata_t pkt_out_meta
30  ) {
31      action drop() {
32          nv_drop();
33      }
34
35      action forward(bit<32> port) {
36          nv_send_to_port(port);
37      }
38
39      table forward_table {
40          key = {
41              headers.ethernet.dst_addr : exact;
42          }
43          actions = {
44              drop;
45              forward;
46              NoAction;
47          }
48          default_action = forward(3);
49          const entries = {
50              (48w0x001111111111) : forward(1);
51              (48w0x002222222222) : forward(2);
52              (48w0x00dddddddddd) : drop();
53              (48w0x00aaaaaaaaaa) : NoAction();
54              (48w0x00bbbbbbbbbb) : NoAction();
```

```
        }
    }
    apply {
        forward_table.apply();
    }
}

NvDocaPipeline(
    nv_fixed_parser(),
    hello_packet()
) main;
```

8.11.7.6.5 GTP Parsing Example

This example demonstrates how to add a simple flex parser node to the existing hardware defined parse graph.



8.11.7.6.5.1 Sample Code

This example starts with some basic definitions.

```
#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>

const bit<32> WIRE_PORT = 32w00;
const bit<32> GTP_VPORT = 32w01;
const bit<32> DEFAULT_VPORT = 32w04;

struct metadata_t {
}
#define GTP_U_PORT 2152
```

Then we define the GTP-U version 1 header.

```
header Gtp_v1_h {
    bit<3>        version;                /** For GTPv1, this has a value of 1. */
    bit           protocol_type;          /** GTP (value 1) from GTP' (value 0) */
    bit           reserved;
    bit           extension_header_flag; /** extension header optional field. */
    bit           seq_number_flag;        /** Sequence Number optional field */
    bit           n_pdu_number_flag;      /** N-PDU number optional field */
    bit<8>        message_type;           /** types of messages are defined in 3GPP TS 29.060 section 7.1 */
    bit<16>       message_length;         /** length of the payload in bytes */
    bit<32>       teid;                   /** Tunnel endpoint identifier */
```

```
    bit<16>       sequence_number;       /** optional */
    bit<8>        n_pdu_number;          /** optional */
    bit<8>        next_extension_hdr_type; /** optional if any of the E, S, or PN bits are on. The field must be
interpreted only if the E bit is on */
}
```

Then we add **NV_FIXED_HEADERS** to the headers struct, along with the new GTP header.

```
struct headers_t {
    NV_FIXED_HEADERS
    Gtp_v1_h   gtpv1;
}
```

Using the **nv_transition_from** annotation, the GTP parser state is connected as a select transition from the UDP state.

```
parser packet_parser(packet_in packet, out headers_t headers) {
    NV_FIXED_PARSER(packet, headers)

    @nv_transition_from("nv_parse_udp", GTP_U_PORT)
    state parse_gtp
    {
        packet.extract(headers.gtpv1);
        transition accept;
    }
}
```

The control example uses a single flow table that matches on input port and GTP tunnel endpoint ID. The policy is then to forward the GTP packet to a port or drop the packet.

```
/**
 * This control admits GTP packets only if the tunnel ID matches
 *
 */
 control gtp_tunnel(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) gtp_counter;

    action send_to_port_gtp_table(nv_logical_port_t port) {
        gtp_counter.count();
        nv_set_l4_src_port(headers, 1234);
        nv_send_to_port(port);
    }

    action drop_gtp_table() {
        gtp_counter.count();
        nv_drop();
    }

    table gtp_table {
        key = {
            std_meta.ingress_port: exact;
            headers.gtpv1.teid: exact;
        }
        actions = {
            send_to_port_gtp_table;
            drop_gtp_table;
            NoAction;
        }
        default_action = NoAction;
        direct_counter = gtp_counter;
    }

    apply {
        if (headers.gtpv1.isValid()) {
            if (gtp_table.apply().miss) {
                nv_send_to_port(DEFAULT_VPORT);
            }
        }
        drop();
    }
}

NvDocaPipeline(
    packet_parser(),
    gtp_tunnel()
) main;
```

Example P4Runtime shell entries:

```
te_tunnel_1 = table_entry["gtp_tunnel.gtp_table"](action="gtp_tunnel.send_to_port_gtp_table")
te_tunnel_1.match["std_meta.ingress_port"] = "0"
te_tunnel_1.match["headers.gtpv1.teid"] = "0x000000001"
te_tunnel_1.action["port"] = "1"
te_tunnel_1.insert()
te_tunnel_2 = table_entry["gtp_tunnel.gtp_table"](action="gtp_tunnel.send_to_port_gtp_table")
te_tunnel_2.match["std_meta.ingress_port"] = "1"
te_tunnel_2.match["headers.gtpv1.teid"] = "0x000000001"
te_tunnel_2.action["port"] = "0"
te_tunnel_2.insert()
te_tunnel_3 = table_entry["gtp_tunnel.gtp_table"](action="gtp_tunnel.drop_gtp_table")
te_tunnel_3.match["std_meta.ingress_port"] = "4"
te_tunnel_3.match["headers.gtpv1.teid"] = "0x000000001"
te_tunnel_3.insert()

te_tunnel_1.read(lambda te_tunnel_1: print(te_tunnel_1))
te_tunnel_2.read(lambda te_tunnel_2: print(te_tunnel_2))
te_tunnel_3.read(lambda te_tunnel_3: print(te_tunnel_3))
```

Example Scapy packets:

```
from scapy.contrib.gtp import GTP_U_Header

tunnel = Ether(src=""3C:6D:66:11:11:11",dst="ff:ff:ff:ff:ff:ff")/IP(src="100.100.100.1", dst="100.100.100.81")/
UDP(sport=2152, dport=2152)
p_l3 = IP(dst='1.2.3.4', src='4.4.4.4') / TCP(sport=100,dport=100,seq=1001,flags='S')

gtp = tunnel / GTP_U_Header(teid=1)/ p_l3
gtp2 = tunnel / GTP_U_Header(teid=2)/ p_l3

sendp(gtp, iface="p1")
sendp(gtp, iface="enp3s0f0s0")
sendp(gtp, iface="enp3s0f0s3")
sendp(gtp2, iface="p1")
```

Expected results:

- Packet gtp is sent on port 0 and is received on port 1 with UDP src port of 1234
- Packet gtp is sent on port 1 and is received on port 0 with UDP src port of 1234
- Packet gtp is sent on port 4 and is dropped
- Packet gtp2 is sent on port 0 and is received on port 4 unmodified

See below for the complete DPL example.

```
/*
 * SPDX-FileCopyrightText: Copyright (c) 2024-2025 NVIDIA CORPORATION & AFFILIATES. All rights reserved.
 * SPDX-License-Identifier: LicenseRef-NvidiaProprietary
 *
 * NVIDIA CORPORATION, its affiliates and licensors retain all intellectual
 * property and proprietary rights in and to this material, related
 * documentation and any modifications thereto. Any use, reproduction,
 * disclosure or distribution of this material and related documentation
 * without an express license agreement from NVIDIA CORPORATION or
 * its affiliates is strictly prohibited.
 */

#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>

/*
 * This sample performs GTP matching using flex parser.
 * The GTP Tunnel Endpoint Identifier is used to selectively
 * forward the packets.
 */

const bit<32> WIRE_PORT = 32w00;
const bit<32> GTP_VPORT = 32w01;
const bit<32> DEFAULT_VPORT = 32w04;

struct metadata_t {
}

#define GTP_U_PORT 2152

header Gtp_v1_h {
    bit<3>        version;              /** For GTPv1, this has a value of 1. */
    bit           protocol_type;        /** GTP (value 1) from GTP' (value 0) */
    bit           reserved;
    bit           extension_header_flag; /** extension header optional field. */
    bit           seq_number_flag;      /** Sequence Number optional field */
    bit           n_pdu_number_flag;    /** N-PDU number optional field */
```

```
    bit<8>        message_type;          /** types of messages are defined in 3GPP TS 29.060 section
7.1 */
    bit<16>       message_length;        /** length of the payload in bytes */
    bit<32>       teid;                  /** Tunnel endpoint identifier */
    bit<16>       sequence_number;       /** optional */
    bit<8>        n_pdu_number;          /** optional */
    bit<8>        next_extension_hdr_type; /** optional if any of the E, S, or PN bits are on. The
field must be interpreted only if the E bit is on */
}

struct headers_t {
    NV_FIXED_HEADERS
    Gtp_v1_h   gtpv1;
}

parser packet_parser(packet_in packet, out headers_t headers) {
    NV_FIXED_PARSER(packet, headers)

    @nv_transition_from("nv_parse_udp", GTP_U_PORT)
    state parse_gtp
    {
        packet.extract(headers.gtpv1);
        transition accept;
    }
}

/**
 * This control admits GTP packets only if the tunnel ID matches
 *
 */
 control gtp_tunnel(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) gtp_counter;

    action drop() {
        nv_drop();
    }

    action send_to_port_gtp_table(nv_logical_port_t port) {
        gtp_counter.count();
        nv_set_l4_src_port(headers, 1234);
        nv_send_to_port(port);
    }

    action drop_gtp_table() {
        gtp_counter.count();
        nv_drop();
    }

    table gtp_table {
        key = {
            std_meta.ingress_port: exact;
            headers.gtpv1.teid: exact;
        }
        actions = {
            send_to_port_gtp_table;
            drop_gtp_table;
            NoAction;
        }
        default_action = NoAction;
        direct_counter = gtp_counter;
    }

    apply {
        if (headers.gtpv1.isValid()) {
            if (gtp_table.apply().miss) {
                nv_send_to_port(DEFAULT_VPORT);
            }
        }
        drop();
    }
}

NvDocaPipeline(
    packet_parser(),
    gtp_tunnel()
) main;
```

8.11.7.6.6 GTP Tunnel Encapsulation Example

The following example demonstrates GTPv1 encapsulation which must be declared as a custom header:

GTPv1 definition

```
header gtp_v1_h {
    bit<3>        version;               /** For GTPv1, this has a value of 1. */
    bit           protocol_type;         /** GTP (value 1) from GTP' (value 0) */
    bit           reserved;
    bit           extension_header_flag; /** extension header optional field. */
```

```
    bit           seq_number_flag;       /** Sequence Number optional field */
    bit           n_pdu_number_flag;     /** N-PDU number optional field */
    bit<8>        message_type;          /** types of messages are defined in 3GPP TS 29.060 section 7.1 */
    bit<16>       message_length;        /** length of the payload in bytes */
    bit<32>       teid;                  /** Tunnel endpoint identifier */
    bit<16>       sequence_number;       /** optional */
    bit<8>        n_pdu_number;          /** optional */
    bit<8>        next_extension_hdr_type; /** optional if any of the E, S, or PN bits are on. The field must be
interpreted only if the E bit is on */
}
```

The custom GTPv1 header can optionally be added to the parse graph (see Custom Parser States) is optional, and is only required if any of the custom header fields need to be read from or modified after encapsulation.

GTPv1 Parsing

```
struct headers_t {
    NV_FIXED_HEADERS
    gtp_v1_h   gtpv1;
}

parser packet_parser(packet_in packet, out headers_t headers) {
    NV_FIXED_PARSER(packet, headers)

    @nv_transition_from("nv_parse_udp", GTP_U_PORT)
    state parse_gtp
    {
        packet.extract(headers.gtpv1);
        transition nv_parse_inner_ipv4;
    }
}
```

The extern object NvTunnelTemplate<HEADER_TYPE> requires a struct type, for which the entire underlay header is defined as a struct:

Underlay headers

```
struct tunnel_headers_t {
    nv_ethernet_h ethernet;
    nv_ipv4_h     ipv4;
    nv_udp_h      udp;
    gtp_v1_h      gtpv1;
}
```

Finally, when declaring a NvTunnelTemplate object, the annotation **@nv_tunnel_fields** must be present. This annotation contains a key-value entry for each header in the specified struct type. Under each header, each header field must be specified in the order that the field appears in the header. For the GTPv1 example, a NvTunnelTemplate would be declared in the main control:

NvTunnelTemplate declaration

```
@nv_tunnel_fields(
    ethernet = {
        dst_addr = "variable",
        src_addr = "variable",
        ether_type = 0x0800
    },
    ipv4 = {
        version = 0x4,
        ihl = 0x5,
        diffserv = 0,
        ecn = "ignore",            // Cannot set
        total_len = "ignore",      // reparse will calculate, cannot set
        identification = "ignore", // Cannot set
        flags = 0,
        frag_offset = 0,
        ttl = 64,
        protocol = 17,
        hdr_checksum = "ignore",   // reparse will calculate, cannot set
        src_addr = "variable",
        dst_addr = "variable"
    },
    udp = {
        src_port = "ignore",       // reparse will set, cannot be set manually
```

```
        dst_port = "variable",
        length   = "ignore",       // reparse will calculate, cannot set
        checksum = "ignore"        // Cannot set
    },
    gtpv1 = {
        version = 1,
        protocol_type = 1,
        reserved = 0,
        extension_header_flag = 1,
        seq_number_flag = 1,
        n_pdu_number_flag = 1,
        message_type = 0xFF,    // T-PDU
        message_length = "variable",
        teid = "variable",
        sequence_number = 0,
        n_pdu_number = 0,
        next_extension_hdr_type = 0
    }
)
NvTunnelTemplate<tunnel_headers_t>() gtpv1Tunnel;
```

The type **tunnel_headers_t** determines the required structure of the annotation. Fields marked "variable" will be determined according to the arguments passed to **nv_set_l2tunnel_underlay** or **nv_set_l3tunnel_underlay**. Some fields will always be overwritten by reparse, such as IPv4 length, IPv4 checksum, UDP length, etc which must be calculated and cannot be set by the user. Such fields can be marked "ignore". Marking other fields "ignore", would set them to zero.

Finally, the instantiated NvTunnelTemplate object can be used in one of the custom tunnel externs. For example:

NvTunnelTemplate usage

```
action tunnel(nv_ipv4_addr_t src_addr) {
    nv_set_l3tunnel_underlay(headers, gtpv1Tunnel, {
        48w0x001A2B3C4D5E, // ethernet.dst_addr
        48w0x00F1E2D3C4B5, // ethernet.src_addr
        src_addr,          // ipv4.src_addr
        32w0xC07B7B01,     // ipv4.dst_addr
        16w0x868,          // udp.dst_port for GTP-U
        16w0x4,            // gtpv1.message_length
        32w0x1234567       // gtpv1.teid
    });
    nv_send_to_port(1);
}
```

The third argument to **nv_set_l3tunnel_underlay** is required to be a list expression where each element corresponds to a header field marked "variable" in the NvTunnelTemplate annotation. The order of the values in the list is the order in which the header fields appear in the packet.

The main table logic first checks the IP source, and then creates the GTP tunnel. Since header modifications are performed immediately, the reparsed packet can now be matched for TEID in the next table apply.

```
    table ip_as_key {
        key = {
            headers.ipv4.src_addr : exact;
        }
        actions = {
            NoAction;
            drop;
            forward;
            tunnel;
        }
        size = 128;
        default_action = NoAction;
    }

    table teid_as_key {
        key = {
            headers.gtpv1.teid : exact;
        }
        actions = {
            NoAction;
            drop;
            forward;
        }
        size = 128;
        default_action = forward(3);
    }
```

```
apply {
    ip_as_key.apply();
    if (headers.gtpv1.isValid()) {
        teid_as_key.apply();
    }
    forward(3);
}
```

See below for the complete DPL example.

```
/*
 * SPDX-FileCopyrightText: Copyright (c) 2025 NVIDIA CORPORATION & AFFILIATES. All rights reserved.
 * SPDX-License-Identifier: LicenseRef-NvidiaProprietary
 *
 * NVIDIA CORPORATION, its affiliates and licensors retain all intellectual
 * property and proprietary rights in and to this material, related
 * documentation and any modifications thereto. Any use, reproduction,
 * disclosure or distribution of this material and related documentation
 * without an express license agreement from NVIDIA CORPORATION or
 * its affiliates is strictly prohibited.
 */

#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>

#define GTP_U_PORT 2152

header gtp_v1_h {
    bit<3>        version;               /** For GTPv1, this has a value of 1. */
    bit           protocol_type;         /** GTP (value 1) from GTP' (value 0) */
    bit           reserved;
    bit           extension_header_flag; /** extension header optional field. */
    bit           seq_number_flag;       /** Sequence Number optional field */
    bit           n_pdu_number_flag;     /** N-PDU number optional field */
    bit<8>        message_type;          /** types of messages are defined in 3GPP TS 29.060 section 7.1 */
    bit<16>       message_length;        /** length of the payload in bytes */
    bit<32>       teid;                  /** Tunnel endpoint identifier */
    bit<16>       sequence_number;       /** optional */
    bit<8>        n_pdu_number;          /** optional */
    bit<8>        next_extension_hdr_type; /** optional if any of the E, S, or PN bits are on. The field must be interpreted only if the E bit is on */
}

struct headers_t {
    NV_FIXED_HEADERS
    gtp_v1_h   gtpv1;
}

parser packet_parser(packet_in packet, out headers_t headers) {
    NV_FIXED_PARSER(packet, headers)

    @nv_transition_from("nv_parse_udp", GTP_U_PORT)
    state parse_gtp
    {
        packet.extract(headers.gtpv1);
        transition nv_parse_inner_ipv4;
    }
}

struct tunnel_headers_t {
    nv_ethernet_h ethernet;
    nv_ipv4_h     ipv4;
    nv_udp_h      udp;
    gtp_v1_h      gtpv1;
}

control c(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    @nv_header_data_fields(
        ethernet = {
            dst_addr = "variable",
            src_addr = "variable",
            ether_type = 0x0800
        },
        ipv4 = {
            version = 0x4,
            ihl = 0x5,
            diffserv = 0,
            ecn = "ignore",            // cannot set
            total_len = "ignore",      // HW will calculate, cannot set
            identification = "ignore", // cannot set
            flags = 0,
            frag_offset = 0,
            ttl = 64,
            protocol = 17,
            hdr_checksum = "ignore",   // HW will calculate, cannot set
            src_addr = "variable",
            dst_addr = "variable"
        },
        udp = {
            src_port = "ignore",       // HW will calculate entropy, cannot set
            dst_port = GTP_U_PORT,
            length   = "ignore",       // HW will calculate, cannot set
```

```
            checksum = "ignore"        // cannot set
        },
        gtpv1 = {
            version = 1,
            protocol_type = 1,
            reserved = 0,
            extension_header_flag = 1,
            seq_number_flag = 1,
            n_pdu_number_flag = 1,
            message_type = 0xFF,
            message_length = "variable",
            teid = "variable",
            sequence_number = 123,
            n_pdu_number = 255,
            next_extension_hdr_type = 0
        }
    )
    NvHeaderDataTemplate<tunnel_headers_t>() gtpv1Tunnel;

    action drop() {
        nv_drop();
    }

    action forward(nv_logical_port_t port) {
        nv_send_to_port(port);
    }

    action tunnel(nv_ipv4_addr_t dst_addr, bit<32> teid) {
        nv_set_l3tunnel_underlay(headers, gtpv1Tunnel, {
            48w0x01005e000001, 48w0x001122334455, 32w0x04040404, dst_addr,
            16w4, /* 4 bytes of optional GTP-U payload */
            teid
        });
    }

    table ip_as_key {
        key = {
            headers.ipv4.src_addr : exact;
        }
        actions = {
            NoAction;
            drop;
            forward;
            tunnel;
        }
        size = 128;
        default_action = NoAction;
    }

    table teid_as_key {
        key = {
            headers.gtpv1.teid : exact;
        }
        actions = {
            NoAction;
            drop;
            forward;
        }
        size = 128;
        default_action = forward(3);
    }

    apply {
        ip_as_key.apply();
        if (headers.gtpv1.isValid()) {
            teid_as_key.apply();
        }
        forward(3);
    }
}

NvDocaPipeline(
    packet_parser(),
    c()
) main;
```

8.11.7.6.7 GTP Full Tunnel Application

This DPL program implements a complete GTP tunnel handler on Bluefield, providing parsing, encapsulation, decapsulation, and flexible forwarding for mobile network traffic.
Incoming packets are processed to:

- extract GTP tunnel information
- encapsulate plain IP packets into GTP tunnels
- decapsulate GTP-tunneled packets to deliver the inner payload

The program supports both user equipment (UE)-to-core uplink and core-to-UE downlink paths, as well as plain IP and GTP transit forwarding. Each operation is hardware-accelerated and leverages programmable match-action tables and tunnel header templates for optimal performance.

8.11.7.6.7.1 Sample Code

Custom Header and Metadata Structures

- GTP Header (`gtp_v1_h`): Defines GTPv1 protocol fields, enabling extraction and manipulation of GTP tunnel information.
- Custom Metadata (`metadata_t`): Stores flags and tunnel parameters to guide encapsulation or decapsulation during pipeline execution.

Parser

- Flexible Parsing: Leverages the fixed parser for standard headers, and adds a GTP-specific parsing state that triggers when the UDP destination port is GTP-U (2152). When detected, it extracts the GTP header and transitions to parsing the inner IP payload.

Tunnel Header Template

- Header Composition (`tunnel_headers_t`): Specifies the exact order and content of Ethernet, IPv4, UDP, and GTP headers for encapsulation, using variables for fields filled at runtime (e.g., TEID, address fields).

Main Control Block – Actions

- Drop Action: Immediately drops non-conforming or unwanted packets.
- Encapsulation (`encapsulate_gtp`): Encapsulates an IP packet within GTP, UDP, IPv4, and Ethernet outer headers, using the DPL extern for L3 tunnel offload.
- Decapsulation (`decapsulate_gtp`): Removes the GTP tunnel header and the outer headers, then rewrites the Ethernet header for the inner packet.
- Forwarding: Directs packets to their designated output port.

Main Control Block – Tables

- Uplink Encapsulation Table: Matches on input port and source IP to select the correct GTP tunnel for encapsulation (UE to core network).
- Downlink Decapsulation Table: Matches on input port and TEID, applying decapsulation to GTP packets for delivery to the user side (core network to UE).
- GTP Forwarding Table: Forwards GTP packets by TEID for nodes that act as GTP relays without terminating or modifying the tunnel.
- Plain IP Forwarding Table: Routes non-GTP IP traffic using destination address longest prefix match.

Packet Processing Logic (`apply` block)

- Non-IPv4 Packets: Dropped immediately, ensuring pipeline only processes valid IPv4.
- GTP Packets: Processed through downlink decapsulation and, if appropriate, GTP forwarding tables.
- Non-GTP Packets: Subject to uplink encapsulation (for new tunnel flows) or regular IP forwarding.

See below for the complete DPL example.

```
/*
 * SPDX-FileCopyrightText: Copyright (c) 2025 NVIDIA CORPORATION & AFFILIATES. All rights reserved.
 * SPDX-License-Identifier: LicenseRef-NvidiaProprietary
 *
 * NVIDIA CORPORATION, its affiliates and licensors retain all intellectual
 * property and proprietary rights in and to this material, related
 * documentation and any modifications thereto. Any use, reproduction,
 * disclosure or distribution of this material and related documentation
 * without an express license agreement from NVIDIA CORPORATION or
 * its affiliates is strictly prohibited.
 *
 * Complete GTP Tunnel Handler - Parse, Encapsulate, and Decapsulate
 *
 * This P4 program provides comprehensive GTP tunnel handling:
 * 1. Parse incoming GTP packets and extract tunnel information
 * 2. Encapsulate plain IP packets into GTP tunnels
 * 3. Decapsulate GTP packets to extract inner payload
 * 4. Forward packets based on TEID or IP addresses
 */

#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>

// Constants
#define GTP_U_PORT 2152
#define GTP_VERSION 1
#define GTP_PROTOCOL_TYPE 1
#define GTP_MESSAGE_TYPE_GPDU 0xFF

// Port definitions - must align with DPL RTD port configuration file
const bit<32> WIRE_PORT = 32w0;
const bit<32> UPLINK_PORT = 32w1;      // Port for uplink traffic (to core network)
const bit<32> DOWNLINK_PORT = 32w2;    // Port for downlink traffic (to RAN)
const bit<32> DEFAULT_PORT = 32w3;     // Default forwarding port
const bit<32> DROP_PORT = 32w4;        // Drop port

// GTP v1 Header Definition
header gtp_v1_h {
    bit<3>        version;
    bit           protocol_type;
    bit           reserved;
    bit           extension_header_flag;
    bit           seq_number_flag;
    bit           n_pdu_number_flag;
    bit<8>        message_type;
    bit<16>       message_length;
    bit<32>       teid;
    bit<16>       sequence_number;
    bit<8>        n_pdu_number;
    bit<8>        next_extension_hdr_type;
}

// Metadata for GTP processing
struct metadata_t {
    bit<1>  needs_encap;        // Flag: packet needs GTP encapsulation
    bit<1>  needs_decap;        // Flag: packet needs GTP decapsulation
    bit<32> tunnel_teid;        // TEID for encapsulation
    bit<32> tunnel_dst_ip;      // Destination IP for tunnel
    bit<48> tunnel_dst_mac;     // Destination MAC for tunnel
    bit<48> tunnel_src_mac;     // Source MAC for tunnel
    bit<32> tunnel_src_ip;      // Source IP for tunnel
}

// Complete header stack
struct headers_t {
    NV_FIXED_HEADERS
    gtp_v1_h   gtpv1;
}

// Parser - handles both outer and inner packets
parser packet_parser(packet_in packet, out headers_t headers) {
    NV_FIXED_PARSER(packet, headers)

    @nv_transition_from("nv_parse_udp", GTP_U_PORT)
    state parse_gtp {
        packet.extract(headers.gtpv1);
        transition nv_parse_inner_ipv4;  // Continue parsing inner IP
    }
}

// Tunnel header template for encapsulation
struct tunnel_headers_t {
    nv_ethernet_h ethernet;
    nv_ipv4_h     ipv4;
    nv_udp_h      udp;
    gtp_v1_h      gtpv1;
}

// Main control block
control gtp_processor(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
```

```
    // Counters for statistics
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) encap_counter;
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) decap_counter;
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) gtp_forward_counter;
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) ip_forward_counter;

    // GTP Tunnel template for encapsulation
    @nv_header_data_fields(
        ethernet = {
            dst_addr = "variable",
            src_addr = "variable",
            ether_type = 0x0800
        },
        ipv4 = {
            version = 0x4,
            ihl = 0x5,
            diffserv = 0,
            ecn = "ignore",
            total_len = "ignore",
            identification = "ignore",
            flags = 0,
            frag_offset = 0,
            ttl = 64,
            protocol = 17,
            hdr_checksum = "ignore",
            src_addr = "variable",
            dst_addr = "variable"
        },
        udp = {
            src_port = "ignore",
            dst_port = GTP_U_PORT,
            length = "ignore",
            checksum = "ignore"
        },
        gtpv1 = {
            version = GTP_VERSION,
            protocol_type = GTP_PROTOCOL_TYPE,
            reserved = 0,
            extension_header_flag = 1,
            seq_number_flag = 1,
            n_pdu_number_flag = 1,
            message_type = GTP_MESSAGE_TYPE_GPDU,
            message_length = "ignore",
            teid = "variable",
            sequence_number = 0,
            n_pdu_number = 0,
            next_extension_hdr_type = 0
        }
    )
    NvHeaderDataTemplate<tunnel_headers_t>() gtp_tunnel_template;

    // ========== ACTIONS ==========

    action drop() {
        nv_drop();
    }

    action forward_to_port(nv_logical_port_t port) {
        nv_send_to_port(port);
    }

    // Encapsulation action - wrap packet in GTP tunnel
    action encapsulate_gtp(
        bit<48> dst_mac,
        bit<48> src_mac,
        bit<32> src_ip,
        bit<32> dst_ip,
        bit<32> teid,
        nv_logical_port_t port
    ) {
        encap_counter.count();
        // Parameters must match the "variable" fields in template order:
        // ethernet.dst_addr, ethernet.src_addr, ipv4.src_addr, ipv4.dst_addr, teid
        nv_set_l3tunnel_underlay(headers, gtp_tunnel_template, {
            dst_mac,      // Ethernet dst_addr
            src_mac,      // Ethernet src_addr
            src_ip,       // IPv4 src_addr
            dst_ip,       // IPv4 dst_addr
            teid          // GTP teid
        });
        nv_send_to_port(port);
    }

    // Decapsulation action - remove GTP tunnel headers and forward inner packet
    action decapsulate_gtp(
        bit<48> dst_mac,
        bit<48> src_mac,
        nv_logical_port_t port
    ) {
        decap_counter.count();
        // Strip outer tunnel headers (Ethernet/IP/UDP/GTP) and rewrite L2 header
        // 0x0800 = IPv4 EtherType, no VLAN tag
        nv_l3_decap(
            headers = headers,
            dst_mac = dst_mac,
            src_mac = src_mac,
            l3_ether_type = 16w0x0800
        );
        // Forward the decapsulated inner packet
        nv_send_to_port(port);
    }

    // ========== TABLES ==========

    // Table 1: Uplink encapsulation (UE -> Core Network)
```

```
    // Match on source IP address to determine which GTP tunnel to use
    table uplink_encap_table {
        key = {
            std_meta.ingress_port: exact;
            headers.ipv4.src_addr: exact;
        }
        actions = {
            encapsulate_gtp;
            forward_to_port;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = NoAction;
        direct_counter = encap_counter;
    }

    // Table 2: Downlink decapsulation and forwarding (Core Network -> UE)
    // Match on GTP TEID to determine forwarding
    table downlink_decap_table {
        key = {
            std_meta.ingress_port: exact;
            headers.gtpv1.teid: exact;
        }
        actions = {
            decapsulate_gtp;
            forward_to_port;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = NoAction;
        direct_counter = decap_counter;
    }

    // Table 3: GTP tunnel forwarding (no decap, just forward GTP packets)
    // For transit nodes that don't terminate the tunnel
    table gtp_forward_table {
        key = {
            std_meta.ingress_port: exact;
            headers.gtpv1.teid: exact;
        }
        actions = {
            forward_to_port;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = NoAction;
        direct_counter = gtp_forward_counter;
    }

    // Table 4: Plain IP forwarding (for non-GTP traffic)
    table ip_forward_table {
        key = {
            headers.ipv4.dst_addr: lpm;
        }
        actions = {
            forward_to_port;
            drop;
            NoAction;
        }
        size = 512;
        default_action = NoAction;
        direct_counter = ip_forward_counter;
    }

    // ========== APPLY LOGIC ==========

    apply {
        // Path 1: Invalid IPv4 - drop
        if (!headers.ipv4.isValid()) {
            drop();
        }
        // Path 2: GTP packet received (potential decapsulation or forwarding)
        else if (headers.gtpv1.isValid()) {
            downlink_decap_table.apply();
            gtp_forward_table.apply();
        }
        // Path 3: Non-GTP packet (potential encapsulation or IP forwarding)
        else {
            uplink_encap_table.apply();
            ip_forward_table.apply();
        }
    }
}

// Pipeline instantiation
NvDocaPipeline(
    packet_parser(),
    gtp_processor()
) main;
```

8.11.7.6.8 Geneve TLV Parsing Example

This example demonstrates how to add a custom protocol header that supports optional Type-Length-Value (TLV) fields. In the P4-16 specification, there is no first class support in the parser primitives, and the user needs to manually.

Geneve is short for Generic Network Virtualization Encapsulation, a common L2 tunnel header. The Geneve header format is:

```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Ver|  Opt Len  |O|C|    Rsvd.  |          Protocol Type        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Virtual Network Identifier (VNI)       |    Reserved   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                    Variable-Length Options                    ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

The Geneve option header is:

 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |          Option Class         |       Type    |R|R|R|  Length |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |               ~ Variable-Length Option Data ~                 |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

8.11.7.6.8.1 Sample Code

First we define the Geneve header, per RFC 8926.

```
header geneve_t {
    bit<2> ver;
    bit<6> opt_len;
    bit<1> o;
    bit<1> c;
    bit<6> reserved1;
    bit<16> protocol_type;
    bit<24> vni;
    bit<8> reserved2;
};
```

Next we define a struct that contains the base Geneve Option header, shared by all option types.

```
// Struct so it can be a field within other headers
struct geneve_option_t {
    bit<24> option_class_type;
    bit<3> reserved;
    bit<5> length;
};
```

Lastly, depending on the option class and type, the variable length option data can be defined. In this example, we model the Geneve options used by the P4 Inband Telemetry (INT) specification.

```
// Intermediate nodes (INT Transit Hops) must process this type of INT Header.
header geneve_option_int_md_t {
    geneve_option_t base;
    bit<4> version;
    bit<1> discard;
    bit<1> exceeded_max_hops;
    bit<1> mtu_exceeded;
    bit<12> reserved;
    bit<5> hop_ml;
    bit<8> remaining_hop_count;
    bit<16> instruction_bitmap;
    bit<16> ds_id;
    bit<16> ds_instruction;
    bit<16> ds_flags;
};
// Destination headers can be used to enable Edge-to-Edge communication between
// the INT Source and INT Sink.
header geneve_option_int_destination_t {
    geneve_option_t base;
    // Destination headers must only be consumed by the INT Sink
};

// Intermediate nodes (INT Transit Hops) must process this type of INT Header
// and generate reports to the monitoring system as instructed.
header geneve_option_int_mx_t {
    geneve_option_t base;
    bit<4> version;
    bit<1> discard;
    bit<27> reserved1;
    bit<16> instruction_bitmap;
    bit<16> ds_id;
    bit<16> ds_instruction;
    bit<16> ds_flags;
```

```
};
```

All the above headers and structs must be added to the application's headers struct, along with the NV_FIXED_HEADERS.

```
struct app_headers {
    NV_FIXED_HEADERS

    geneve_t custom_geneve;
    // geneve_option_t is not explicitly used by the P4 program, so a reference
    // placed in the headers the type is not optimized out. The NvOptionParser
    // instantiation references this type it by string name.
    geneve_option_t geneve_opt;
    geneve_option_int_md_t geneve_opt_int_md;
    geneve_option_int_destination_t geneve_opt_int_destination;
    geneve_option_int_mx_t geneve_opt_int_mx;
};
```

The last step for the parser is to define the parser state that will perform Geneve parsing. An extern object, NvOptionsParser, is used to further parse into the Geneve options. In this example, the TLV "type" is a combination of Option class and type:

- Class 0x103, type 1 (INT-MD)
- Class 0x103, type 2 (Destination-type)
- Class 0x103, type 3 (INT-MX)

```
parser geneve_parser(
    packet_in packet,
    out app_headers headers
) {
    NvOptionParser<bit<24>, _>(
        "opt_len",                  // options_length_field
        2,                          // options_length_shift
        0,                          // options_length_add
        "geneve_option_t",          // option_layout_header_type
        "length",                   // option_length_field
        0,                          // option_length_shift
        4,                          // option_length_add
        "option_class_type",        // option_type_field
        (list<tuple<bit<24>, _>>){  // options
            {24w0x010301, "headers.geneve_opt_int_md"},
            {24w0x010302, "headers.geneve_opt_int_destination"},
            {24w0x010303, "headers.geneve_opt_int_mx"}
        }
    ) geneveOptions;

    NV_FIXED_PARSER(packet, headers)

    @nv_transition_from("nv_parse_udp", GENEVE_PORT)
    state parse_geneve {
        // Fixed geneve is only an example; "base" header must be flex.
        packet.extract(headers.custom_geneve);
        geneveOptions.parseOptions(packet, headers);
        transition select(headers.custom_geneve.protocol_type) {
            NV_TYPE_IPV4 : nv_parse_inner_ipv4;
            NV_TYPE_IPV6 : nv_parse_inner_ipv6;
            NV_TYPE_MAC : nv_parse_inner_ethernet;
            default : accept;
        }
    }
}
```

The control below shows how a match action table can be configured to now match on a specific INT domain, and perform per ID forwarding.

```
control int_over_geneve(
    inout app_headers headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    action forward(bit<32> port) {
        nv_send_to_port(port);
    }

    action drop() {
        nv_drop();
    }

    table geneve_option_int_md_table {
        key = {
            headers.geneve_opt_int_md.ds_id : exact;
        }
```

```
        actions = {
            forward;
            NoAction;
        }
        default_action = NoAction();
        const entries = {
            (16w0x100) : forward(1);
        }
    }

    table geneve_option_int_destination_table {
        key = {
            headers.custom_geneve.vni : exact;
        }
        actions = {
            forward;
            NoAction;
        }
        default_action = NoAction();
        const entries = {
            (24w0x200) : forward(2);
        }
    }

    table geneve_option_int_mx_table {
        key = {
            headers.geneve_opt_int_mx.ds_id : exact;
        }
        actions = {
            forward;
            NoAction;
        }
        default_action = NoAction();
        const entries = {
            (16w0x300) : forward(3);
        }
    }

    apply {
        if (headers.custom_geneve.isValid()) {
            geneve_option_int_md_table.apply();
            geneve_option_int_destination_table.apply();
            geneve_option_int_mx_table.apply();
        }
        drop();
    }
}
```

See below for the complete DPL example.

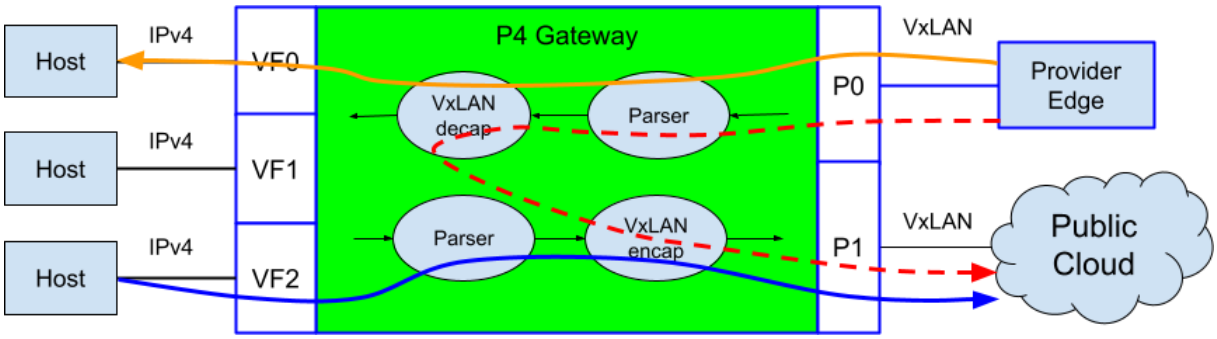
```
/*
 * SPDX-FileCopyrightText: Copyright (c) 2024-2025 NVIDIA CORPORATION & AFFILIATES. All rights reserved.
 * SPDX-License-Identifier: LicenseRef-NvidiaProprietary
 *
 * NVIDIA CORPORATION, its affiliates and licensors retain all intellectual
 * property and proprietary rights in and to this material, related
 * documentation and any modifications thereto. Any use, reproduction,
 * disclosure or distribution of this material and related documentation
 * without an express license agreement from NVIDIA CORPORATION or
 * its affiliates is strictly prohibited.
 */

#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>

#define GENEVE_PORT       6082

header geneve_t {
    bit<2> ver;
    bit<6> opt_len;
    bit<1> o;
    bit<1> c;
    bit<6> reserved1;
    bit<16> protocol_type;
    bit<24> vni;
    bit<8> reserved2;
};

// Struct so it can be a field within other headers
struct geneve_option_t {
    bit<24> option_class_type;
    bit<3> reserved;
    bit<5> length;
};

// Intermediate nodes (INT Transit Hops) must process this type of INT Header.
header geneve_option_int_md_t {
    geneve_option_t base;
    bit<4> version;
    bit<1> discard;
    bit<1> exceeded_max_hops;
    bit<1> mtu_exceeded;
    bit<12> reserved;
    bit<5> hop_ml;
    bit<8> remaining_hop_count;
    bit<16> instruction_bitmap;
    bit<16> ds_id;
    bit<16> ds_instruction;
```

```
    bit<16> ds_flags;
};

// Destination headers can be used to enable Edge-to-Edge communication between
// the INT Source and INT Sink.
header geneve_option_int_destination_t {
    geneve_option_t base;
    // Destination headers must only be consumed by the INT Sink
};

// Intermediate nodes (INT Transit Hops) must process this type of INT Header
// and generate reports to the monitoring system as instructed.
header geneve_option_int_mx_t {
    geneve_option_t base;
    bit<4> version;
    bit<1> discard;
    bit<27> reserved1;
    bit<16> instruction_bitmap;
    bit<16> ds_id;
    bit<16> ds_instruction;
    bit<16> ds_flags;
};

struct app_headers {
    NV_FIXED_HEADERS

    geneve_t custom_geneve;
    // geneve_option_t is not explicitly used by the DPL program, so a reference
    // placed in the headers the type is not optimized out. The NvOptionParser
    // instantiation references this type it by string name.
    geneve_option_t geneve_opt;
    geneve_option_int_md_t geneve_opt_int_md;
    geneve_option_int_destination_t geneve_opt_int_destination;
    geneve_option_int_mx_t geneve_opt_int_mx;
};

parser geneve_parser(
    packet_in packet,
    out app_headers headers
) {
    NvOptionParser<bit<24>, _>(
        "opt_len",                  // options_length_field
        2,                          // options_length_shift
        0,                          // options_length_add
        "geneve_option_t",          // option_layout_header_type
        "length",                   // option_length_field
        2,                          // option_length_shift
        4,                          // option_length_add
        "option_class_type",        // option_type_field
        (list<tuple<bit<24>, _>>){  // options
            {24w0x010301, "headers.geneve_opt_int_md"},
            {24w0x010302, "headers.geneve_opt_int_destination"},
            {24w0x010303, "headers.geneve_opt_int_mx"}
        }
    ) geneveOptions;

    NV_FIXED_PARSER(packet, headers)

    @nv_transition_from("nv_parse_udp", GENEVE_PORT)
    state parse_geneve {
        // Fixed geneve is only an example; "base" header must be flex.
        packet.extract(headers.custom_geneve);
        geneveOptions.parseOptions(packet, headers);
        transition select(headers.custom_geneve.protocol_type) {
            NV_TYPE_IPV4 : nv_parse_inner_ipv4;
            NV_TYPE_IPV6 : nv_parse_inner_ipv6;
            // flex transition to nv_parse_inner_ethernet not supported by current firmware
            // NV_TYPE_MAC : nv_parse_inner_ethernet;
            default : accept;
        }
    }
}

control int_over_geneve(
    inout app_headers headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    action forward(bit<32> port) {
        nv_send_to_port(port);
    }

    action drop() {
        nv_drop();
    }

    table geneve_option_int_md_table {
        key = {
            headers.geneve_opt_int_md.ds_id : exact;
        }
        actions = {
            forward;
            NoAction;
        }
        default_action = NoAction();
        const entries = {
            (16w0x100) : forward(1);
        }
    }

    table geneve_option_int_destination_table {
        key = {
            headers.custom_geneve.vni : exact;
        }
```

```
        actions = {
            forward;
            NoAction;
        }
        default_action = NoAction();
        const entries = {
            (24w0x200) : forward(2);
        }
    }

    table geneve_option_int_mx_table {
        key = {
            headers.geneve_opt_int_mx.ds_id : exact;
        }
        actions = {
            forward;
            NoAction;
        }
        default_action = NoAction();
        const entries = {
            (16w0x300) : forward(3);
        }
    }

    apply {
        if (headers.custom_geneve.isValid()) {
            geneve_option_int_md_table.apply();
            geneve_option_int_destination_table.apply();
            geneve_option_int_mx_table.apply();
        }
        drop();
    }
}

NvDocaPipeline(
    geneve_parser(),
    int_over_geneve()
) main;
```

8.11.7.6.9 VXLAN Tunnel Gateway Example

This example illustrates how to write a basic VXLAN tunnel gateway using the DOCA target architecture. A tunnel gateway allows programmatic control over how VXLAN traffic can be "stitched" packets across tenant domains. In this example, end point traffic destined to local bare metal hosts can be decapsulated and forwarded to a VF, while gateway traffic can be decapsulated, re-encapsulated and sent back to the wire. For example, this program can be easily extended to be a gateway connecting legacy NVGRE networks to a VXLAN-GPE network.



8.11.7.6.9.1 Sample Code

This example users the native parser and 2 tables, of size 32K each. The wire port is configured with P4 port ID 0. A single bit in the user metadata structure is used to keep the decapsulation state.

```
/*
* Table sizes.
*/
const bit<32> DECAP_TABLE_SIZE = 32768;
const bit<32> ENCAP_TABLE_SIZE = 32768;
```

```
/* The directionality is based on network to host
 * The user will configure the DPL port IDs in the DPL RT configuration
 */
const bit<32> WIRE_PORT = 32w0;

struct headers_t {
    NV_FIXED_HEADERS
}

parser packet_parser(packet_in packet, out headers_t headers) {
    NV_FIXED_PARSER(packet, headers)
 }
```

The encapsulation control has a single table, matching on IPv4 destination address. If an entry matches, the packet is VXLAN encapsulated and forwarded to the specified port. If the packet does not hit any entry, then the packet is dropped. It is simple to add more complex policy rules such as a 5 tuple ACL.

```
/*
* This control performs the overlay policy including L2 encap with VXLAN
*/
control overlay_encap(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) encap_counter;

    action deny() {
        nv_drop();
    }

    action to_port(nv_logical_port_t port) {
        nv_send_to_port(port);
    }

    action vxlan_v4_encap(nv_mac_addr_t underlay_src_mac, nv_mac_addr_t underlay_dst_mac,
                nv_ipv4_addr_t underlay_sip, nv_ipv4_addr_t underlay_dip, bit<24> vni, nv_logical_port_t port) {
        nv_set_vxlan_v4_underlay(headers, underlay_dst_mac, underlay_src_mac, underlay_sip, underlay_dip, vni);
        encap_counter.count();
        nv_send_to_port(port);
    }

    table encap_v4_table {
        key = {
            headers.ipv4.dst_addr : exact;
        }
        actions = {
            vxlan_v4_encap;
            to_port;
            deny;
        }
        size = ENCAP_TABLE_SIZE;
        default_action = deny;
        direct_counter = encap_counter;
    }

    apply {
       if (headers.ipv4.isValid()) {
           encap_v4_table.apply();
       }
    }
}
```

The decapsulation control simply checks if the packet is VXLAN, and decapsulates it. From there the packet can be sent directly to a port, or hair-pinned back to the wire.

```
/*
* This control is for packets from wire to host (RX)
* and includes policy for L2 decap
*/
control underlay_decap(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) decap_counter;

    action deny() {
        nv_drop();
    }

    action decap() {
        decap_counter.count();
        nv_l2_decap(headers);
    }

    action to_port(nv_logical_port_t port) {
        nv_send_to_port(port);
```

```
    }

    action decap_to_port(nv_logical_port_t port) {
        decap_counter.count();
        nv_l2_decap(headers);
        nv_send_to_port(port);
    }

    table decap_v4_table {
        key = {
            headers.vxlan.vni : exact;
        }
        actions = {
            decap;
            to_port;
            decap_to_port;
            deny;
            NoAction;
        }
        size = DECAP_TABLE_SIZE;
        direct_counter = decap_counter;
        default_action = NoAction;
    }

    apply {
        if (headers.vxlan.isValid()) {
            decap_v4_table.apply();
        }
    }
}
```

The main control checks the ingress port to determine if the packet is Network to Host, or Host to Network. Depending on the direction, it applies the decap_flow control, or performs an overlay encapsulation.

```
control gateway(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    overlay_encap() over;
    underlay_decap() under;

    /* user should add entries that correspond to the wire ports
     * A hit means this is an RX packet, miss means a TX packet
    */
    table direction_table {
        key = {
            std_meta.ingress_port : exact;
        }
        actions = {
            NoAction;
        }
        default_action = NoAction;
        const entries = {
            (WIRE_PORT) : NoAction();
        }
    }

    apply {
        if (direction_table.apply().hit) {
            under.apply(headers, std_meta, user_meta, pkt_out_meta);
        }
        else {
            over.apply(headers, std_meta, user_meta, pkt_out_meta);
        }
    }
}
```

See below for the complete DPL example.

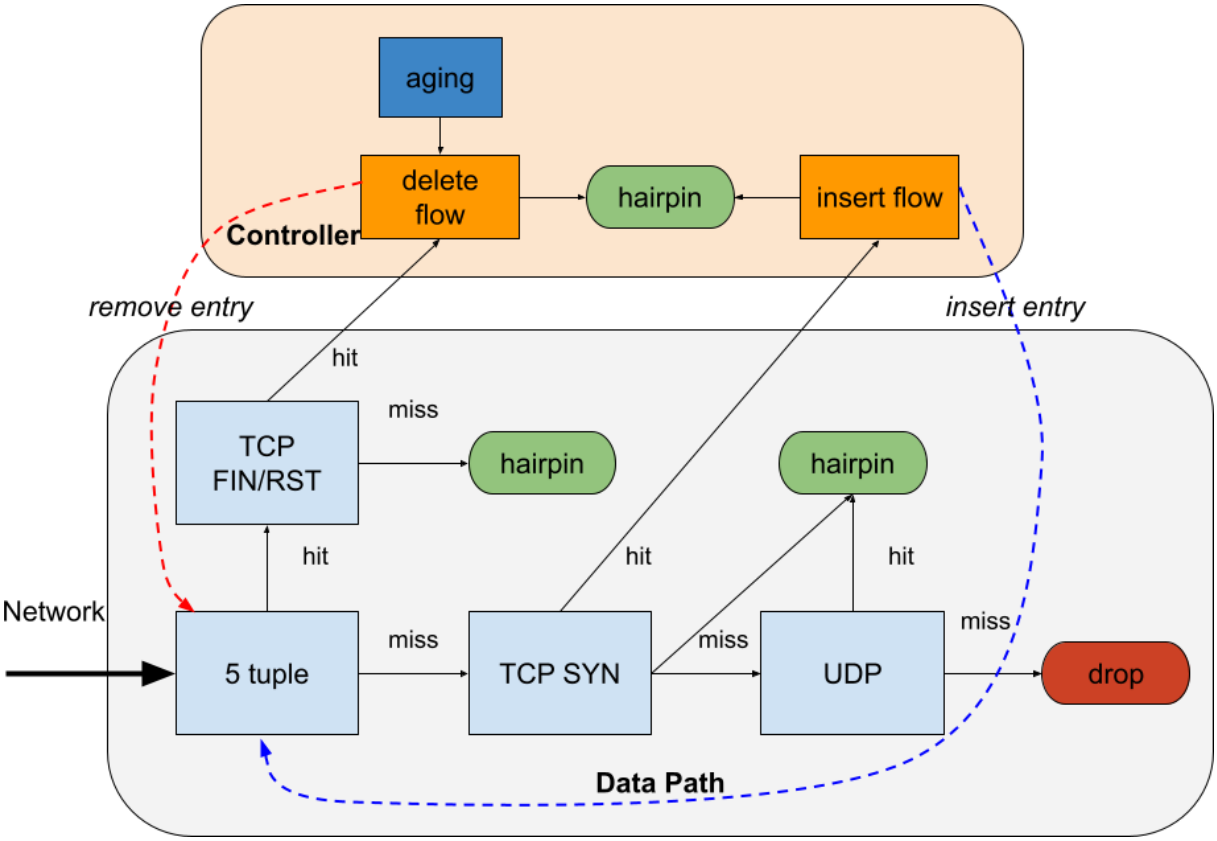
```
 1  /*
 2   * SPDX-FileCopyrightText: Copyright (c) 2024-2025 NVIDIA CORPORATION & AFFILIATES. All rights
    reserved.
 3   * SPDX-License-Identifier: LicenseRef-NvidiaProprietary
 4   *
 5   * NVIDIA CORPORATION, its affiliates and licensors retain all intellectual
 6   * property and proprietary rights in and to this material, related
 7   * documentation and any modifications thereto. Any use, reproduction,
 8   * disclosure or distribution of this material and related documentation
 9   * without an express license agreement from NVIDIA CORPORATION or
10   * its affiliates is strictly prohibited.
11   */
12
13  #include <doca_model.p4>
14  #include <doca_headers.p4>
15  #include <doca_externs.p4>
16  #include <doca_parser.p4>
17
18  /*
19   * VxLAN tunnel gateway
20   * This application allows the user have a customized tunnel gateway, that can stitch VxLAN
21   * packets across tenant domains. End point traffic destined to local bare metal hosts can
```

```
 * be decapsulated and forwarded, while gateway traffic can be decapsulated, then
 * encapsulated back to the wire. This program can be easily extended to be a gateway across
 * different tunnel types as well.
 */

/*
* Table sizes.
*/
const bit<32> DECAP_TABLE_SIZE = 32768;
const bit<32> ENCAP_TABLE_SIZE = 32768;

/* The directionality is based on network to host
 * The user will configure the DPL port IDs in the DPL RT configuration
 */
const bit<32> WIRE_PORT = 32w0;

struct headers_t {
    NV_FIXED_HEADERS
}

parser packet_parser(packet_in packet, out headers_t headers) {
    NV_FIXED_PARSER(packet, headers)
 }

/**
* This control performs the overlay policy including L2 encap with VxLAN
*/
control overlay_encap(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) encap_counter;

    action deny() {
        nv_drop();
    }

    action to_port(nv_logical_port_t port) {
        nv_send_to_port(port);
    }

    action vxlan_v4_encap(nv_mac_addr_t underlay_src_mac, nv_mac_addr_t underlay_dst_mac,
                          nv_ipv4_addr_t underlay_sip, nv_ipv4_addr_t underlay_dip, bit<24> vni,
nv_logical_port_t port) {
        nv_set_vxlan_v4_underlay(headers, underlay_dst_mac, underlay_src_mac, underlay_sip,
underlay_dip, vni);
        encap_counter.count();
        nv_send_to_port(port);
    }

    table encap_v4_table {
        key = {
            headers.ipv4.dst_addr : exact;
        }
        actions = {
            vxlan_v4_encap;
            to_port;
            deny;
        }
        size = ENCAP_TABLE_SIZE;
        default_action = deny;
        direct_counter = encap_counter;
    }

    apply {
       if (headers.ipv4.isValid()) {
           encap_v4_table.apply();
       }
    }
}


/**
* This control is for packets from wire to host (RX)
* and includes policy for L2 decap
*/
control underlay_decap(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) decap_counter;

    action deny() {
        nv_drop();
    }

    action decap() {
        decap_counter.count();
        nv_l2_decap(headers);
    }

    action to_port(nv_logical_port_t port) {
        nv_send_to_port(port);
    }

    action decap_to_port(nv_logical_port_t port) {
        decap_counter.count();
        nv_l2_decap(headers);
        nv_send_to_port(port);
    }

```

```
    table decap_v4_table {
        key = {
            headers.vxlan.vni : exact;
        }
        actions = {
            decap;
            to_port;
            decap_to_port;
            deny;
            NoAction;
        }
        size = DECAP_TABLE_SIZE;
        direct_counter = decap_counter;
        default_action = NoAction;
    }

    apply {
        if (headers.vxlan.isValid()) {
            decap_v4_table.apply();
        }
    }
}

control gateway(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    overlay_encap() over;
    underlay_decap() under;

    /* user should add entries that correspond to the wire ports
     * A hit means this is an RX packet, miss means a TX packet
    */
    table direction_table {
        key = {
            std_meta.ingress_port : exact;
        }
        actions = {
            NoAction;
        }
        default_action = NoAction;
        const entries = {
            (WIRE_PORT) : NoAction();
        }
    }

    apply {
        if (direction_table.apply().hit) {
            under.apply(headers, std_meta, user_meta, pkt_out_meta);
        }
        else {
            over.apply(headers, std_meta, user_meta, pkt_out_meta);
        }
    }
}

NvDocaPipeline(
    packet_parser(),
    gateway()
) main;
```

8.11.7.6.10 Connection Tracking Example

Connection tracking is a stateful process that follows the a TCP/UDP session from establishment to termination. It consists of a 5 tuple match on <Source IP address, Destination IP address, IP protocol, L4 Source port, L4 Destination port>. The DPL datapath relies on a controller to insert entries for new 5 tuple flows that have not been seen yet.



8.11.7.6.10.1 Sample Code

Connection metadata is a struct that sent with a packet that is sent to the P4Runtime controller. It requires a special annotation with the label "packet_in". The user can define up to 32 bits of data in this data structure.

```
#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>

@nv_controller_metadata("packet_in")
struct connection_meta_t {
    bit<2> type;
    bit<2> _reserved;
    bit<28> zone;
}
```

In this sample, the packet processing pipeline examines the state of the connection based on the TCP flags and set in the connection metadata an enum type describing the state of the flow.

```
/* Meta connection type */
enum bit<2> ct_type {
    PASS = 0,
    NEW = 1,
    RST = 2,
    FIN = 3
}
```

For readability, constants are used to assign P4 port IDs to more a more meaningful symbol name. The mapping of PFs and VFs to P4 port IDs is done separately in the Service Configuration.

```
/*
   VF port to SW entity that manages the flow insertion,
   deletion and expiry times
*/
const bit<32> WIRE_PORT = 32w0;
const bit<32> HAIRPIN_PORT = 32w1;
```

```
control conn_track(
    inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    connection_meta_t ctrl_meta;

    /* 5-tuple matching for L4 TCP/UDP flows */
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) table_t5_counter;

    action set_connection_id_ct_table_t5(bit<28> zone) {
        table_t5_counter.count();
        ctrl_meta.zone = zone;
        ctrl_meta.type = ct_type.PASS;
    }

    action no_action_ct_table_t5() {
        table_t5_counter.count();
    }

    table ct_table_t5 {
        key = {
            headers.ipv4.src_addr : exact;
            headers.ipv4.dst_addr : exact;
            headers.ipv4.protocol : exact;
            headers.tcp.src_port : exact;
            headers.tcp.dst_port : exact;
        }
        actions = {
            set_connection_id_ct_table_t5;
            no_action_ct_table_t5;
        }
        direct_counter = table_t5_counter;
        // on hit table ct_table_known
        // on miss table ct_table_tcp_miss
        default_action = no_action_ct_table_t5;
        size = 1048576;
    }

    /*
    * Known connections handling.
    * Precondition - must be a TCP packet
    *  RST:    TYPE_RST
    *  FIN:    TYPE_FIN
    *  FINRST: TYPE_RST
    *
    * Match: tcp.flags
    * Actions: meta.type, next PIPE
    * MISS: next PIPE
    */
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) table_known_counter;

    action set_connection_type_ct_table_known(bit<2> type) {
        table_known_counter.count();
        ctrl_meta.type = type;
    }

    table ct_table_known {
        key = {
            headers.tcp.flags : exact;
        }
        actions = {
            set_connection_type_ct_table_known;
        }
        direct_counter = table_known_counter;
        default_action = set_connection_type_ct_table_known(ct_type.PASS);
        const entries = {
            ( 0x1) : set_connection_type_ct_table_known(ct_type.FIN);
            ( 0x4) : set_connection_type_ct_table_known(ct_type.RST);
            (/*NV_TCP_PROTOCOL,*/ 0x5) : set_connection_type_ct_table_known(ct_type.RST); /* RST+FIN */
        }
    }

    /*
    * Unknown TCP connections handling
    * Precondition - must be a TCP packet
    *  SYN: TYPE_NEW
    *
    * Match: tcp.flags
    * Actions: PIPE, meta.type=NEW
    * MISS: NEXT PIPE
    */
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) table_tcp_miss_counter;

    action set_connection_type_ct_table_tcp(bit<2> type) {
        table_tcp_miss_counter.count();
        ctrl_meta.type = type;
    }
```

```
    action no_action_ct_table_tcp() {
        table_tcp_miss_counter.count();
    }

    table ct_table_tcp {
        key = {
            headers.tcp.flags : exact;
        }
        actions = {
            set_connection_type_ct_table_tcp;
            no_action_ct_table_tcp;
        }
        direct_counter = table_tcp_miss_counter;
        default_action = no_action_ct_table_tcp;
        const entries = {
            (0x02) : set_connection_type_ct_table_tcp(ct_type.NEW);  /* SYN=1 ACK=0 */
        }
    }

    /*
    * Unknown UDP connections handling
    *
    * Match: UDP
    * Actions: PIPE, meta.type=NEW
    * MISS: DROP
    */
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) table_udp_miss_counter;

    action set_connection_type_ct_table_udp(bit<2> type) {
        table_udp_miss_counter.count();
        ctrl_meta.type = type;
    }

    action drop_ct_table_udp() {
        table_udp_miss_counter.count();
        nv_drop();
    }

    table ct_table_udp {
        key = {
            std_meta.l4_type : exact;
        }
        actions = {
            set_connection_type_ct_table_udp;
            drop_ct_table_udp();
        }
        direct_counter = table_udp_miss_counter;
        default_action = drop_ct_table_udp();
        const entries = {
            (L4_TYPE_UDP) : set_connection_type_ct_table_udp(ct_type.NEW);  /* SYN=1 ACK=0 */
        }
    }

    /* user should add entries that correspond to the wire ports
     * A hit means this is an RX packet, miss means a TX packet
    */
    table direction_table {
        key = {
            std_meta.ingress_port : exact;
        }
        actions = {
            NoAction;
        }
        default_action = NoAction;
        const entries = {
            (WIRE_PORT) : NoAction();
        }
    }

    apply {
        ctrl_meta.type = ct_type.PASS;
        ctrl_meta._reserved = 0;
        ctrl_meta.zone = 0;

        if (direction_table.apply().hit) {
            if (std_meta.is_l4_ok == 1w1) {
                if (ct_table_t5.apply().hit) {
                    if (ct_table_known.apply().hit) {
                        nv_send_to_controller(ctrl_meta);
                    }
                } else if (headers.tcp.isValid()) {
                    if (ct_table_tcp.apply().hit) {
                        nv_send_to_controller(ctrl_meta);
                    }
                } else {
                    if (ct_table_udp.apply().hit) {
                        nv_send_to_controller(ctrl_meta);
                    }
                }
            }
        }
        nv_send_to_port(HAIRPIN_PORT);
    }
}
```

See below for the complete DPL example.

```
1  /*
2   * SPDX-FileCopyrightText: Copyright (c) 2024 NVIDIA CORPORATION & AFFILIATES. All rights reserved.
3   * SPDX-License-Identifier: LicenseRef-NvidiaProprietary
4   *
```

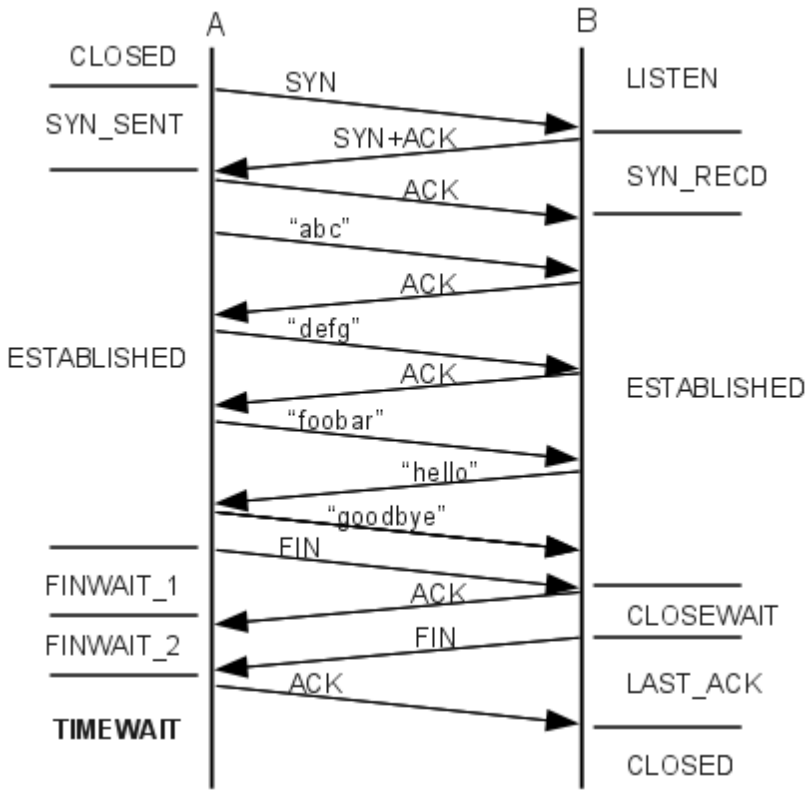
```
 * NVIDIA CORPORATION, its affiliates and licensors retain all intellectual
 * property and proprietary rights in and to this material, related
 * documentation and any modifications thereto. Any use, reproduction,
 * disclosure or distribution of this material and related documentation
 * without an express license agreement from NVIDIA CORPORATION or
 * its affiliates is strictly prohibited.
 */

#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>


/* Basic Connection Tracking example for SDN */

@nv_controller_metadata("packet_in")
struct connection_meta_t {
    bit<2> type;
    bit<2> _reserved;
    bit<28> zone;
}

/* Meta connection type */
enum bit<2> ct_type {
    PASS = 0,
    NEW = 1,
    RST = 2,
    FIN = 3
}

/*
   VF port to SW entity that manages the flow insertion,
   deletion and expiry times
*/
const bit<32> WIRE_PORT = 32w00;
const bit<32> HAIRPIN_PORT = 32w01;


control conn_track(
    inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    connection_meta_t ctrl_meta;

    /* 5-tuple matching for L4 TCP/UDP flows */
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) table_t5_counter;

    action set_connection_id_ct_table_t5(bit<28> zone) {
        table_t5_counter.count();
        ctrl_meta.zone = zone;
        ctrl_meta.type = ct_type.PASS;
    }

    action no_action_ct_table_t5() {
        table_t5_counter.count();
    }

    table ct_table_t5 {
        key = {
            headers.ipv4.src_addr : exact;
            headers.ipv4.dst_addr : exact;
            headers.ipv4.protocol : exact;
            headers.tcp.src_port : exact;
            headers.tcp.dst_port : exact;
        }
        actions = {
            set_connection_id_ct_table_t5;
            no_action_ct_table_t5;
        }
        direct_counter = table_t5_counter;
        // on hit table ct_table_known
        // on miss table ct_table_tcp_miss
        default_action = no_action_ct_table_t5;
        size = 1048576;
    }

    /*
    * Known connections handling.
    * Precondition - must be a TCP packet
    *  RST:    TYPE_RST
    *  FIN:    TYPE_FIN
    *  FINRST: TYPE_RST
    *
    * Match: tcp.flags
    * Actions: meta.type, next PIPE
    * MISS: next PIPE
    */
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) table_known_counter;

    action set_connection_type_ct_table_known(bit<2> type) {
        table_known_counter.count();
        ctrl_meta.type = type;
    }

    table ct_table_known {
        key = {
            headers.tcp.flags : exact;
        }
        actions = {
            set_connection_type_ct_table_known;
        }
        direct_counter = table_known_counter;
```

```
110             default_action = set_connection_type_ct_table_known(ct_type.PASS);
111             const entries = {
112                 ( 0x1) : set_connection_type_ct_table_known(ct_type.FIN);
113                 ( 0x4) : set_connection_type_ct_table_known(ct_type.RST);
114                 (/*NV_TCP_PROTOCOL,*/ 0x5) : set_connection_type_ct_table_known(ct_type.RST); /* RST+FIN */
115             }
116         }
117 
118         /*
119         * Unknown TCP connections handling
120         * Precondition - must be a TCP packet
121         *  SYN: TYPE_NEW
122         *
123         * Match: tcp.flags
124         * Actions: PIPE, meta.type=NEW
125         * MISS: NEXT PIPE
126         */
127         NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) table_tcp_miss_counter;
128 
129         action set_connection_type_ct_table_tcp(bit<2> type) {
130             table_tcp_miss_counter.count();
131             ctrl_meta.type = type;
132         }
133 
134         action no_action_ct_table_tcp() {
135             table_tcp_miss_counter.count();
136         }
137 
138         table ct_table_tcp {
139             key = {
140                 headers.tcp.flags : exact;
141             }
142             actions = {
143                 set_connection_type_ct_table_tcp;
144                 no_action_ct_table_tcp;
145             }
146             direct_counter = table_tcp_miss_counter;
147             default_action = no_action_ct_table_tcp;
148             const entries = {
149                 (0x02) : set_connection_type_ct_table_tcp(ct_type.NEW);  /* SYN=1 ACK=0 */
150             }
151         }
152 
153         /*
154         * Unknown UDP connections handling
155         *
156         * Match: UDP
157         * Actions: PIPE, meta.type=NEW
158         * MISS: DROP
159         */
160         NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) table_udp_miss_counter;
161 
162         action set_connection_type_ct_table_udp(bit<2> type) {
163             table_udp_miss_counter.count();
164             ctrl_meta.type = type;
165         }
166 
167         action drop_ct_table_udp() {
168             table_udp_miss_counter.count();
169             nv_drop();
170         }
171 
172         table ct_table_udp {
173             key = {
174                 std_meta.l4_type : exact;
175             }
176             actions = {
177                 set_connection_type_ct_table_udp;
178                 drop_ct_table_udp();
179             }
180             direct_counter = table_udp_miss_counter;
181             default_action = drop_ct_table_udp();
182             const entries = {
183                 (L4_TYPE_UDP) : set_connection_type_ct_table_udp(ct_type.NEW);  /* SYN=1 ACK=0 */
184             }
185         }
186 
187         /* user should add entries that correspond to the wire ports
188          * A hit means this is an RX packet, miss means a TX packet
189         */
190         table direction_table {
191             key = {
192                 std_meta.ingress_port : exact;
193             }
194             actions = {
195                 NoAction;
196             }
197             default_action = NoAction;
198             const entries = {
199                 (WIRE_PORT) : NoAction();
200             }
201         }
202 
203         apply {
204             ctrl_meta.type = ct_type.PASS;
205             ctrl_meta._reserved = 0;
206             ctrl_meta.zone = 0;
207 
208             if (direction_table.apply().hit) {
209                 if (std_meta.is_l4_ok == 1w1) {
210                     if (ct_table_t5.apply().hit) {
211                         if (ct_table_known.apply().hit) {
212                             nv_send_to_controller(ctrl_meta);
213                         }
214                     } else if (headers.tcp.isValid()) {
```

```
                    if (ct_table_tcp.apply().hit) {
                        nv_send_to_controller(ctrl_meta);
                    }
                } else {
                    if (ct_table_udp.apply().hit) {
                        nv_send_to_controller(ctrl_meta);
                    }
                }
            }
        }
        nv_send_to_port(HAIRPIN_PORT);
    }
}

// Instantiate the top-level DOCA Rx package
NvDocaPipeline(
    nv_fixed_parser(),
    conn_track()
) main;
```

8.11.7.6.11 TCP State Tracking

The TCP State feature in the DOCA Target Architecture provides a robust mechanism for tracking TCP flow parameters, specifically sequence and acknowledgment numbers, directly within the data plane of the BlueField device. The ability to monitor and maintain stateful information about TCP flows offloads this task that traditionally required host CPU processing. With the DPL NvTcpState extern object, developers can programmatically store and query TCP state for each flow at high speeds, enabling fine-grained management of network connections on the DPU.



Tracking TCP sequence and acknowledgment numbers in networking devices, especially within network interface controllers (NICs), empowers a wide range of advanced use cases:

- Enables in-line enforcement of connection security and integrity by validating packet order, detecting replayed or duplicated segments, and preventing sequence-based attacks.

- Provides the basis for in-hardware filtering or offloading of application protocols, as flows can be programmatically distinguished by their connection phases (SYN, SYN-ACK, FIN, RST), supporting real-time acceleration, monitoring, and troubleshooting.
- Facilitates granular stateful firewalling, DDoS mitigation, and anomaly detection by mapping the bi-directional progress of TCP sessions and quickly identifying abnormal or terminated connections at line-rate without host intervention.

By integrating TCP state tracking at the NIC level, operators gain a powerful tool for building scalable, high-performance network functions—from load balancers and firewalls to traffic steering and protocol offloading—while conserving host resources and reducing application latency.

8.11.7.6.11.1 Sample Code

This pipeline consists of a single control, with the following control level objects:

- Three direct counters: for connection tracking, RX/TX forwarding, and ARP table accesses.
- A hardware-accelerated TCP state tracker (**NvTcpState**) for storing and updating TCP connection state information.

```
control c(
    inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout usermeta_conntrack_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) conntrack_counter;
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) rx_tx_table_counter;
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) arp_table_counter;
    NvTcpState(TABLE_SIZE) ct_tcp_state;
```

An ARP table is used to:

- tracks ARP requests and responses, matching packets based on EtherType and ingress port, and forwarding them to the opposite side (uplink or VF) as appropriate.
- Includes per-entry counters and default no-op handling for unmatched ARP messages.

```
    action arp_forward(nv_logical_port_t port) {
        arp_table_counter.count();
        nv_send_to_port(port);
    }

    table arp_table {
        key = {
            std_meta.last_l2_ether_type : exact;
            std_meta.ingress_port : exact;
        }
        actions = {
            arp_forward;
            NoAction;
        }
        size = TABLE_SIZE;
        direct_counter = arp_table_counter;
        default_action = NoAction();
        const entries = {
            (NV_TYPE_ARP, UPLINK_DPL_LOGICAL_PORT_ID): arp_forward(VF_DPL_LOGICAL_PORT_ID);
            (NV_TYPE_ARP, VF_DPL_LOGICAL_PORT_ID): arp_forward(UPLINK_DPL_LOGICAL_PORT_ID);
        }
    }
```

If the packet has TCP flags of FIN, RST or SYN-ACK, the packet is sent to a controller to insert or remove flow entries. The handshake packets are then reinjected to the pipeline by the controller, and are forwarded using a RX/TX table. This table

- Defines actions to forward (**rx_tx_forward**) packets to a given port with accounting, or to drop (**rx_tx_drop**) a packet, both incrementing their respective counters.

- Uses the packet's IPv4 source address to determine its forwarding. Its entries map well-known local/remote IPs to hardware port IDs, forwarding accordingly or dropping unmatched packets.
- By default drops unrecognized traffic

```
action rx_tx_forward(nv_logical_port_t port) {
    rx_tx_table_counter.count();
    nv_send_to_port(port);
}

action rx_tx_drop() {
    rx_tx_table_counter.count();
    nv_drop();
}

table rx_tx_table {
    key = {
        headers.ipv4.src_addr : exact;
    }
    actions = {
        rx_tx_forward;
        rx_tx_drop;
        NoAction;
    }
    size = TABLE_SIZE;
    direct_counter = rx_tx_table_counter;
    default_action = rx_tx_drop();
    const entries = {
        (LOCAL_IP_ADDRESS): rx_tx_forward(UPLINK_DPL_LOGICAL_PORT_ID);
        (REMOTE_IP_ADDRESS): rx_tx_forward(VF_DPL_LOGICAL_PORT_ID);
    }
}
```

The remaining TCP session packets are processed by the main Match/Action block consisting of a Conntrack Table, with:

- An action (**track_tcp_conn**) that increments a connection tracking counter and updates the TCP state tracker with flow index and direction.
- A stateful table (**conntrack_table**) that matches on the IPv4 5-tuple and protocol for each TCP flow and applies either **track_tcp_conn** or no action.
- A per-entry direct counter for packet and byte statistics on each tracked connection.

The **NvTcpState** method **store** is called for all ongoing packets of the TCP session, which records the current SEQ/ACK numbers in stateful memory. The TCP state can be retrieved by the controller application via APIs, and are not available to be read by the data plane.

```
action track_tcp_conn(bit<32> index, NvTcpFlowDirection direction) {
    conntrack_counter.count();
    user_meta.conntrackOut = ct_tcp_state.store(index, direction);
}

table conntrack_table {
    key = {
        headers.ipv4.src_addr : exact;
        headers.ipv4.dst_addr : exact;
        headers.ipv4.protocol : exact;
        headers.tcp.src_port : exact;
        headers.tcp.dst_port : exact;
    }
    actions = {
        track_tcp_conn;
        NoAction;
    }
    size = TABLE_SIZE;
    direct_counter = conntrack_counter;
    default_action = NoAction();
}
```

Finally, the TCP session data packets are sent to the **rx_tx_table** to reached the desired destination.

See below for the complete DPL example.

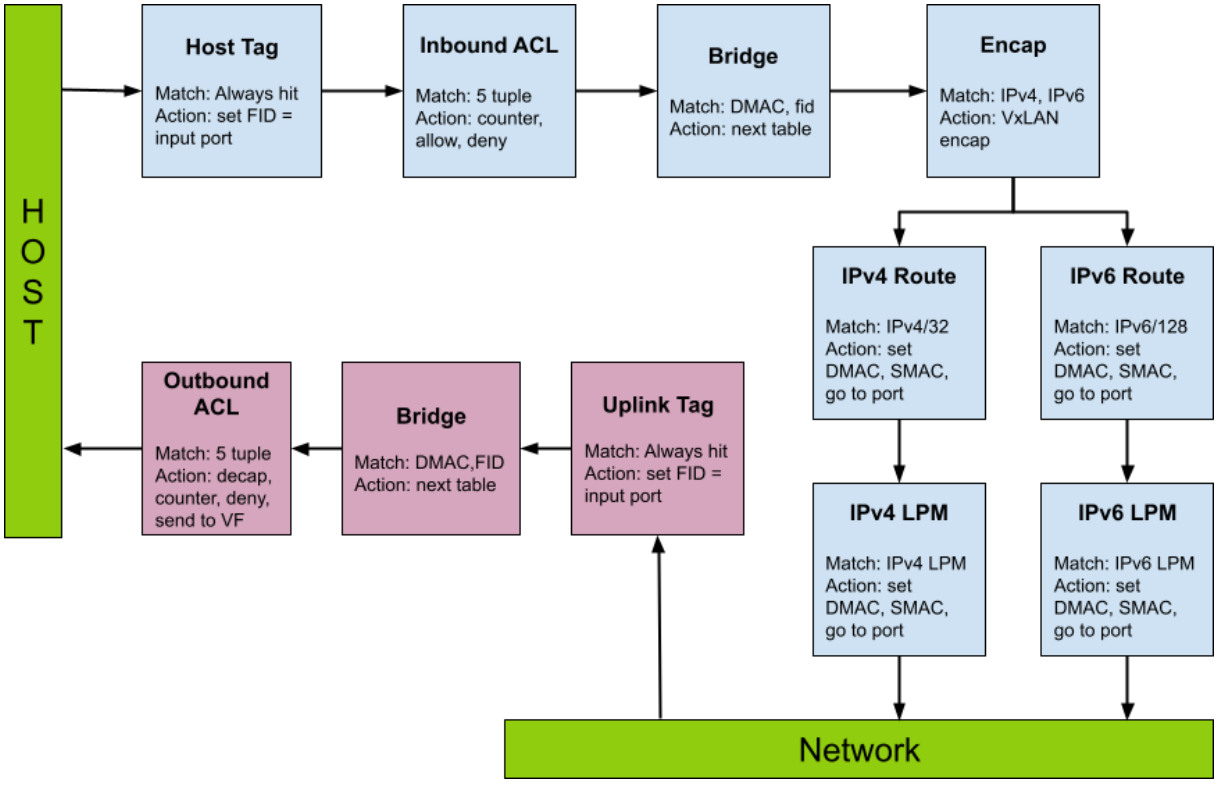
```
1  /*
```

```
 2  * SPDX-FileCopyrightText: Copyright (c) 2025 NVIDIA CORPORATION & AFFILIATES. All rights reserved.
 3  * SPDX-License-Identifier: LicenseRef-NvidiaProprietary
 4  *
 5  * NVIDIA CORPORATION, its affiliates and licensors retain all intellectual
 6  * property and proprietary rights in and to this material, related
 7  * documentation and any modifications thereto. Any use, reproduction,
 8  * disclosure or distribution of this material and related documentation
 9  * without an express license agreement from NVIDIA CORPORATION or
10  * its affiliates is strictly prohibited.
11  */
12
13 #include <doca_model.p4>
14 #include <doca_headers.p4>
15 #include <doca_externs.p4>
16 #include <doca_parser.p4>
17
18 // The port number of VF/SF that is managed by the controller DPDK application.
19 // The controller adds and removes entries from the conntrack table.
20 #define CONTROLLER_DPL_LOGICAL_PORT_ID 2
21
22 // The port number of the uplink port.
23 #define UPLINK_DPL_LOGICAL_PORT_ID 0
24
25 // The port number of the VF port.
26 #define VF_DPL_LOGICAL_PORT_ID 1
27
28 #define LOCAL_IP_ADDRESS  32w0x63000008 // 99.0.0.8
29 #define REMOTE_IP_ADDRESS 32w0x63000016 // 99.0.0.22
30
31 #define TCP_FLAG_FIN_BIT_INDEX 0
32 #define TCP_FLAG_SYN_BIT_INDEX 1
33 #define TCP_FLAG_RST_BIT_INDEX 2
34 #define TCP_FLAG_PUSH_BIT_INDEX 3
35 #define TCP_FLAG_ACK_BIT_INDEX 4
36
37 struct usermeta_conntrack_t {
38     bit<8> conntrackOut;
39 };
40
41 #define TABLE_SIZE 128
42
43 // Pipeline section
44 control c(
45     inout nv_headers_t headers,
46     in nv_standard_metadata_t std_meta,
47     inout usermeta_conntrack_t user_meta,
48     inout nv_empty_metadata_t pkt_out_meta
49 ) {
50     NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) conntrack_counter;
51     NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) rx_tx_table_counter;
52     NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) arp_table_counter;
53     NvTcpState(TABLE_SIZE) ct_tcp_state;
54
55     /*****************************************************************************************
56      * Conntrack table
57      */
58     action track_tcp_conn(bit<32> index, NvTcpFlowDirection direction) {
59         conntrack_counter.count();
60         user_meta.conntrackOut = ct_tcp_state.store(index, direction);
61     }
62
63     table conntrack_table {
64         key = {
65             headers.ipv4.src_addr : exact;
66             headers.ipv4.dst_addr : exact;
67             headers.ipv4.protocol : exact;
68             headers.tcp.src_port : exact;
69             headers.tcp.dst_port : exact;
70         }
71         actions = {
72             track_tcp_conn;
73             NoAction;
74         }
75         size = TABLE_SIZE;
76         direct_counter = conntrack_counter;
77         default_action = NoAction();
78     }
79
80     /*****************************************************************************************
81      * RX/TX table
82      */
83
84     action rx_tx_forward(nv_logical_port_t port) {
85         rx_tx_table_counter.count();
86         nv_send_to_port(port);
87     }
88
89     action rx_tx_drop() {
90         rx_tx_table_counter.count();
91         nv_drop();
92     }
93
94     table rx_tx_table {
95         key = {
96             headers.ipv4.src_addr : exact;
97         }
98         actions = {
99             rx_tx_forward;
100            rx_tx_drop;
101            NoAction;
102        }
103        size = TABLE_SIZE;
104        direct_counter = rx_tx_table_counter;
105        default_action = rx_tx_drop();
106        const entries = {
```

```
            (LOCAL_IP_ADDRESS): rx_tx_forward(UPLINK_DPL_LOGICAL_PORT_ID);
            (REMOTE_IP_ADDRESS): rx_tx_forward(VF_DPL_LOGICAL_PORT_ID);
        }
    }

    /*****************************************************************************************
     * ARP table
     */
    action arp_forward(nv_logical_port_t port) {
        arp_table_counter.count();
        nv_send_to_port(port);
    }

    table arp_table {
        key = {
            std_meta.last_l2_ether_type : exact;
            std_meta.ingress_port : exact;
        }
        actions = {
            arp_forward;
            NoAction;
        }
        size = TABLE_SIZE;
        direct_counter = arp_table_counter;
        default_action = NoAction();
        const entries = {
            (NV_TYPE_ARP, UPLINK_DPL_LOGICAL_PORT_ID): arp_forward(VF_DPL_LOGICAL_PORT_ID);
            (NV_TYPE_ARP, VF_DPL_LOGICAL_PORT_ID): arp_forward(UPLINK_DPL_LOGICAL_PORT_ID);
        }
    }

    /*****************************************************************************************
     * Control plane
     */
    apply {
        user_meta.conntrackOut = 8w0;

        if (std_meta.ingress_port == CONTROLLER_DPL_LOGICAL_PORT_ID) {
            // Packets transmitted from the controller: Send the packet to the desired destination.
            rx_tx_table.apply(); // execution ends here.
        } else {
            // Normal packets from network or VF.

            // Handle ARP packets.
            arp_table.apply(); // on hit execution ends here.

            // Handle valid TCP packets.
            if (headers.ipv4.isValid() && headers.tcp.isValid()) {
                // Check if the packet has a FIN, RST or SYN_ACK TCP flag.
                if ((headers.tcp.flags[TCP_FLAG_FIN_BIT_INDEX:TCP_FLAG_FIN_BIT_INDEX] == 1) ||
                    (headers.tcp.flags[TCP_FLAG_RST_BIT_INDEX:TCP_FLAG_RST_BIT_INDEX] == 1) ||
                    (headers.tcp.flags[TCP_FLAG_SYN_BIT_INDEX:TCP_FLAG_SYN_BIT_INDEX] == 1 &&
                     headers.tcp.flags[TCP_FLAG_ACK_BIT_INDEX:TCP_FLAG_ACK_BIT_INDEX] == 1)) {
                    // Send the packet to the controller to add or remove entries.
                    // Note: The controller is expected to send back the packet once the entries are
updated.
                    nv_send_to_port(CONTROLLER_DPL_LOGICAL_PORT_ID); // execution ends here.
                } else {
                    conntrack_table.apply(); // execution does NOT end here.
                }
            }

            // Send the packet to the desired destination.
            rx_tx_table.apply(); // execution ends here.
        }
    }
}

// Instantiate the top-level DPU pipeline package
NvDocaPipeline(
    nv_fixed_parser(),
    c()
) main;
```

8.11.7.6.12 Host-based Networking Example

HBN is service that provides controller-less VPC networking solution for bare metal as a service (BMaaS), featuring tenant isolation with EVPN VXLAN and accelerated routing on the host. This example demonstrates how to build a DPL datapath that supports host-based networking (HBN).



8.11.7.6.12.1 Sample Code

This example uses indirect counters to monitor the number of packets that are allowed, denied and encapsulated/decapsulated. A user defined forwarding ID is used for determining how the bridge behaves.

```
 enum bit<32> AdmissionCounter_t {
    DENY = 0,
    ALLOW = 1,
    VXLAN_V4_ENCAP = 2,
    VXLAN_V6_ENCAP = 3
 }

 enum bit<32> DecapFlowCounter_t {
    DENY = 0,
    ALLOW = 1,
 }

/* The directionality is based on network to host
 * The user will configure the P4 port IDs in OVS
 */
const bit<32> WIRE_PORT = 32w0;

/* fid is 32 bits
*/
typedef bit<32> fid_t;

struct metadata_t {
    fid_t fid;
}
```

The default parser is used for this example:

```
struct headers_t {
    NV_FIXED_HEADERS
}

parser packet_parser(packet_in packet, out headers_t headers) {
    NV_FIXED_PARSER(packet, headers)
```

```
}
```

Finally, the main control body invokes three separate sub-controls:

- **overlay_encap** – defines the logic and policies for applying VXLAN encapsulations along with ACLs
- **underlay_route** – applies a custom IPv4/IPv6 routing tables with exact and LPM matching. The next hop is set, TTL decremented, and the packet is routed.

```
/**
* This control performs the overlay policy including L2 encap with VxLAN
*/
control overlay_encap(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) fid_counter;
    NvCounter(4, NvCounterType.PACKETS_AND_BYTES) admission_counter;

    action set_fid(fid_t fid) {
        user_meta.fid = fid;
    }
    action deny() {
        admission_counter.count(AdmissionCounter_t.DENY);
        nv_drop();
    }
    action allow() {
        admission_counter.count(AdmissionCounter_t.ALLOW);
    }
    action vxlan_v4_encap(nv_mac_addr_t underlay_src_mac, nv_mac_addr_t underlay_dst_mac,
                nv_ipv4_addr_t underlay_sip, nv_ipv4_addr_t underlay_dip, bit<24> vni) {
        nv_set_vxlan_v4_underlay(headers, underlay_dst_mac, underlay_src_mac, 0, underlay_sip, underlay_dip, vni);
        admission_counter.count(AdmissionCounter_t.VXLAN_V4_ENCAP);
    }
    action vxlan_v6_encap(nv_mac_addr_t underlay_src_mac, nv_mac_addr_t underlay_dst_mac,
                nv_ipv6_addr_t underlay_sip, nv_ipv6_addr_t underlay_dip, bit<24> vni) {
        nv_set_vxlan_v6_underlay(headers, underlay_dst_mac, underlay_src_mac, 0, underlay_sip, underlay_dip, vni);
        admission_counter.count(AdmissionCounter_t.VXLAN_V6_ENCAP);
    }

    // A FID could be as simple as ingress port to FID mapping
    // e.g. ports 1 and 2 are in FID 1, ports 3 and 4 are in FID 2
    table fid_table {
        key = {
            std_meta.ingress_port : exact;
        }
        actions = {
            set_fid;
            NoAction;
        }
        size = FID_TABLE_SIZE;
        default_action = NoAction;
        direct_counter = fid_counter;
    }

    table admit_v4_table {
        key = {
            headers.ipv4.src_addr : exact;
            headers.ipv4.dst_addr : exact;
            headers.ipv4.protocol : exact;
            headers.tcp.src_port : exact;
            headers.tcp.dst_port : exact;
        }
        actions = {
            allow;
            deny;
            NoAction;
        }
        size = ADMIT_TABLE_SIZE;
        default_action = NoAction;
    }

    table admit_v6_table {
        key = {
            headers.ipv6.src_addr : exact;
            headers.ipv6.dst_addr : exact;
            headers.ipv6.next_header : exact;
            headers.tcp.src_port : exact;
            headers.tcp.dst_port : exact;
        }
        actions = {
            allow;
            deny;
            NoAction;
        }
        size = ADMIT_TABLE_SIZE;
        default_action = NoAction;
    }

    table bridge {
        key = {
            headers.ethernet.dst_addr : exact;
            user_meta.fid : exact;
```

```
        }
        actions = {
            NoAction;
        }
        size = BRIDGE_TABLE_SIZE;
        default_action = NoAction;
    }

    table encap_v4_table {
        key = {
            headers.ipv4.dst_addr : lpm;
        }
        actions = {
            vxlan_v4_encap;
        }
        size = ENCAP_TABLE_SIZE;
        default_action = vxlan_v4_encap(DEFAULT_DST_MAC, DEFAULT_SRC_MAC,
                                        DEFAULT_SRC_IPV4, DEFAULT_DST_IPV4, DEFAULT_VNI);
    }

    table encap_v6_table {
        key = {
            headers.ipv6.dst_addr : lpm;
        }

        actions = {
            vxlan_v6_encap;
        }
        size = ENCAP_TABLE_SIZE;
        default_action = vxlan_v6_encap(DEFAULT_DST_MAC, DEFAULT_SRC_MAC,
                                         DEFAULT_SRC_IPV6, DEFAULT_DST_IPV6, DEFAULT_VNI);

    }
    apply {
        user_meta.fid = 0;
        fid_table.apply();
        if (headers.ipv4.isValid()) {
            admit_v4_table.apply();
            if (bridge.apply().hit) {
                encap_v4_table.apply();
            }
        }
        else if (headers.ipv6.isValid()) {
            admit_v6_table.apply();
            if (bridge.apply().hit) {
                encap_v6_table.apply();
            }
        }
    }
}

/**
* This control performs the underlay policy with routing
*/
control underlay_route(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    action set_port_and_route(nv_logical_port_t port, nv_mac_addr_t src_mac, nv_mac_addr_t dst_mac) {
        headers.ethernet.src_addr = src_mac;
        headers.ethernet.dst_addr = dst_mac;
        nv_dec_ip_ttl(headers, 1);
        nv_send_to_port(port);
    }

    table exact_v4_route {
        key = {
            headers.ipv4.dst_addr : exact;
        }
        actions = {
            set_port_and_route;
            NoAction;
        }
        size = ROUTE_TABLE_SIZE;
        default_action = NoAction;
    }
    table lpm_v4_route {
        key = {
            headers.ipv4.dst_addr : lpm;
        }
        actions = {
            set_port_and_route;
            NoAction;
        }
        size = ROUTE_TABLE_SIZE;
        default_action = NoAction;
    }
    table exact_v6_route {
        key = {
            headers.ipv6.dst_addr : exact;
        }
        actions = {
            set_port_and_route;
            NoAction;
        }
        size = ROUTE_TABLE_SIZE;
        default_action = NoAction;
    }
    table lpm_v6_route {
        key = {
            headers.ipv6.dst_addr : lpm;
        }
        actions = {
```

```
            set_port_and_route;
            NoAction;
        }
        size = ROUTE_TABLE_SIZE;
        default_action = NoAction;
    }

    apply {
        if (headers.ipv4.isValid()) {
            if (exact_v4_route.apply().miss) {
                lpm_v4_route.apply();
            }
        }
        else if (headers.ipv6.isValid()) {
            if (exact_v6_route.apply().miss) {
                lpm_v6_route.apply();
            }
        }
    }
}
```

- **decap_flow** – defines the policies for forwarding ID, L2 bridging, decapsulation of VXLAN and VXLAN-GPE tunnels, and forwarding to the VF

```
/**
* This control is for packets from wire to host (RX)
* and includes policy for L2 decap
*/
control decap_flow(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) fid_counter;
    NvCounter(2, NvCounterType.PACKETS_AND_BYTES) decap_counter;

    action deny() {
        decap_counter.count(DecapFlowCounter_t.DENY);
        nv_drop();
    }
    action set_fid(fid_t fid) {
        user_meta.fid = fid;
    }
    action decap_l2_and_send(nv_logical_port_t port) {
        decap_counter.count(DecapFlowCounter_t.ALLOW);
        nv_l2_decap(headers);
        nv_send_to_port(port);
    }
    action decap_ipv4_and_send(nv_logical_port_t port) {
        decap_counter.count(DecapFlowCounter_t.ALLOW);
        nv_l3_decap(headers, 0xffffffffff, 0x112233445566,NV_TYPE_IPV4,0);
        nv_send_to_port(port);
    }
    action decap_ipv6_and_send(nv_logical_port_t port) {
        decap_counter.count(DecapFlowCounter_t.ALLOW);
        nv_l3_decap(headers, 0xffffffffff, 0x112233445566,NV_TYPE_IPV6,0);
        nv_send_to_port(port);
    }

    table fid_table {
        key = {
            std_meta.ingress_port : exact;
        }
        actions = {
            set_fid;
            NoAction;
        }
        size = FID_TABLE_SIZE;
        default_action = NoAction;
        direct_counter = fid_counter;
    }

    table bridge {
        key = {
            headers.ethernet.dst_addr : exact;
            user_meta.fid : exact;
        }
        actions = {
            deny;
            NoAction;
        }
        size = BRIDGE_TABLE_SIZE;
        default_action = deny;
    }

    table decap_v4_table {
        key = {
            headers.ipv4.src_addr : exact;
            headers.ipv4.dst_addr : exact;
            headers.ipv4.protocol : exact;
            headers.udp.src_port : exact;
            headers.udp.dst_port : exact;
        }
        actions = {
            decap_l2_and_send;
            decap_ipv4_and_send;
            deny;
        }
```

```
        size = DECAP_TABLE_SIZE;
        default_action = deny;
    }

    table decap_v6_table {
        key = {
            headers.ipv6.src_addr : exact;
            headers.ipv6.dst_addr : exact;
            headers.ipv6.next_header : exact;
            headers.udp.src_port : exact;
            headers.udp.dst_port : exact;
        }
        actions = {
            decap_l2_and_send;
            decap_ipv6_and_send;
            deny;
        }
        size = DECAP_TABLE_SIZE;
        default_action = deny;
    }

    apply {
        user_meta.fid = (fid_t) 0;
        fid_table.apply();
        if (bridge.apply().hit) {
            if (headers.ipv4.isValid()) {
                decap_v4_table.apply();
            }
            else if (headers.ipv6.isValid()) {
                decap_v6_table.apply();
            }
        }
    }
}
```

The main control checks which direction the packet came from.

- if the packet is from network to host, it invokes the decap_flow control
- if the packet is from host to network, it invokes the overlay_encap and routing controls

```
control host_based_networking(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    overlay_encap() overlay;
    underlay_route() route;
    decap_flow() decap;

    /* user should add entries that correspond to the wire ports
     * A hit means this is an RX packet, miss means a TX packet
    */
    table direction_table {
        key = {
            std_meta.ingress_port : exact;
        }
        actions = {
            NoAction;
        }
        default_action = NoAction;
        const entries = {
            (WIRE_PORT) : NoAction();
        }
    }

    apply {
        if (direction_table.apply().miss) {
            overlay.apply(headers, std_meta, user_meta, pkt_out_meta);
            route.apply(headers, std_meta, user_meta, pkt_out_meta);
        }
        else {
            decap.apply(headers, std_meta, user_meta, pkt_out_meta);
        }
    }
}
```

See below for the complete DPL example.

```
 1  /*
 2   * SPDX-FileCopyrightText: Copyright (c) 2024-2025 NVIDIA CORPORATION & AFFILIATES. All rights
    reserved.
 3   * SPDX-License-Identifier: LicenseRef-NvidiaProprietary
 4   *
 5   * NVIDIA CORPORATION, its affiliates and licensors retain all intellectual
 6   * property and proprietary rights in and to this material, related
 7   * documentation and any modifications thereto. Any use, reproduction,
 8   * disclosure or distribution of this material and related documentation
 9   * without an express license agreement from NVIDIA CORPORATION or
10   * its affiliates is strictly prohibited.
11   */
12  #include <doca_model.p4>
13  #include <doca_headers.p4>
```

```
#include <doca_externs.p4>
#include <doca_parser.p4>

/*
 * Stretched L2 tenant's network over provider routed network
 * - Underlay router
 * - Overlay L2 encapsulated in VxLAN header, multi tenant's per server (FID to VF mapping)
 * - Policy engine for tenant's 5 tuple + underlay IP
 * - Bridge + policy counter
 */
const nv_mac_addr_t DEFAULT_SRC_MAC = 48w0x1070fd658c88;
const nv_mac_addr_t DEFAULT_DST_MAC = 48w0x9c05911f2a00;
const nv_ipv4_addr_t DEFAULT_SRC_IPV4 = 32w0x01020304;
const nv_ipv4_addr_t DEFAULT_DST_IPV4 = 32w0x11223344;
const nv_ipv6_addr_t DEFAULT_SRC_IPV6 = 128w0x00010002000300040005000600070008;
const nv_ipv6_addr_t DEFAULT_DST_IPV6 = 128w0x11112222333344445555666677778888;
const bit<24> DEFAULT_VNI = 24w0xa;

/*
* Table sizes.
*/
const bit<32> FID_TABLE_SIZE = 1024;
const bit<32> DECAP_TABLE_SIZE = 1048576;
const bit<32> ADMIT_TABLE_SIZE = 1048576;
const bit<32> BRIDGE_TABLE_SIZE = 32768;
const bit<32> ENCAP_TABLE_SIZE = 32768;
const bit<32> ROUTE_TABLE_SIZE = 32768;

 enum bit<32> AdmissionCounter_t {
    DENY = 0,
    ALLOW = 1,
    VXLAN_V4_ENCAP = 2,
    VXLAN_V6_ENCAP = 3
 }

 enum bit<32> DecapFlowCounter_t {
    DENY = 0,
    ALLOW = 1,
 }


/* The directionality is based on network to host
 * The user will configure the DPL port IDs in the DPL RT configuration
 */
const bit<32> WIRE_PORT = 32w0;

/* fid is 32 bits
*/
typedef bit<32> fid_t;

struct metadata_t {
    fid_t fid;
}

struct headers_t {
    NV_FIXED_HEADERS
}

parser packet_parser(packet_in packet, out headers_t headers) {
    NV_FIXED_PARSER(packet, headers)
 }

/**
* This control performs the overlay policy including L2 encap with VxLAN
*/
control overlay_encap(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) fid_counter;
    NvCounter(4, NvCounterType.PACKETS_AND_BYTES) admission_counter;

    action deny() {
        admission_counter.count(AdmissionCounter_t.DENY);
        nv_drop();
    }

    action allow() {
        admission_counter.count(AdmissionCounter_t.ALLOW);
    }

    action vxlan_v4_encap(nv_mac_addr_t underlay_src_mac, nv_mac_addr_t underlay_dst_mac,
                          nv_ipv4_addr_t underlay_sip, nv_ipv4_addr_t underlay_dip, bit<24> vni) {
        nv_set_vxlan_v4_underlay(headers, underlay_dst_mac, underlay_src_mac, 0, underlay_sip,
underlay_dip, vni);
        admission_counter.count(AdmissionCounter_t.VXLAN_V4_ENCAP);
    }

    action vxlan_v6_encap(nv_mac_addr_t underlay_src_mac, nv_mac_addr_t underlay_dst_mac,
                          nv_ipv6_addr_t underlay_sip, nv_ipv6_addr_t underlay_dip, bit<24> vni) {
        nv_set_vxlan_v6_underlay(headers, underlay_dst_mac, underlay_src_mac, 0, underlay_sip,
underlay_dip, vni);
        admission_counter.count(AdmissionCounter_t.VXLAN_V6_ENCAP);
    }

    action set_fid_fid_table(fid_t fid) {
        fid_counter.count();
        user_meta.fid = fid;
    }

    action no_action_fid_table() {
        fid_counter.count();
    }
```

```
117
118         // A FID could be as simple as ingress port to FID mapping
119         // e.g. ports 1 and 2 are in FID 1, ports 3 and 4 are in FID 2
120         table fid_table {
121             key = {
122                 std_meta.ingress_port : exact;
123             }
124             actions = {
125                 set_fid_fid_table;
126                 no_action_fid_table;
127             }
128             size = FID_TABLE_SIZE;
129             default_action = no_action_fid_table;
130             direct_counter = fid_counter;
131         }
132
133         table admit_v4_table {
134             key = {
135                 headers.ipv4.src_addr : exact;
136                 headers.ipv4.dst_addr : exact;
137                 headers.ipv4.protocol : exact;
138                 headers.tcp.src_port : exact;
139                 headers.tcp.dst_port : exact;
140             }
141             actions = {
142                 allow;
143                 deny;
144                 NoAction;
145             }
146             size = ADMIT_TABLE_SIZE;
147             default_action = NoAction;
148         }
149
150         table admit_v6_table {
151             key = {
152                 headers.ipv6.src_addr : exact;
153                 headers.ipv6.dst_addr : exact;
154                 headers.ipv6.next_header : exact;
155                 headers.tcp.src_port : exact;
156                 headers.tcp.dst_port : exact;
157             }
158             actions = {
159                 allow;
160                 deny;
161                 NoAction;
162             }
163             size = ADMIT_TABLE_SIZE;
164             default_action = NoAction;
165         }
166
167         table bridge {
168             key = {
169                 headers.ethernet.dst_addr : exact;
170                 user_meta.fid : exact;
171             }
172             actions = {
173                 NoAction;
174             }
175             size = BRIDGE_TABLE_SIZE;
176             default_action = NoAction;
177         }
178
179         table encap_v4_table {
180             key = {
181                 headers.ipv4.dst_addr : lpm;
182             }
183             actions = {
184                 vxlan_v4_encap;
185             }
186             size = ENCAP_TABLE_SIZE;
187             default_action = vxlan_v4_encap(DEFAULT_DST_MAC, DEFAULT_SRC_MAC,
188                                             DEFAULT_SRC_IPV4, DEFAULT_DST_IPV4, DEFAULT_VNI);
189         }
190
191         table encap_v6_table {
192             key = {
193                 headers.ipv6.dst_addr : lpm;
194             }
195
196             actions = {
197                 vxlan_v6_encap;
198             }
199             size = ENCAP_TABLE_SIZE;
200             default_action = vxlan_v6_encap(DEFAULT_DST_MAC, DEFAULT_SRC_MAC,
201                                             DEFAULT_SRC_IPV6, DEFAULT_DST_IPV6, DEFAULT_VNI);
202
203         }
204         apply {
205             user_meta.fid = 0;
206             fid_table.apply();
207             if (headers.ipv4.isValid()) {
208                 admit_v4_table.apply();
209                 if (bridge.apply().hit) {
210                     encap_v4_table.apply();
211                 }
212             }
213             else if (headers.ipv6.isValid()) {
214                 admit_v6_table.apply();
215                 if (bridge.apply().hit) {
216                     encap_v6_table.apply();
217                 }
218             }
219         }
220     }
221
```

```
/**
* This control performs the underlay policy with routing
*/
control underlay_route(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    action set_port_and_route_v4(nv_logical_port_t port, nv_mac_addr_t src_mac, nv_mac_addr_t dst_mac)
    {
        headers.ethernet.src_addr = src_mac;
        headers.ethernet.dst_addr = dst_mac;
        headers.ipv4.ttl -= 1;
        nv_send_to_port(port);
    }

    action set_port_and_route_v6(nv_logical_port_t port, nv_mac_addr_t src_mac, nv_mac_addr_t dst_mac)
    {
        headers.ethernet.src_addr = src_mac;
        headers.ethernet.dst_addr = dst_mac;
        headers.ipv6.hop_limit -= 1;
        nv_send_to_port(port);
    }

    table exact_v4_route {
        key = {
            headers.ipv4.dst_addr : exact;
        }
        actions = {
            set_port_and_route_v4;
            NoAction;
        }
        size = ROUTE_TABLE_SIZE;
        default_action = NoAction;
    }
    table lpm_v4_route {
        key = {
            headers.ipv4.dst_addr : lpm;
        }
        actions = {
            set_port_and_route_v4;
            NoAction;
        }
        size = ROUTE_TABLE_SIZE;
        default_action = NoAction;
    }
    table exact_v6_route {
        key = {
            headers.ipv6.dst_addr : exact;
        }
        actions = {
            set_port_and_route_v6;
            NoAction;
        }
        size = ROUTE_TABLE_SIZE;
        default_action = NoAction;
    }
    table lpm_v6_route {
        key = {
            headers.ipv6.dst_addr : lpm;
        }
        actions = {
            set_port_and_route_v6;
            NoAction;
        }
        size = ROUTE_TABLE_SIZE;
        default_action = NoAction;
    }

    apply {
        if (headers.ipv4.isValid()) {
            if (exact_v4_route.apply().miss) {
                lpm_v4_route.apply();
            }
        }
        else if (headers.ipv6.isValid()) {
            if (exact_v6_route.apply().miss) {
                lpm_v6_route.apply();
            }
        }
    }
}

/**
* This control is for packets from wire to host (RX)
* and includes policy for L2 decap
*/
control decap_flow(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) fid_counter;
    NvCounter(2, NvCounterType.PACKETS_AND_BYTES) decap_counter;

    action deny() {
        decap_counter.count(DecapFlowCounter_t.DENY);
        nv_drop();
    }

    action decap_l2_and_send(nv_logical_port_t port) {
        decap_counter.count(DecapFlowCounter_t.ALLOW);
        nv_l2_decap(headers);
```

```
325             nv_send_to_port(port);
326         }
327
328         action decap_ipv4_and_send(nv_logical_port_t port) {
329             decap_counter.count(DecapFlowCounter_t.ALLOW);
330             nv_l3_decap(headers, 0xfffffffffff, 0x112233445566,NV_TYPE_IPV4,0);
331             nv_send_to_port(port);
332         }
333
334         action decap_ipv6_and_send(nv_logical_port_t port) {
335             decap_counter.count(DecapFlowCounter_t.ALLOW);
336             nv_l3_decap(headers, 0xfffffffffff, 0x112233445566,NV_TYPE_IPV6,0);
337             nv_send_to_port(port);
338         }
339
340         action set_fid_fid_table(fid_t fid) {
341             fid_counter.count();
342             user_meta.fid = fid;
343         }
344
345         action no_action_fid_table() {
346             fid_counter.count();
347         }
348
349         table fid_table {
350             key = {
351                 std_meta.ingress_port : exact;
352             }
353             actions = {
354                 set_fid_fid_table;
355                 no_action_fid_table;
356             }
357             size = FID_TABLE_SIZE;
358             default_action = no_action_fid_table;
359             direct_counter = fid_counter;
360         }
361
362         table bridge {
363             key = {
364                 headers.ethernet.dst_addr : exact;
365                 user_meta.fid : exact;
366             }
367             actions = {
368                 deny;
369                 NoAction;
370             }
371             size = BRIDGE_TABLE_SIZE;
372             default_action = deny;
373         }
374
375         table decap_v4_table {
376             key = {
377                 headers.ipv4.src_addr : exact;
378                 headers.ipv4.dst_addr : exact;
379                 headers.ipv4.protocol : exact;
380                 headers.udp.src_port : exact;
381                 headers.udp.dst_port : exact;
382             }
383             actions = {
384                 decap_l2_and_send;
385                 decap_ipv4_and_send;
386                 deny;
387             }
388             size = DECAP_TABLE_SIZE;
389             default_action = deny;
390         }
391
392         table decap_v6_table {
393             key = {
394                 headers.ipv6.src_addr : exact;
395                 headers.ipv6.dst_addr : exact;
396                 headers.ipv6.next_header : exact;
397                 headers.udp.src_port : exact;
398                 headers.udp.dst_port : exact;
399             }
400             actions = {
401                 decap_l2_and_send;
402                 decap_ipv6_and_send;
403                 deny;
404             }
405             size = DECAP_TABLE_SIZE;
406             default_action = deny;
407         }
408
409         apply {
410             user_meta.fid = (fid_t) 0;
411             fid_table.apply();
412             if (bridge.apply().hit) {
413                 if (headers.ipv4.isValid()) {
414                     decap_v4_table.apply();
415                 }
416                 else if (headers.ipv6.isValid()) {
417                     decap_v6_table.apply();
418                 }
419             }
420         }
421     }
422
423     control host_based_networking(
424         inout headers_t headers,
425         in nv_standard_metadata_t std_meta,
426         inout metadata_t user_meta,
427         inout nv_empty_metadata_t pkt_out_meta
428     ) {
429         overlay_encap() overlay;
```

```
    underlay_route() route;
    decap_flow() decap;

    /* user should add entries that correspond to the wire ports
     * A hit means this is an RX packet, miss means a TX packet
    */
    table direction_table {
        key = {
            std_meta.ingress_port : exact;
        }
        actions = {
            NoAction;
        }
        default_action = NoAction;
        const entries = {
            (WIRE_PORT) : NoAction();
        }
    }

    apply {
        if (direction_table.apply().miss) {
            overlay.apply(headers, std_meta, user_meta, pkt_out_meta);
            route.apply(headers, std_meta, user_meta, pkt_out_meta);
        }
        else {
            decap.apply(headers, std_meta, user_meta, pkt_out_meta);
        }
    }
}

NvDocaPipeline(
    packet_parser(),
    host_based_networking()
) main;
```

8.11.7.6.13 Equal-Cost Multi-Path Routing Example

Equal-Cost Multi-Path Routing (ECMP) is a network routing strategy that enables packet forwarding to a single destination over multiple paths that have the same routing cost or metric. This approach is widely used in modern data center and enterprise networks to improve bandwidth utilization, redundancy, and network resilience. ECMP allows the eSwitch on the DPU install multiple next-hop entries for a destination prefix if those paths have equal cost according a hash function based on selected packet header fields. When forwarding packets, the DPL program can select among these equal-cost paths, ensuring that all packets belonging to the same flow follow the same path, while different flows may be distributed across different paths.

8.11.7.6.13.1 Sample Code

The ECMP metadata is a struct that will be used to store the result of a hash calculation over the desired fields.

```
struct usermeta_t {
    bit<4> ecmp_select;
};
```

In he main control of the ECMP example, the logic is as follows:

- create a NvHash object
- define an ECMP group table that filters out the destination IP addresses that will be routed
- calculate the hash over the desired packet fields
- apply a second table that matches on the hash result, and performs the next hop according to the table entries.

```
#define ECMP_BUCKETS 16

control c(
    inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout usermeta_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    // Create a hash object with the selected algorithm
    NvHash(NvHashAlgorithm.CRC32) hash;
```

```
    action drop() {
        nv_drop();
    }

    action next_hop(
        nv_mac_addr_t dmac,
        nv_ipv4_addr_t ipv4,
        nv_logical_port_t port
    ) {
        headers.ethernet.dst_addr = dmac;
        headers.ipv4.dst_addr = ipv4;
        headers.ipv4.ttl -= 1;
        nv_send_to_port(port);
    }

    table ecmp_group {
        key = {
            headers.ipv4.dst_addr: lpm;
        }
        actions = {
            drop;
            NoAction;
        }
        default_action = drop;
        size = 1024;
    }

    table ecmp_nhop {
        key = {
            user_meta.ecmp_select: exact;
        }
        actions = {
            drop;
            next_hop;
        }
        default_action = drop;
        size = ECMP_BUCKETS;
    }
```

In this apply block example, once the packet is validated and part of an ECMP group policy, the hash is calculated over the 5-tuple. Then the next hop is applied with a final destination.

The user has control of how many ECMP buckets are needed. By varying the entries added to the ecmp_nhop table, equal cost can be assigned to each final port destination. Fine grain weighted ECMP can be achieved by increasing the number of buckets and by weighting the entries in an unequal distribution (e.g. to favor one path more than the others). The DPL programmer can also create additional custom logic to affect the distribution.

```
apply {
    if (headers.ipv4.isValid() && std_meta.is_l4_ok && headers.ipv4.ttl != 0) {
        ecmp_group.apply();
        user_meta.ecmp_select =
            (bit<4>)hash.get_hash(
                ECMP_BUCKETS, // size
                {
                    headers.ipv4.src_addr,
                    headers.ipv4.dst_addr,
                    headers.ipv4.protocol,
                    std_meta.l4_src_port,
                    std_meta.l4_dst_port
                }
        );
        ecmp_nhop.apply();
    }
}
```

See below for the complete DPL example.

```
 1  /*
 2   * SPDX-FileCopyrightText: Copyright (c) 2025 NVIDIA CORPORATION & AFFILIATES. All rights reserved.
 3   * SPDX-License-Identifier: LicenseRef-NvidiaProprietary
 4   *
 5   * NVIDIA CORPORATION, its affiliates and licensors retain all intellectual
 6   * property and proprietary rights in and to this material, related
 7   * documentation and any modifications thereto. Any use, reproduction,
 8   * disclosure or distribution of this material and related documentation
 9   * without an express license agreement from NVIDIA CORPORATION or
10   * its affiliates is strictly prohibited.
11   */
12
13  #include <doca_model.p4>
14  #include <doca_headers.p4>
15  #include <doca_externs.p4>
16  #include <doca_parser.p4>
17
18  #define ECMP_BUCKETS 16
19
20  struct usermeta_t {
21      bit<4> ecmp_select;
22  };
```

```
23
24  control c(
25      inout nv_headers_t headers,
26      in nv_standard_metadata_t std_meta,
27      inout usermeta_t user_meta,
28      inout nv_empty_metadata_t pkt_out_meta
29  ) {
30      // Create a hash object with the selected algorithm
31      NvHash(NvHashAlgorithm.CRC32) hash;
32
33      action drop() {
34          nv_drop();
35      }
36
37      action next_hop(
38          nv_mac_addr_t dmac,
39          nv_ipv4_addr_t ipv4,
40          nv_logical_port_t port
41      ) {
42          headers.ethernet.dst_addr = dmac;
43          headers.ipv4.dst_addr = ipv4;
44          headers.ipv4.ttl -= 1;
45          nv_send_to_port(port);
46      }
47
48      table ecmp_group {
49          key = {
50              headers.ipv4.dst_addr: lpm;
51          }
52          actions = {
53              drop;
54              NoAction;
55          }
56          default_action = drop;
57          size = 1024;
58      }
59
60      table ecmp_nhop {
61          key = {
62              user_meta.ecmp_select: exact;
63          }
64          actions = {
65              drop;
66              next_hop;
67          }
68          default_action = drop;
69          size = ECMP_BUCKETS;
70      }
71
72      apply {
73          if (headers.ipv4.isValid() && std_meta.is_l4_ok == 1 && headers.ipv4.ttl != 0) {
74              ecmp_group.apply();
75              user_meta.ecmp_select =
76                  (bit<4>)hash.get_hash(
77                      ECMP_BUCKETS, // size
78                      {
79                          headers.ipv4.src_addr,
80                          headers.ipv4.dst_addr,
81                          headers.ipv4.protocol,
82                          std_meta.l4_src_port,
83                          std_meta.l4_dst_port
84                      }
85              );
86              ecmp_nhop.apply();
87          }
88      }
89  }
90
91  NvDocaPipeline(
92      nv_fixed_parser(),
93      c()
94  ) main;
```

8.11.7.6.14 Entropy in Hash-Based Load Balancing Example

Most load-balancing systems use hash functions to map traffic flows to specific paths. A typical example of a hash input includes packet header fields of IP source/destination addresses, L4 source/destination port, and IP protocol. The effectiveness of load balancing depends on the entropy (randomness) of these fields. A high entropy is obtained when the input values are varied (e.g. diverse source ports) lead to even traffic distribution.

This example demonstrates how DPL can be used to set the entropy for the UDP source port, based on a hash of the 5-tuple. Then for load balancing devices further downstream of the DPU, the UDP source port is consumed as a part of the load balancer's algorithm.

8.11.7.6.14.1 Sample Code

A user metadata struct is defined to hold the temporary result of the entropy calculation:

```
struct usermeta_t {
    bit<32> entropy;
};
```

In this sample, the control needs to

- define the hash object
- match and select the VxLAN packets which to apply the entropy calculation
- calculate the hash over the inner 5-tuple
- set the outer UDP source port with the hash value

Since the underlay header 5-tuple is often the same for different inner 5 tuple flows, setting the outer UDP source port in this manner allows external network devices to load balance based solely on the underlay packet fields.

```
control c(
    inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout usermeta_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    // Create a hash object with the selected algorithm
    NvHash(NvHashAlgorithm.CRC32) hash;

    action drop() {
        nv_drop();
    }

    action transmit(nv_logical_port_t port) {
        nv_send_to_port(port);
    }

    table balancer {
        key = {
            headers.ipv4.dst_addr: lpm;
        }
        actions = {
            drop;
            transmit;
        }
        default_action = drop;
        size = 1024;
    }
```

The apply logic glues the tables details above by filtering out VxLAN packets with a valid inner TCP/UDP.

```
apply {
    if (headers.vxlan.isValid() && std_meta.is_inner_l4_ok != 0) {
        user_meta.entropy =
            hash.get_hash(
                65536,
                {
                    headers.inner_ipv4.src_addr,
                    headers.inner_ipv4.dst_addr,
                    headers.inner_ipv4.protocol,
                    std_meta.inner_l4_src_port,
                    std_meta.inner_l4_dst_port
                }
        );
        nv_set_l4_src_port(headers, (bit<16>)user_meta.entropy);
        balancer.apply();
    }
}
```

See below for the complete DPL example.

```
/*
 * SPDX-FileCopyrightText: Copyright (c) 2025 NVIDIA CORPORATION & AFFILIATES. All rights reserved.
 * SPDX-License-Identifier: LicenseRef-NvidiaProprietary
 *
 * NVIDIA CORPORATION, its affiliates and licensors retain all intellectual
 * property and proprietary rights in and to this material, related
 * documentation and any modifications thereto. Any use, reproduction,
 * disclosure or distribution of this material and related documentation
 * without an express license agreement from NVIDIA CORPORATION or
 * its affiliates is strictly prohibited.
 */

#include <doca_model.p4>
#include <doca_headers.p4>
```

```
#include <doca_externs.p4>
#include <doca_parser.p4>

struct usermeta_t {
    bit<32> entropy;
};

control c(
    inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout usermeta_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    // Create a hash object with the selected algorithm
    NvHash(NvHashAlgorithm.CRC32) hash;

    action drop() {
        nv_drop();
    }

    action transmit(nv_logical_port_t port) {
        nv_send_to_port(port);
    }

    table balancer {
        key = {
            headers.ipv4.dst_addr: lpm;
        }
        actions = {
            drop;
            transmit;
        }
        default_action = drop;
        size = 1024;
    }

    apply {
        if (headers.vxlan.isValid() && std_meta.is_inner_l4_ok != 0) {
            user_meta.entropy =
                hash.get_hash(
                    65536,
                    {
                        headers.inner_ipv4.src_addr,
                        headers.inner_ipv4.dst_addr,
                        headers.inner_ipv4.protocol,
                        std_meta.inner_l4_src_port,
                        std_meta.inner_l4_dst_port
                    }
            );
            nv_set_l4_src_port(headers, (bit<16>)user_meta.entropy);
            balancer.apply();
        }
    }
}

NvDocaPipeline(
    nv_fixed_parser(),
    c()
) main;
```

8.11.7.6.15 NAT64 Example

Network Address Translation (NAT64) is a mechanism enabling connections between IPv6 clients and IPv4 servers. It serves as a gateway, allowing devices on IPv6-only networks to access resources available over IPv4.

Unlike RFC 6052, which uses an algorithmic mapping of IPv6 prefixes to embed IPv4 addresses, this example demonstrates Stateful NAT64. In this model, the gateway maintains a mapping table for active flows, allowing for per-entry user policies.

8.11.7.6.15.1 Sample Code

When translating an IP packet, various packet fields can be saved between IPv6 and IPv4. To accommodate this, create a metadata structure to hold these temporary values:

```
struct ip_data_t {
    bit<16> packet_len_field;
    bit<8> ttl;
    bit<6> tos_field;
    bit<2> ecn_field;
    bit<8> ip_protocol;
}
```

Next, instantiate a **NvHeaderDataTemplate** object. This object is used to define the header and field values that will be pushed into the packet. The template requires two items:

- A **struct** containing the set of headers that will be pushed. The headers are pushed in the order in which they appear in the **struct**, and can be natively supported headers for custom headers.
- An annotation **@nv_header_data_fields** on the **NvHeaderDataTemplate** object that serves as an initializer to the **struct** of headers. Each header must be initialized in the order in which they appear in the structure, and each header field must be initialized in the order in which they appear in the header.

```
struct v4_headers_t {
    nv_ipv4_h      nat_v4;
}

control nat64(
    inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout ip_data_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
      @nv_header_data_fields(
        nat_v4 = {
            version = 4,
            ihl = 5,
            diffserv = 0,
            ecn = 0,
            total_len = 0,
            identification = 0,
            flags = 2,       // Do not fragment
            frag_offset = 0,
            ttl = 64,
            protocol = 6,
            hdr_checksum = 0xf93a, //  Prealculated CRC based on this pattern data
            src_addr = 0,          // must be zero for TCP pseudo-header CRC
            dst_addr = 0,
        }
    )
    NvHeaderDataTemplate<v4_headers_t>() v4DataTemplate;
```

The P4 action contains the logical steps to translate the incoming IPv6 packet to an IPv4 packet. The steps are:

1. Save the necessary IPv6 fields in the **user_metadata** **struct**, such as the TTL and DSCP fields. These will be restored later.
2. Calculate the new packet length field.
3. Push the new IPv4 header, with mostly empty fields. The IPv4 addresses are set to zero for the pseudo-header CRC calculation. Since the push operation leaves the packet in an intermediate state, set the **do_reparse** = false so that the offsets for **IP_START_OF_HEADER** are not updated yet.
4. Pop the IPv6 header with packet updates and reparse disabled.
5. Change the L2 ethertype to IPv4 and trigger a reparse now the packet header rearrangement is complete. This ensures that subsequent modifications to the L3 layer will occur at the right offset in the packet.
6. Change the L3 protocol field to the stored value and trigger a second reparse. This ensures that L3/L4 modifications will correctly update the L4 checksum.
7. Copy the remaining **user_metadata** to the new IPv4 header, and set the source and destination IP addresses.

```
action translate64(nv_ipv4_addr_t src_ip, nv_ipv4_addr_t dst_ip) {
    ip64_counter.count();
    user_meta.packet_len_field = headers.ipv6.payload_length;
    user_meta.packet_len_field += 20;
    user_meta.ip_protocol = headers.ipv6.next_header;
    user_meta.ttl = headers.ipv6.hop_limit;
```

```
    user_meta.tos_field = headers.ipv6.diffserv;
    user_meta.ecn_field = headers.ipv6.ecn;
    // partial TCP pseudo-header CRC calculation
    headers.ipv6.src_addr = 0;
    headers.ipv6.dst_addr = 0;
    // push a mostly empty IPv4 header with the partial checksum
    nv_push_headers(headers, v4DataTemplate,
        { },
        NvAnchor.IP_START_OF_HEADER,
        0, false, false);
    // remove IPv6 header without updates or reparse as packet is in mid-modification
    nv_pop_headers(headers, NvAnchor.IP_START_OF_HEADER, NvAnchor.L4_START_OF_HEADER,
        false, false);
    // change the ethertype to IPv4 and reparse to update L3 field offsets
    nv_set_l2_ethertype(headers, 16w0x0800);
    nv_force_reparse(headers);
    // Update the IP protocol and reparse to update L4 fields
    nv_set_ip_protocol(headers, user_meta.ip_protocol);
    nv_force_reparse(headers);
    // restore the remaining IP fields
    headers.ipv4.src_addr = src_ip;
    headers.ipv4.dst_addr = dst_ip;
    headers.ipv4.ttl = user_meta.ttl;
    headers.ipv4.total_len = user_meta.packet_len_field;
    nv_set_ip_ecn(headers, user_meta.ecn_field);
    nv_set_ip_dscp(headers, user_meta.tos_field);
}
```

Based on the desired policy, a P4 table with the desired keyset is defined. In the **apply** block, the packet is determined to be a valid IPv6 packet. The table is applied during the final step; when a match is found, NAT64 is performed, and the packet is sent to a specific port.

```
    table ipv6_to_ipv4 {
        key = {
            headers.ethernet.src_addr : exact;
        }
        actions = {
            translate64;
            NoAction;
        }
        default_action = NoAction;
        direct_counter = ip64_counter;
    }

    apply {
        if (std_meta.is_l4_ok == 1) {
            if (ipv6_to_ipv4.apply().hit) {
                nv_send_to_port(3);
            }
        }
    }
```

See below for the complete DPL example.

```
/*
 * SPDX-FileCopyrightText: Copyright (c) 2025 NVIDIA CORPORATION & AFFILIATES. All rights reserved.
 * SPDX-License-Identifier: LicenseRef-NvidiaProprietary
 *
 * NVIDIA CORPORATION, its affiliates and licensors retain all intellectual
 * property and proprietary rights in and to this material, related
 * documentation and any modifications thereto. Any use, reproduction,
 * disclosure or distribution of this material and related documentation
 * without an express license agreement from NVIDIA CORPORATION or
 * its affiliates is strictly prohibited.
 */

#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>

/*
 * This basic application demonstrates a NAT from IPv6 to IPv4 and IPv4 to IPv6.
 * Unlike RFC 6052, this version of NAT64 is dynamic in the sense that it varies
 * per entry depending on the user's policy.
 */

struct ip_data_t {
    bit<16> packet_len_field;
    bit<8> ttl;
    bit<6> tos_field;
    bit<2> ecn_field;
    bit<8> ip_protocol;
}

struct v4_headers_t {
    nv_ipv4_h      nat_v4;
}

struct v6_headers_t {
    nv_ipv6_h      nat_v6;
}

control nat64(
```

```
    inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout ip_data_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) ip64_counter;

      @nv_header_data_fields(
        nat_v4 = {
            // WORD 1
            version = 4,
            ihl = 5,
            diffserv = 0,
            ecn = 0,
            // WORD 2
            total_len = 0,
            // WORD 3
            identification = 0,
            // WORD 4
            flags = 2,       // Do not fragment
            frag_offset = 0,
            // WORD 5
            ttl = 64,
            protocol = 6,
            // WORD 6
            hdr_checksum = 0xf93a, //  Prealculated CRC based on this pattern data
            // WORD 7
            src_addr = 0,    // must be zero for TCP pseudo-header CRC
            // WORD 9
            dst_addr = 0,    // must be zero for TCP pseudo-header CRC
        }
    )
    NvHeaderDataTemplate<v4_headers_t>() v4DataTemplate;

    action drop() {
        nv_drop();
    }

    action translate64(nv_ipv4_addr_t src_ip, nv_ipv4_addr_t dst_ip) {
        ip64_counter.count();
        user_meta.packet_len_field = headers.ipv6.payload_length;
        user_meta.packet_len_field += 20;
        user_meta.ip_protocol = headers.ipv6.next_header;
        user_meta.ttl = headers.ipv6.hop_limit;
        user_meta.tos_field = headers.ipv6.diffserv;
        user_meta.ecn_field = headers.ipv6.ecn;
        // partial TCP pseudo-header CRC calculation
        headers.ipv6.src_addr = 0;
        headers.ipv6.dst_addr = 0;
        // push a mostly empty IPv4 header with the partial checksum
        nv_push_headers(headers, v4DataTemplate,
            { },
            NvAnchor.IP_START_OF_HEADER,
            0, false, false);
        // remove IPv6 header without updates or reparse as packet is in mid-modification
        nv_pop_headers(headers, NvAnchor.IP_START_OF_HEADER, NvAnchor.L4_START_OF_HEADER,
            false, false);
        // change the ethertype to IPv4 and reparse to update L3 field offsets
        nv_set_l2_ethertype(headers, 16w0x0800);
        nv_force_reparse(headers);
        // Update the IP protocol and reparse to update L4 fields
        nv_set_ip_protocol(headers, user_meta.ip_protocol);
        nv_force_reparse(headers);
        // restore the remaining IP fields
        headers.ipv4.src_addr = src_ip;
        headers.ipv4.dst_addr = dst_ip;
        headers.ipv4.ttl = user_meta.ttl;
        headers.ipv4.total_len = user_meta.packet_len_field;
        nv_set_ip_ecn(headers, user_meta.ecn_field);
        nv_set_ip_dscp(headers, user_meta.tos_field);
    }

    table ipv6_to_ipv4 {
        key = {
            headers.ethernet.src_addr : exact;
        }
        actions = {
            translate64;
            NoAction;
        }
        default_action = NoAction;
        direct_counter = ip64_counter;
    }

    apply {
        if (std_meta.is_l4_ok == 1) {
            if (ipv6_to_ipv4.apply().hit) {
                nv_send_to_port(3);
            }
        }
    }
}

control nat46(
    inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout ip_data_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) ip46_counter;

      @nv_header_data_fields(
        nat_v6 = {
            // WORD 1
            version = 6,
```

```
            diffserv = 5,
            ecn = 0,
            flow_label = 0,
            // WORD 3
            payload_length = 0,
            // WORD 4
            next_header = 6,
            hop_limit = 64,
            // WORD 5
            src_addr = 0,    // must be zero for TCP pseudo-header CRC
            // WORD 9
            dst_addr = 0,    // must be zero for TCP pseudo-header CRC
        }
    )
    NvHeaderDataTemplate<v6_headers_t>() v6DataTemplate;

    action drop() {
        nv_drop();
    }

    action translate46(nv_ipv6_addr_t src_ip, nv_ipv6_addr_t dst_ip) {
        ip46_counter.count();
        user_meta.packet_len_field = headers.ipv4.total_len;
        user_meta.packet_len_field -= 20;
        user_meta.ip_protocol = headers.ipv4.protocol;
        user_meta.ttl = headers.ipv4.ttl;
        user_meta.tos_field = headers.ipv4.diffserv;
        user_meta.ecn_field = headers.ipv4.ecn;
        // partial TCP pseudo-header CRC calculation
        headers.ipv4.src_addr = 0;
        headers.ipv4.dst_addr = 0;
        // push a mostly empty IPv6 header and with the partial checksum
        nv_push_headers(headers, v6DataTemplate,
            { },
            NvAnchor.IP_START_OF_HEADER,
            0, false, false);
        // remove IPv4 header without updates or reparse as packet is in mid-modification
        nv_pop_headers(headers, NvAnchor.IP_START_OF_HEADER, NvAnchor.L4_START_OF_HEADER,
            false, false);
        // change the ethertype to IPv6 and reparse to update L3 field offsets
        nv_set_l2_ethertype(headers, 16w0x86dd);
        nv_force_reparse(headers);
        // Update the IP protocol and reparse to update L4 fields
        nv_set_ip_protocol(headers, user_meta.ip_protocol);
        nv_force_reparse(headers);
        // restore the remaining IP fields
        headers.ipv6.src_addr = src_ip;
        headers.ipv6.dst_addr = dst_ip;
        headers.ipv6.hop_limit = user_meta.ttl;
        headers.ipv6.payload_length = user_meta.packet_len_field;
        nv_set_ip_ecn(headers, user_meta.ecn_field);
        nv_set_ip_dscp(headers, user_meta.tos_field);
    }

    table ipv4_to_ipv6 {
        key = {
            headers.ethernet.src_addr : exact;
        }
        actions = {
            translate46;
            NoAction;
        }
        default_action = NoAction;
        direct_counter = ip46_counter;
    }

    apply {
        if (std_meta.is_l4_ok == 1) {
            if (ipv4_to_ipv6.apply().hit) {
                nv_send_to_port(3);
            }
        }
    }
}

control nat(inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout ip_data_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta) {

    nat64() v6_to_v4;
    nat46() v4_to_v6;

    apply {
        if (headers.ipv6.isValid()) {
            v6_to_v4.apply(headers, std_meta, user_meta, pkt_out_meta);
            nv_send_to_port(2);
        }
        else if (headers.ipv4.isValid()) {
            v4_to_v6.apply(headers, std_meta, user_meta, pkt_out_meta);
            nv_send_to_port(2);
        }
        // unknown traffic
        nv_send_to_port(1);
    }
}

NvDocaPipeline(
    nv_fixed_parser(),
    nat()
) main;
```

8.11.7.6.16 Add on Miss

This DPL program implements an efficient model for dynamic IPv6 flow tracking and statistics collection within the DPU's programmable data plane. By leveraging the **nv_add_entry** extern, the program autonomously inserts new entries into a match-action table based on observed traffic patterns, effectively offloading connection learning and state management from the control plane. Each invocation dynamically creates an entry keyed by programmable fields (e.g., the IPv6/TCP 5-tuple) assigns an action with parameters, and configures an optional timeout for expiration. This approach enables on-the-fly flow learning directly in hardware, optimizing real-time performance. The program adheres to the run-to-completion DPL architecture, ensuring that packets are parsed, matched, processed, and forwarded in a single logical pass through the pipeline.

8.11.7.6.16.1 Sample Code

IPv6 Flow Match Table Definition (fiveTupleTable_v6)

- Defines a match-action table keyed on a five-tuple corresponding to IPv6 TCP flows: source address, destination address, protocol (next header), source port, and destination port.
- Uses a direct counter (**NvDirectCounter**) to measure per-entry packet and byte statistics.
- Enables dynamic data-plane learning through **nv_support_add_entry** and time-based expiration with **nv_support_timeout** .
- The **nv_delayed_counter_stats** attribute ensures efficient updates to counters.

Table Actions (count_v6_hit)

- Implements an action that increments the per-entry counter and forwards the packet to a specified destination port.

Dynamic Entry Insertion (nv_add_entry)

- On table miss, the data plane uses **nv_add_entry** to dynamically install a new table entry into **fiveTupleTable_v6** .
- This entry is created using runtime key values extracted directly from the incoming packet headers (IPv6 and TCP fields).
- The entry associates the **count_v6_hit** action with parameters such as the forwarding port and sets a timeout of 1000 ms before aging out.
- This approach significantly reduces round-trips to the P4Runtime controller for flow installation, achieving fast insertion of new flows.

Apply Logic (Control Flow)

- The **apply** block orchestrates the overall packet-processing sequence:
  a. Verifies that the packet includes valid IPv6 headers.
  b. Applies the match table to attempt flow lookup.
  c. On a miss, invokes **nv_add_entry** to install the flow rule dynamically.

d. Forwards all packets to a predefined logical port (**nv_send_to_port(1)**), representing a final egress or next processing stage.

See below for the complete DPL example.

```
/*
 * SPDX-FileCopyrightText: Copyright (c) 2025 NVIDIA CORPORATION & AFFILIATES. All rights reserved.
 * SPDX-License-Identifier: LicenseRef-NvidiaProprietary
 *
 * NVIDIA CORPORATION, its affiliates and licensors retain all intellectual
 * property and proprietary rights in and to this material, related
 * documentation and any modifications thereto. Any use, reproduction,
 * disclosure or distribution of this material and related documentation
 * without an express license agreement from NVIDIA CORPORATION or
 * its affiliates is strictly prohibited.
 */

#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>

// Pipeline section
control c(
    inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) v6tableDirectCounter;

    action count_v6_hit(nv_logical_port_t port) {
        v6tableDirectCounter.count();
        nv_send_to_port(port);
    }

    table fiveTupleTable_v6 {
        key = {
            headers.ipv6.src_addr : exact;
            headers.ipv6.dst_addr : exact;
            headers.ipv6.next_header : exact;
            headers.tcp.src_port : exact;
            headers.tcp.dst_port : exact;
        }
        actions = {
            count_v6_hit;
            NoAction;
        }
        default_action = NoAction;
        direct_counter = v6tableDirectCounter;
        nv_support_add_entry = true;
        nv_support_timeout = true;
        nv_delayed_counter_stats = true;
    }

    apply {
        if (headers.ipv6.isValid() && headers.tcp.isValid()) {
            if (fiveTupleTable_v6.apply().miss) {
                nv_add_entry(
                    "fiveTupleTable_v6",
                    {
                        headers.ipv6.src_addr,
                        headers.ipv6.dst_addr,
                        NV_TCP_PROTOCOL,
                        headers.tcp.src_port,
                        headers.tcp.dst_port
                    },
                    "count_v6_hit",
                    {
                        32w1
                    },
                    1000
                );
            }
        }
        nv_send_to_port(1);
    }
}

// Instantiate the top-level DPU pipeline package
NvDocaPipeline(
    nv_fixed_parser(),
    c()
) main;
```

8.11.7.6.17 PSP Encryption

This DPL program implements the PSP security protocol to provide line-rate cryptographic protection and encapsulation for IPv4 packets. By leveraging BlueField's PSP hardware primitives, the program offloads encryption and decryption tasks, establishing secure, transparent tunnels

between endpoints. The pipeline dynamically distinguishes between outbound (TX) encryption and inbound (RX) decryption paths based on the ingress port, incorporating integrated error handling for cryptographic failures. Flow identifiers are used within match-action tables to select the appropriate keys, tunnel parameters, or remediation actions. Notably, PSP key exchange is out of scope for this DPL program and is assumed to have been completed prior to pipeline execution.

8.11.7.6.17.1 Sample Code

Direction Selection Stage

- Purpose: Directs each packet through either encryption or decryption logic.
- Table `direction_table`: Matches ingress port against configured "wire port" numbers, distinguishing between RX (decap) and TX (encap) path.
- Logic:
  - Miss: Packet is from the host/network stack, send it through encapsulation/encryption logic.
  - Hit: Packet is from the wire, run through decapsulation/decryption logic.

PSP Encapsulation/Encrypt Stage

- Purpose: Used for TX/network-bound packets, encapsulates and encrypts matching flows.
- Table `encap_table`: Matches outgoing IPv4 flows using source and destination address. For matched flows, triggers encapsulation with given MACs, tunnel IPs, and Security Association—applies DOCA PSP tunnel header and performs hardware encryption.
- Actions: The `encap` action builds the outer encapsulation (Ethernet, IPv4, UDP, PSP), sets keys/crypt parameters, and calls the hardware to encrypt payload.
- Packet Output: After encapsulation, packets are sent to the wire (external port) if matched, or to a default "other" port if not.

PSP Decrypt/Decapsulation Stage

- Purpose: Used for RX/wire-to-host direction, decompresses, decrypts, and validates PSP-encrypted packets.
- Decryption: If PSP header is present, calls hardware decryption with flow state defined in the hardware.
- Table `decap_table`: Matches on PSP SPI field. For matched packets, runs the `decap` action which strips outer tunnel and restores the original packet.
- Syndrome/Error Handling: Table `syndrome` matches on decryption error syndromes (ICV, bad trailer/context, general error). For recognized errors, counts and drops the packet; otherwise, processing continues.
- Packet Output: Successfully decrypted/decapsulated packets go to a designated internal port; others are sent to a fallback port.

See below for the complete DPL example.

```
1  /*
2   * SPDX-FileCopyrightText: Copyright (c) 2025 NVIDIA CORPORATION & AFFILIATES. All rights reserved.
3   * SPDX-License-Identifier: LicenseRef-NvidiaProprietary
4   *
5   * NVIDIA CORPORATION, its affiliates and licensors retain all intellectual
6   * property and proprietary rights in and to this material, related
7   * documentation and any modifications thereto. Any use, reproduction,
8   * disclosure or distribution of this material and related documentation
```

```
 * without an express license agreement from NVIDIA CORPORATION or
 * its affiliates is strictly prohibited.
 */
#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>

const bit<32> WIRE_PORT = 0;
const bit<16> PSP_UDP_SRC_PORT = 1234;

NvPSP(1024) psp_crypto;

control psp_encap(
    inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    action encap_tunnel(
        nv_mac_addr_t  dst_addr,
        nv_mac_addr_t  src_addr,
        nv_ipv4_addr_t src_ip,
        nv_ipv4_addr_t dst_ip,
        bit<32> spi
    ) {
        nv_encap_psp_v4_tunnel(headers, dst_addr, src_addr, src_ip, dst_ip,
            PSP_UDP_SRC_PORT, // udp_src_port,
            NvIPEncapProtocol.IPV4, // next header
            6w0, // crypt_offset
            0, // needs_sampling
            0, // drop
            NvPSPVersion.V1_AES_GCM_256,
            spi,
            0 // init_vector
        );
    }
    action encrypt_tunnel(
        nv_mac_addr_t  dst_addr,
        nv_mac_addr_t  src_addr,
        nv_ipv4_addr_t src_ip,
        nv_ipv4_addr_t dst_ip,
        bit<32> spi,
        bit<32> sa_index
    ) {
        encap_tunnel(dst_addr, src_addr, src_ip, dst_ip, spi);
        psp_crypto.encrypt(headers, sa_index);
    }
    action encap_transport(bit<32> spi
    ) {
        nv_encap_psp_transport(headers,
            PSP_UDP_SRC_PORT, // udp_src_port,
            6w0, // crypt_offset
            0, // needs_sampling
            0, // drop
            NvPSPVersion.V1_AES_GCM_256,
            spi,
            0 // init_vector
        );
    }
    action encrypt_transport(
        bit<32> spi,
        bit<32> sa_index
    ) {
        encap_transport(spi);
        psp_crypto.encrypt(headers, sa_index);
    }
    table encap_table {
        key = {
            headers.ipv4.src_addr : exact;
            headers.ipv4.dst_addr : exact;
        }
        actions = {
            encrypt_tunnel;
            encap_tunnel;
            encrypt_transport;
            encap_transport;
            NoAction;
        }
        default_action = NoAction();
    }
    apply {
        if (encap_table.apply().hit) {
            nv_send_to_port(WIRE_PORT);
        } else {
            // This packet is not PSP
            nv_send_to_port(3);
        }

    }
}

control psp_decap(
    inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvCounter(8, NvCounterType.PACKETS_AND_BYTES) syndrome_counter;

    action decrypt() {
        psp_crypto.decrypt(headers);
    }

    action decap(
```

```
        nv_mac_addr_t dst_addr,
        nv_mac_addr_t src_addr
    ) {
        nv_decap_psp_tunnel(headers, dst_addr, src_addr,
            NvPspInnerProtocolType.IPV4);
    }

    table decap_table {
        key = {
            headers.psp.security_parameters_index : exact;
        }
        actions = {
            decap;
            NoAction;
        }
        default_action = NoAction();
    }

    action drop(nv_psp_status_t status) {
        syndrome_counter.count(status);
        nv_drop();
    }

    table syndrome {
         key = {
            std_meta.psp_syndrome : exact;
        }
        actions = {
            drop;
            NoAction;
        }
        default_action = NoAction();
        const entries = {
            (NvPSPStatus.ICV_FAIL) : drop(NvPSPStatus.ICV_FAIL);
            (NvPSPStatus.BAD_TRAILER) : drop(NvPSPStatus.BAD_TRAILER);
            (NvPSPStatus.BAD_CONTEXT) : drop(NvPSPStatus.BAD_CONTEXT);
            (NvPSPStatus.GENERAL_ERROR) : drop(NvPSPStatus.GENERAL_ERROR);
        }
    }

    apply {
        if (headers.psp.isValid()) {
            decrypt();
            if (syndrome.apply().miss) {
                decap_table.apply();
                nv_send_to_port(1);
            }
        }
        // it is not PSP
        nv_send_to_port(2);
    }
}

control psp(
    inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    psp_decap() decap;
    psp_encap() encap;

    /* user should add entries that correspond to the wire ports
     * A hit means this is an RX packet, miss means a TX packet
    */
    table direction_table {
        key = {
            std_meta.ingress_port : exact;
        }
        actions = {
            NoAction;
        }
        default_action = NoAction();
        const entries = {
            (WIRE_PORT) : NoAction();
        }
    }

    apply {
        if (direction_table.apply().miss) {
            // TX direction
            encap.apply(headers, std_meta, user_meta, pkt_out_meta);
        }
        else {
            // RX direction
            decap.apply(headers, std_meta, user_meta, pkt_out_meta);
        }
    }
}

NvDocaPipeline(
    nv_fixed_parser(),
    psp()
) main;
```

8.11.7.7 Inspecting DPL Programs

This page describes the DPL Nspect tool, which is used to inspect DPL programs loaded onto the NVIDIA® BlueField® DPU.

8.11.7.7.1 1. Introduction

The DOCA Dependency Compatibility Policy is designed to provide customers with high velocity and flexibility during updates, enabling them to update specific layers of their system without requiring a full stack overhaul.

The primary objectives of this policy are to support decoupled updates for teams managing separate firmware and software stacks, allow for targeted upgrades on specific devices, minimize maintenance downtime by updating only necessary components, and guarantee investment protection by ensuring application continuity across platform releases.

This policy applies to all NVIDIA network device products and their associated software and firmware stacks.

8.11.7.7.2 2. Release Notes

8.11.7.7.2.1 2.1. New Features

- Externs Command: This new command replaces the per-extern commands (**counters** , **meters** , **tcpstates**) with a single **externs** command. Use the **types** argument to filter by extern type, including selecting only direct or indirect externs.
- Memory Extern: Added support for querying the **memory** extern type with the **externs** command.
- IPsec Extern: (BETA) Added support for querying the **ipsec** extern type with the **externs** command.
- Additional Results Hint: Commands that support batching and max results now show a hint when more results are available beyond what was returned, so you can decide whether to increase the batch size or max results accordingly.
- New Performance Stats:
  - Added per-table entry modification statistics (add / modify / delete counts and rates) to the **perf** command.
  - Added per-table ternary runtime-optimization (RTO) statistics to the **perf** command, reported when the table's cache is enabled.
- Filter Improvements: Added support for P4 objects alias names. You can now reference and filter P4 objects using their optional **p4_alias** .
- Nspect Textual UI App: (BETA) Introduced a new Textual-based interactive UI for **dpl_nspect** . Launch it with the **--tui** flag for an interactive experience. The traditional headless CLI mode remains available and is the default mode.

8.11.7.7.2.2 2.2. Fixes

- The **meter** extern's **initial_color** was not populated in related commands.
- Extern filtering by name of direct externs.
- **perf** command truncating long numeric values with an ellipsis.
- **perf** command failing when running against an RT Service with Packet I/O disabled.
- Race condition that could cause intermittent query failures when new entries were being added concurrently.

8.11.7.7.2.3 2.3. Other

- Replaced **pickle** with **msgpack** package for data serialization to address a security concern.
- Log files are now automatically rotated to prevent unbounded disk usage, and the verbosity level remains configurable.

8.11.7.7.3 3. DPL Nspect Execution

To view basic usage information, run the **dpl nspect** without arguments. For a detailed usage guide, use the **-h** or **--help** flags. You can also view command-specific help by running **dpl nspect [command] -h**.

8.11.7.7.3.1 3.1. Prerequisites and Configuration

- Server Connection: DPL Nspect requires a connection to a host (local or remote) where the DPL Nspect Server is running as part of the DPL Runtime Service.
- Address Configuration: The host address and port can be configured in two ways:
  a. Directly using the **-a** / **--address** argument.
  b. By setting the **DPL_NSPECT_SERVER_ADDRESS** environment variable (e.g., **DPL_NSPECT_SERVER_ADDRESS=127.0.0.1:9560**).

  > ⚠ Hostnames are resolved at runtime (e.g., **localhost** resolves to **127.0.0.1**).

- Data Directory: Configuration and logs are stored in the Data directory.
- Refer to the "DPL Service Configuration" for a TLS setup example

8.11.7.7.3.2 3.2. Command Usage

```
usage: dpl_nspect [-h] [-v] [-nh] [-ll] -a ADDRESS [-t TIMEOUT] [--tui] [--cacert PATH_TO_CA_CERT] [--cert
PATH_TO_CERT]
                  [--private-key PATH_TO_KEY]
                  {system-info,devices,tables,query,externs,graph,config,hw-steering,debug,perf} ...

options:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  -nh, --no-hints       Disable hints for further information.
                        You may set the DPL_NSPECT_DISABLE_HINTS environment variable instead of using this flag:
                        "export DPL_NSPECT_DISABLE_HINTS=1" (default: False)
  -ll, --low-level      Display low level internal information. (default: False)
  -a ADDRESS, --address ADDRESS
                        The dpl_nspect server socket address of the form IPv4/IPv6* [address][:port]
                        IPv6*: To use with a port the address must be enclosed in literal square brackets.
                        Example: ipv6:[2607:f8b0:400e:c00::ef]:9560 or ipv6:[::]:9560
                        The address/port can be assigned to and read from the environment variable:
DPL_NSPECT_SERVER_ADDRESS. (default: None)
  -t TIMEOUT, --timeout TIMEOUT
                        [Optional] gRPC channel connectivity timeout. (default: 0)
  --tui                 Run the application in TUI mode. (default: False)

commands:
  For further info run dpl_nspect <command> -h

  {system-info,devices,tables,query,externs,graph,config,hw-steering,debug,perf}
    system-info         Display system information.
    devices             List the configured virtual devices.
    tables              List tables details.
    query               List table entries, keys and actions of the loaded program.
    externs             List externs details and values.
    graph               Generate a graph using DOT language.
                        The generated graph can be rendered using DOT renderers, such as online DOT graphics
visualizer websites.
    config              Get/Set the device configuration.
    hw-steering         Display HW Steering rules dump.
    debug               Start a debug recording session.
    perf                Display live performance monitor statistics.
```

```
Authentication:
  Mutually inclusive arguments for secure connection using SSL authentication

  --cacert PATH_TO_CA_CERT
                        CA certificate path for secure connections (default: None)
  --cert PATH_TO_CERT   Client certificate path for mutual authentication (default: None)
  --private-key PATH_TO_KEY
                        Client private key path for mutual authentication (default: None)

Data directory: /root/.local/share/dpl_devtools
```

8.11.7.7.4 4. DPL Nspect Commands and Flags

The following sections describe specific commands and their outputs.

> ⚠ The DPL program **hello_packet.p4** (referenced at the end of this section) was used to generate the examples below.

> ✓ Most commands support the **--json** argument to output data in JSON format, facilitating integration with automated scripts.

8.11.7.7.4.1 4.1. system-info

```
dpl nspect system-info
```

Displays system information.

Example output:

System Information

Key	Value
Linux OS version	"Ubuntu 24.04.4 LTS"
DPL Runtime Service version	Not found
DOCA version	3.4.0072-1
RDMA Core Version	2604.0.5-1
OFED Version	OFED-internal-26.04-0.4.0:
Debug Schema Version	Device ID: 1000, doctype: /nvp4c/devtools/v20, target: doca-bf3

8.11.7.7.4.2 4.2. devices

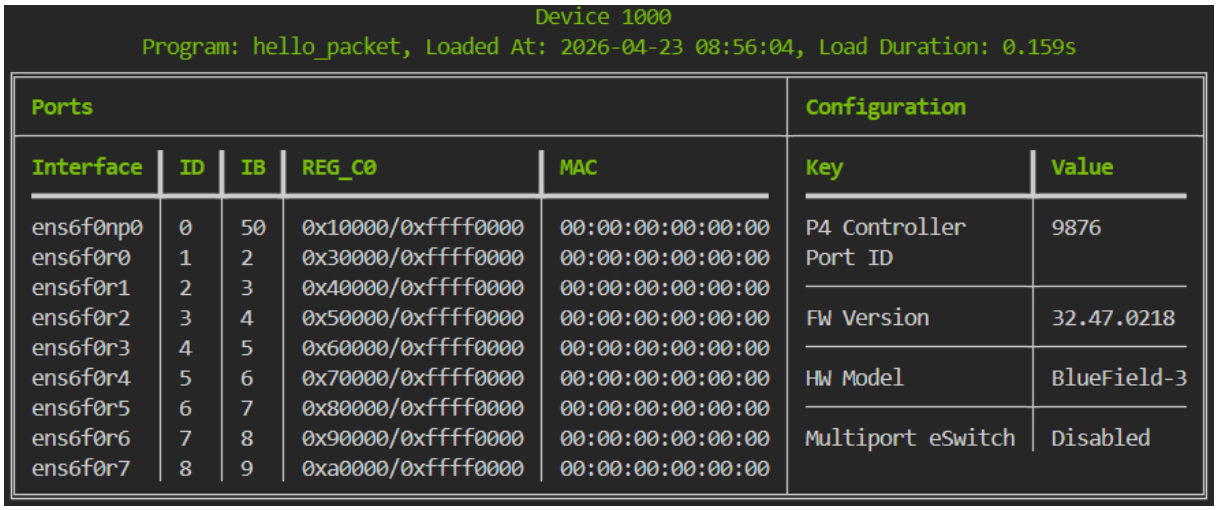
```
dpl nspect devices
```

Lists all configured DPL devices, displaying a table for each device.

- Title: Contains program details only if a program is loaded.

- Interface: The interface name as defined in the DP device configuration file.
- ID: The port ID as defined in the DPL device configuration file.
- IB: The physical InfiniBand port ID used for the DPL port ID.

Example output:



Device 1000
Program: hello_packet, Loaded At: 2026-04-23 08:56:04, Load Duration: 0.159s

Ports

Interface	ID	IB	REG_C0	MAC
ens6f0np0	0	50	0x10000/0xffff0000	00:00:00:00:00:00
ens6f0r0	1	2	0x30000/0xffff0000	00:00:00:00:00:00
ens6f0r1	2	3	0x40000/0xffff0000	00:00:00:00:00:00
ens6f0r2	3	4	0x50000/0xffff0000	00:00:00:00:00:00
ens6f0r3	4	5	0x60000/0xffff0000	00:00:00:00:00:00
ens6f0r4	5	6	0x70000/0xffff0000	00:00:00:00:00:00
ens6f0r5	6	7	0x80000/0xffff0000	00:00:00:00:00:00
ens6f0r6	7	8	0x90000/0xffff0000	00:00:00:00:00:00
ens6f0r7	8	9	0xa0000/0xffff0000	00:00:00:00:00:00

Configuration

Key	Value
P4 Controller Port ID	9876
FW Version	32.47.0218
HW Model	BlueField-3
Multiport eSwitch	Disabled

8.11.7.7.4.3 4.3. tables

```
dpl nspect tables [--json]
```

Lists all tables defined in the DPL program.

- ID: The P4 runtime ID of the object as defined in the **p4info.txt** file.
- Direct Externs: Lists the names of all direct extern instances defined for the table (e.g., direct counter, direct meter).

Output example:

hello_packet.forward_table - 50144908

Match Fields			Actions			Direct Externs	Entries Count	Source Location	Supports Timeout
ID	Name	Type	ID	Name	Params				
1	headers.ethernet.src_addr	exact	19510525	hello_packet.drop		hello_packet.hello_direct_counter hello_packet.hello_direct_meter	6	hello_packet.p4:45:11	False
			19237259	hello_packet.forward	ID: 1, Name: port, Size: 32				
			21257015	NoAction					

The **--json** flag provides the output in JSON format. For example:

```
[
  {
    "title": {
      "Table ID": 40048916,
      "Table Name": "tbl_drop"
    },
    "rows": [
      {
```

```
        "Keys": {
          "title": {},
          "rows": []
        },
        "Actions": {
          "title": {},
          "rows": [
            {
              "ID": "19510525",
              "Name": "hello_packet.drop",
              "Params": ""
            }
          ]
        },
        "Direct Externs": [],
        "Entries Count": "1",
        "Source Location": "None",
        "Supports Timeout": "False"
      }
    ]
  },
  {
    "title": {
      "Table ID": 50144908,
      "Table Name": "hello_packet.forward_table"
    },
    "rows": [
      {
        "Keys": {
          "title": {},
          "rows": [
            {
              "ID": "1",
              "Name": "headers.ethernet.src_addr",
              "Type": "exact"
            }
          ]
        },
        "Actions": {
          "title": {},
          "rows": [
            {
              "ID": "19510525",
              "Name": "hello_packet.drop",
              "Params": ""
            },
            {
              "ID": "19237259",
              "Name": "hello_packet.forward",
              "Params": {
                "title": {},
                "rows": [
                  {
                    "ID": "1",
                    "Name": "port",
                    "Size": "32"
                  }
                ]
              }
            },
            {
              "ID": "21257015",
              "Name": "NoAction",
              "Params": ""
            }
          ]
        },
        "Direct Externs": [
          "hello_packet.hello_direct_counter",
          "hello_packet.hello_direct_meter"
        ],
        "Entries Count": "6",
        "Source Location": "hello_packet.p4:46",
        "Supports Timeout": "False"
      }
    ]
  }
]
```


8.11.7.7.4.4 4.4. query

```
dpl nspect query
```

Lists all entries currently contained in each table.

- Entry ID: The runtime ID assigned to the entry at creation time (not a P4 context ID).
- Keys: If no key is displayed, it indicates a default entry.
- Extern entries: Displays data for extern entries associated with the table entry (result of an explicit extern call within the Action).

Example output:

hello_packet.forward_table - 50144908

Entry ID [Priority]	Match	Action	Externs
5[0]	headers.ethernet.src_addr == 0xbbbbbbbbbb (exact)	NoAction()	
4[0]	headers.ethernet.src_addr == 0xaaaaaaaaaa (exact)	NoAction()	
3[0]	headers.ethernet.src_addr == 0xdddddddddd (exact)	hello_packet.drop()	hello_packet.hello_counter[1] = (packets=0, bytes=0)
2[0]	headers.ethernet.src_addr == 0x2222222222 (exact)	hello_packet.forward(port=0x2)	hello_packet.hello_direct_counter[2] = (packets=0, bytes=0)
1[0]	headers.ethernet.src_addr == 0x1111111111 (exact)	hello_packet.forward(port=0x1)	hello_packet.hello_direct_counter[1] = (packets=0, bytes=0)
0[0]		hello_packet.forward(port=0x3)	hello_packet.hello_direct_counter[0] = (packets=0, bytes=0)

8.11.7.7.4.5 4.5. externs

```
dpl nspect externs
```

Lists all externs in the DPL program (one table per extern instance???).

- Indirect Externs: First column shows Index.
- Direct Externs: First column shows Table Entry ID (corresponds to the ID seen in the **query** command).

NvDirectCounter(Pkts/Bts) hello_packet.hello_direct_counter - 320479669

Table Entry ID	Packets	Bytes
5	0	0
4	0	0
3	0	0
2	0	0
1	0	0
0	0	0

NvCounter(4, Pkts/Bts) hello_packet.hello_counter - 307179965

Index	Packets	Bytes
0	0	0

1	0	0
2	0	0
3	0	0

NvDirectMeterPeakTrTCM(128), Pkts hello_packet.hello_direct_meter - 359616562

Table Entry ID	CIR	CBS	PIR	PBS	Initial Color
5	1	2	3	4	UNDEFINED
4	1	2	3	4	UNDEFINED
3	1	2	3	4	UNDEFINED
2	1	2	3	4	UNDEFINED
1	1	2	3	4	UNDEFINED
0	1	2	3	4	UNDEFINED

NvTcpState(4) hello_packet.hello_tcp - 2319426381

Index	Last Direction	Last ACK Number	Last SEQ Number
0	ORIGINAL	0	0
1	ORIGINAL	0	0
2	ORIGINAL	0	0
3	ORIGINAL	0	0

The **--types** argument allows querying specific extern categories (counter, meter, tcp_state, memory) and/or scope (direct for table-bound, indirect for standalone).

```
dpl nspect externs --types counter,meter,direct
```

NvDirectCounter(Pkts/Bts) hello_packet.hello_direct_counter - 320479669

Table Entry ID	Packets	Bytes
5	0	0
4	0	0
3	0	0
2	0	0
1	0	0
0	0	0

NvDirectMeterPeakTrTCM(128), Pkts hello_packet.hello_direct_meter - 359616562

Table Entry ID	CIR	CBS	PIR	PBS	Initial Color
5	1	2	3	4	UNDEFINED
4	1	2	3	4	UNDEFINED
3	1	2	3	4	UNDEFINED
2	1	2	3	4	UNDEFINED
1	1	2	3	4	UNDEFINED
0	1	2	3	4	UNDEFINED

8.11.7.7.4.6 4.6. graph

```
dpl nspect --low-level graph
```

Generates a DOT graph representing all potential packet paths within the pipeline.

Example output:

```
digraph hello_packet_pipeline {
    graph [compound=true nodesep=0.1]
    node [height=0 width=0]
    // Legend
    subgraph cluster_legend {
        graph [fontsize="6pt" label=Legend]
        control [label=Control color=blue fontsize="6pt" id=control shape=rectangle]
        table [label=Table color=orange fontsize="6pt" id=table shape=rectangle]
        ll_table [label=<<B>Low Level Table ID</B><BR/>[Origin(s)]> fontsize="6pt" id=ll_table shape=diamond]
        legend_invis [height=0 id=legend_invis shape=point style=invis width=0]
        ll_root_table [label=<<B>Root LL Table</B>> color=green fontsize="6pt" id=ll_root_table shape=diamond]
        ll_unreachable_table [label=<<B>Unreachable LL Table</B>> color=red fontsize="6pt" id=ll_unreachable_table
shape=diamond]
        default_rule_invis_tail [fontsize="6pt" height=0 id=default_rule_invis_tail shape=point style=invis
width=0]
        default_rule_invis_head [fontsize="6pt" height=0 id=default_rule_invis_head shape=point style=invis
width=0]
```

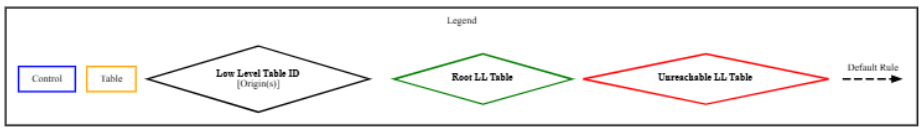
```
        default_rule_invis_tail -> default_rule_invis_head [label="Default Rule" arrowsize="0.5px" fontsize="6pt"
id="default_rule_invis_tail-default_rule_invis_head" style=dashed]
        subgraph rank_0 {
            rank=same
            control [id=control]
            default_rule_invis_head [id=default_rule_invis_head]
            default_rule_invis_tail [id=default_rule_invis_tail]
            ll_root_table [id=ll_root_table]
            ll_table [id=ll_table]
            ll_unreachable_table [id=ll_unreachable_table]
            table [id=table]
        }
    }
    // Pipeline Stage: main
    subgraph cluster_hello_packet {
        graph [color=blue label=hello_packet shape=rectangle]
        legend_invis [height=0 id=legend_invis shape=point style=invis width=0]
        "51[4]" [label=<<B>51[4]</B><BR/>[hello_packet]> color=green id="51[4]" shape=diamond]
        "42[2]" [label=<<B>42[2]</B><BR/>[hello_packet, hello_packet.forward_table]> id="42[2]" shape=diamond]
        DROP [fontcolor=red id=DROP shape=box]
        "47[3]" [label=<<B>47[3]</B><BR/>[hello_packet]> id="47[3]" shape=diamond]
        legend_invis -> "51[4]" [id="legend_invis-51[4]" style=invis]
        "51[4]" -> "42[2]" [id="51[4]-42[2]"]
        "42[2]" -> DROP [id="42[2]-DROP"]
        "51[4]" -> "47[3]" [id="51[4]-47[3]" style=dashed]
        "47[3]" -> DROP [id="47[3]-DROP"]
        subgraph rank_0 {
            rank=same
            "51[4]" [id="51[4]"]
        }
        subgraph rank_1 {
            rank=same
            "42[2]" [id="42[2]"]
            "47[3]" [id="47[3]"]
        }
        subgraph rank_2 {
            rank=same
            DROP [id=DROP]
        }
    }
}
```

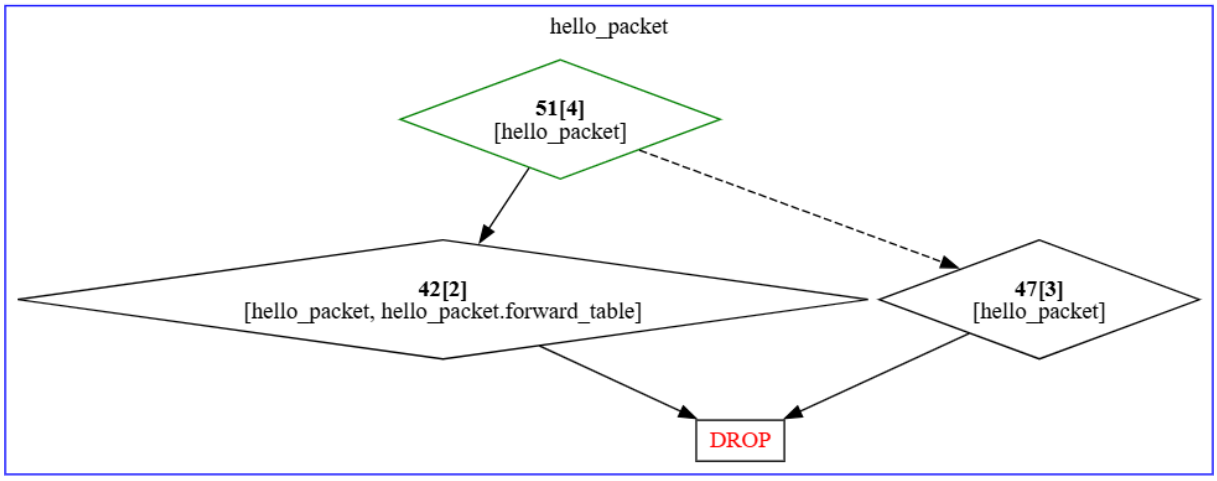
```
dpl nspect --low-level graph --url
```

The **`--url`** flag generates an HTTP URL link to render the graph using GraphvizOnline.

Legend:

- Low Level Table (LL Table): Refers to the internal HWS Table object created from loading the DPL program.
- Root LL Table: The starting point of the generated pipeline.





8.11.7.7.4.7 4.7. debug

Starts a debug session. Packets sent during this session are traced and recorded to an output archive, which can be inspected using the DPL Debugger.

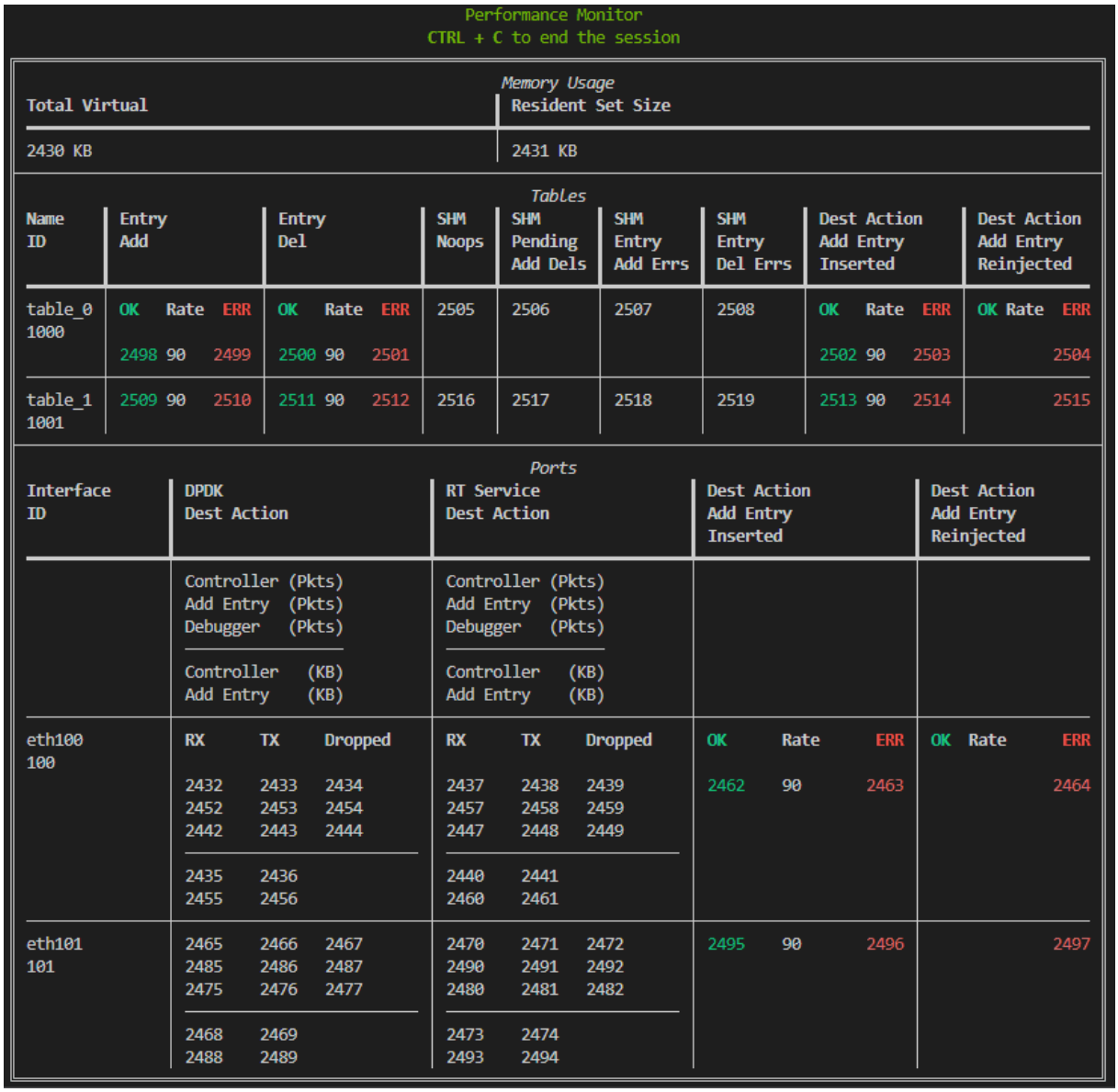
```
dpl nspect debug -o hello_packet.tar.gz
```

8.11.7.7.4.8 4.8. perf

```
dpl nspect perf
```

Displays real-time performance statistics.

Example output:



The following subsections break down the statistics provided in the example output.

4.8.1. Memory Usage Statistics

Displays the real-time memory consumption of the DPL Runtime Service.

Metric	Description
Total Virtual Memory	The total amount of virtual memory used by the service.
Resident Set Size	The amount of physical memory (RAM) currently being used (in KB).

4.8.2. Table Entry Operations Statistics

Tracks operations for each DPL table (one row per table).

Category	Column	Description	Notes
Entry Add/Del	OK	Total number of successful additions or deletions	Tracks *all* table entry operations.
	ERR	Total number of failed additions or deletions	
	Rate	The difference in successful operations between the current and previous sample	
SHM	Noops	Count of entries where a delete was requested before the addition was fully processed	**Verbose mode only.** Tracks *entries*. Only for SHM-enabled tables
	Pending Add Dels	Count of delete operations received for entries that are still pending hardware confirmation	**Verbose mode only.** Tracks *operations*
	Entry Add/Del Errs	Total number of failed SHM entry additions or deletions	**Verbose mode only.** Tracks *operations*
Dest Action Add Entry	Inserted	Successful table entry additions triggered specifically by the *Add Entry* action	**Verbose mode only.** Tracks *table entries*
	Reinjected	Total number of *failed* reinjection attempts (ERR column).	**Verbose mode only.** Tracks *packets*. The OK/Rate columns are always empty

4.8.3. Port Statistics Statistics

Tracks packet flow through destination actions (one row per Port Interface ID).

Layer	Row/ Action	Description	Notes
DPDK Dest Action	N/A	Packet flow statistics at the hardware (DPDK) layer	
RT Service Dest Action	N/A	Packet flow statistics at the application (RT Service) layer	
Action Types	Controller	Flow statistics for packets destined for the Controller	
	Add Entry	Flow statistics for packets destined for the Add Entry action	Same metrics as the *Tables* section, but tracked per port
	Debugger	Flow statistics for packets destined for the Debugger	Only displays Packet Counts (no KB)
Flow Direction	RX / TX	Packets received or transmitted in this layer/ action	Displays both Packet Counts and KB
	Dropped	Packets dropped in this layer/action	Only displays Packet Counts (no KB)

8.11.7.7.4.9 4.9. hello_packet.p4

```
/*
 * SPDX-FileCopyrightText: Copyright (c) 2025 NVIDIA CORPORATION & AFFILIATES. All rights reserved.
 * SPDX-License-Identifier: LicenseRef-NvidiaProprietary
 *
 * NVIDIA CORPORATION, its affiliates and licensors retain all intellectual
```

```
 * property and proprietary rights in and to this material, related
 * documentation and any modifications thereto. Any use, reproduction,
 * disclosure or distribution of this material and related documentation
 * without an express license agreement from NVIDIA CORPORATION or
 * its affiliates is strictly prohibited.
 */

#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>

/*
 * This basic application demonstrates a simple match/action pipeline using NVIDIA
 * DOCA P4 Target Architecture.
 * - Match on destination MAC address
 * - Forward or drop the packet
 */

control hello_packet(
    inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvCounter(4, NvCounterType.PACKETS_AND_BYTES) hello_counter;
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) hello_direct_counter;
    NvDirectMeterPeakTrTCM(NvMeterUnits.PACKETS, 1, 2, 3, 4) hello_direct_meter;
    NvTcpState(4) hello_tcp;

    action drop() {
        hello_counter.count(1);
        nv_send_debug_pkt();
        nv_drop();
    }

    action forward(bit<32> port) {
        hello_direct_counter.count();
        nv_send_to_port(port);
    }

    table forward_table {
        key = {
            headers.ethernet.src_addr : exact;
        }
        actions = {
            drop;
            forward;
            NoAction;
        }
        default_action = forward(3);
        direct_meter = hello_direct_meter;
        direct_counter = hello_direct_counter;
        const entries = {
            (48w0x001111111111) : forward(1);
            (48w0x002222222222) : forward(2);
            (48w0x00dddddddddd) : drop();
            (48w0x00aaaaaaaaaa) : NoAction();
            (48w0x00bbbbbbbbbb) : NoAction();
        }
    }
    apply {
        hello_counter.count(0);
        if (headers.ipv4.isValid()) {
            forward_table.apply();
        }
        drop();
    }
}

NvDocaPipeline(
    nv_fixed_parser(),
    hello_packet()
) main;
```

8.11.7.8 Debugging DPL Programs

This section describes the DPL Debugger Tools, used to debug pipeline packets that DPL programs are processing in NVIDIA® BlueField®.

> ⚠ The debugger functions only with DPL programs compiled in these debug mode.

8.11.7.8.1 1. Introduction

The DPL Debugger is a graphical user interface (GUI) application used to debug network packets processed by DPL programs on NVIDIA® BlueField® devices. It allows developers to trace packet flows throughout a DPL program's pipeline, inspecting detailed states at each stage.

⚠ For details, see the documentation on the extern function nv_send_debug_pkt.

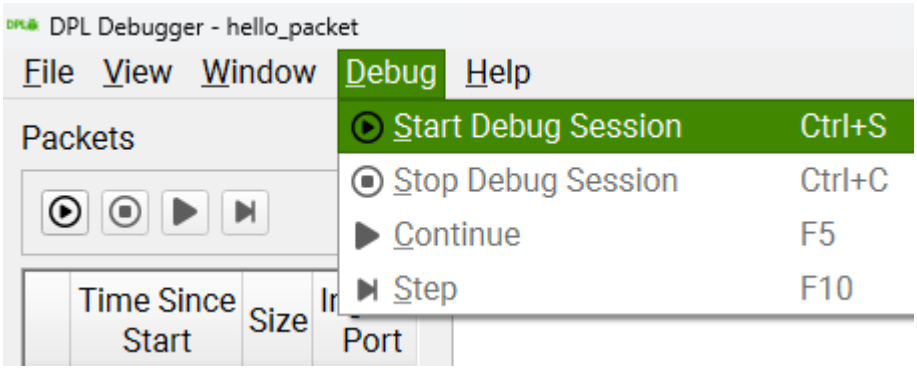
8.11.7.8.2 2. Launching the Debugger

You can launch the DPL Debugger to either run a live debug session or analyze a pre-recorded session file. Running `dpl debugger` from the command line with no arguments opens the standard GUI.

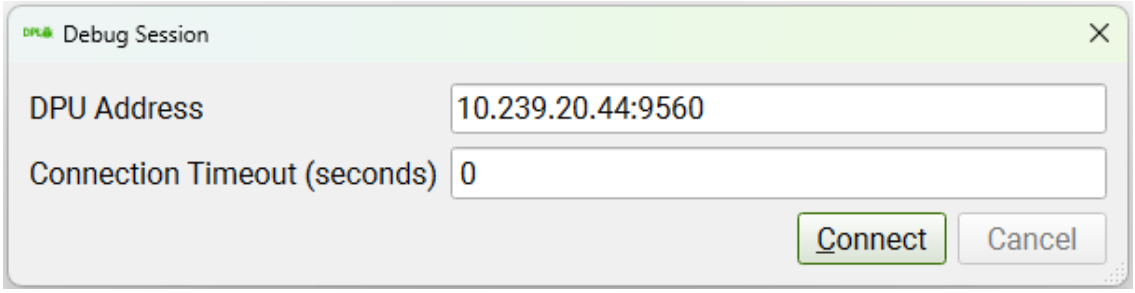
8.11.7.8.2.1 2.1. Option A: Live Debug Session

Running a live session allows you to process, inspect, and reinject packets at runtime.

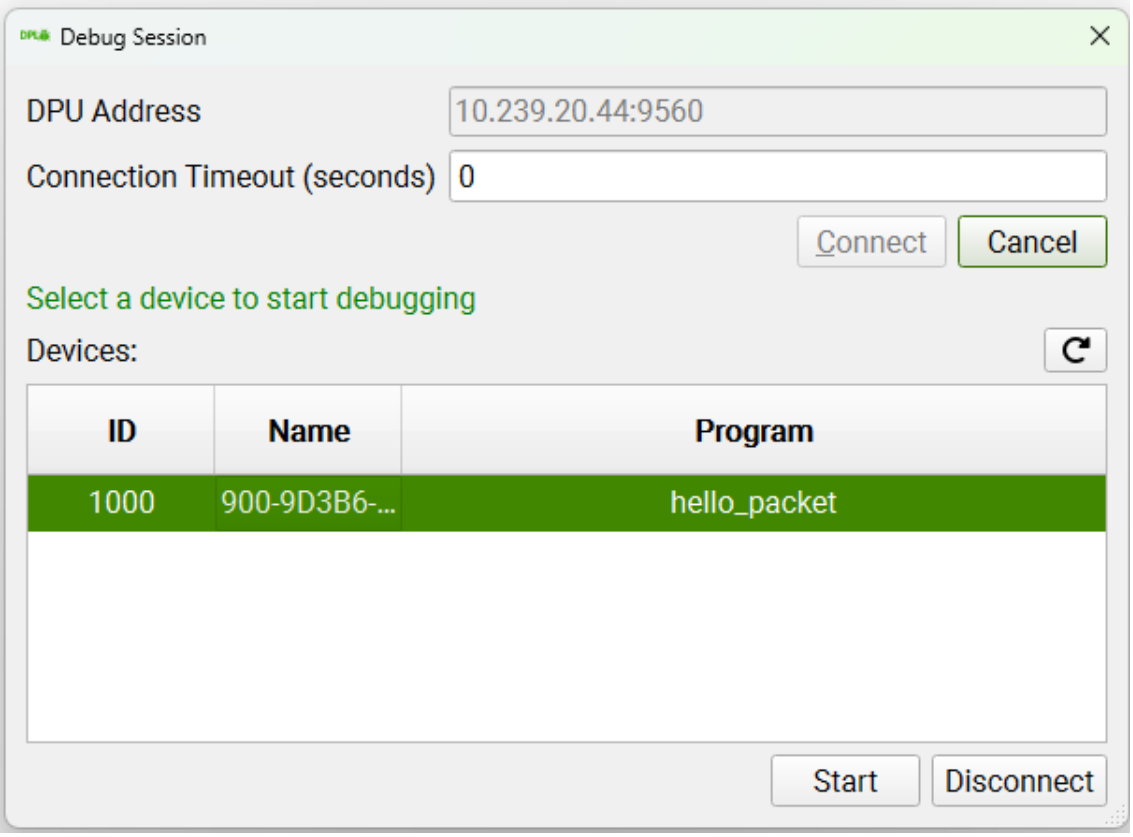
1. Click the Start Debug Session button, press Ctrl+S, or navigate to Debug → Start Debug Session.



2. Enter the DPU address, port (default is 9560), and connection timeout (use 0 for no timeout), then click Connect.



3. Select the desired device and click **Start**.



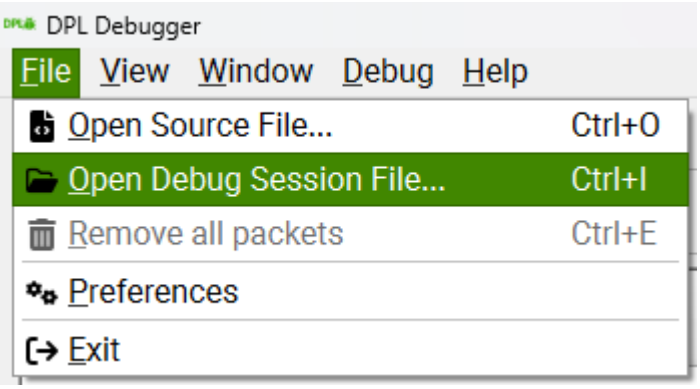
8.11.7.8.2.2 2.2. Option B: Pre-recorded Debug Session

Pre-recorded sessions are `.tar.gz` archives generated using the `nspect debug` command.

- Via CLI: Specify the archive path when launching the tool:

```
dpl debugger hello_packet.tar.gz
```

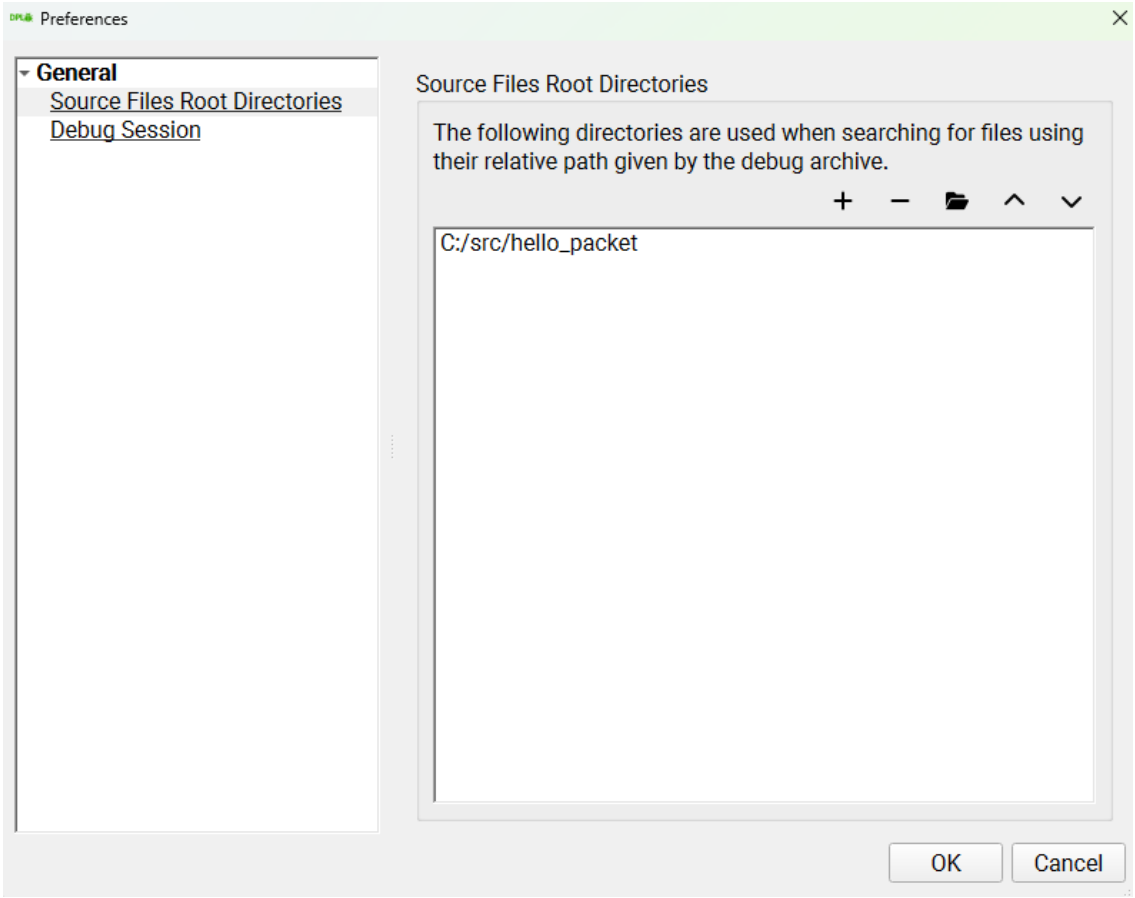
- Via GUI: Click **File** → **Open Debug Session File** and select the archive.



8.11.7.8.3 3. Setting Source Directories

To ensure the debugger can display the relevant code when a debug point is hit, configure the location of your DPL program sources.

1. Navigate to File → Preferences.



2. Set the search paths for your DPL source files.

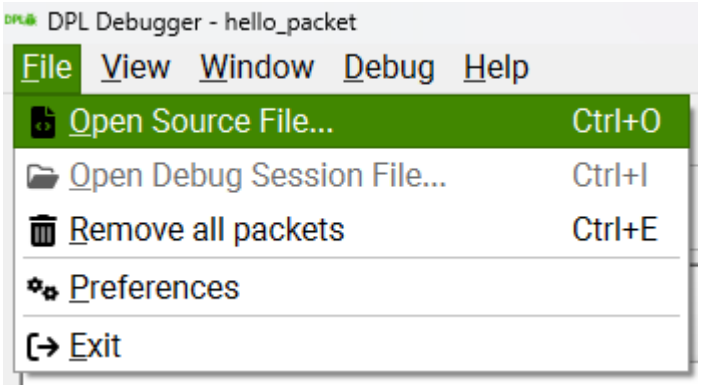
8.11.7.8.4 4. Debug Workflow

8.11.7.8.4.1 4.1. Enabling Debug Points

When a live session starts, all debug points in the program are disabled by default. Packets flow through the pipeline unaffected until a debug point is enabled.

To enable a debug point:

1. Open the source file within the debugger (File → Open Source File).



2. Click the left margin next to the desired line, or place your caret on the line and press F9.

> If a line does not have an associated debug point, the next available debug point will be enabled instead.

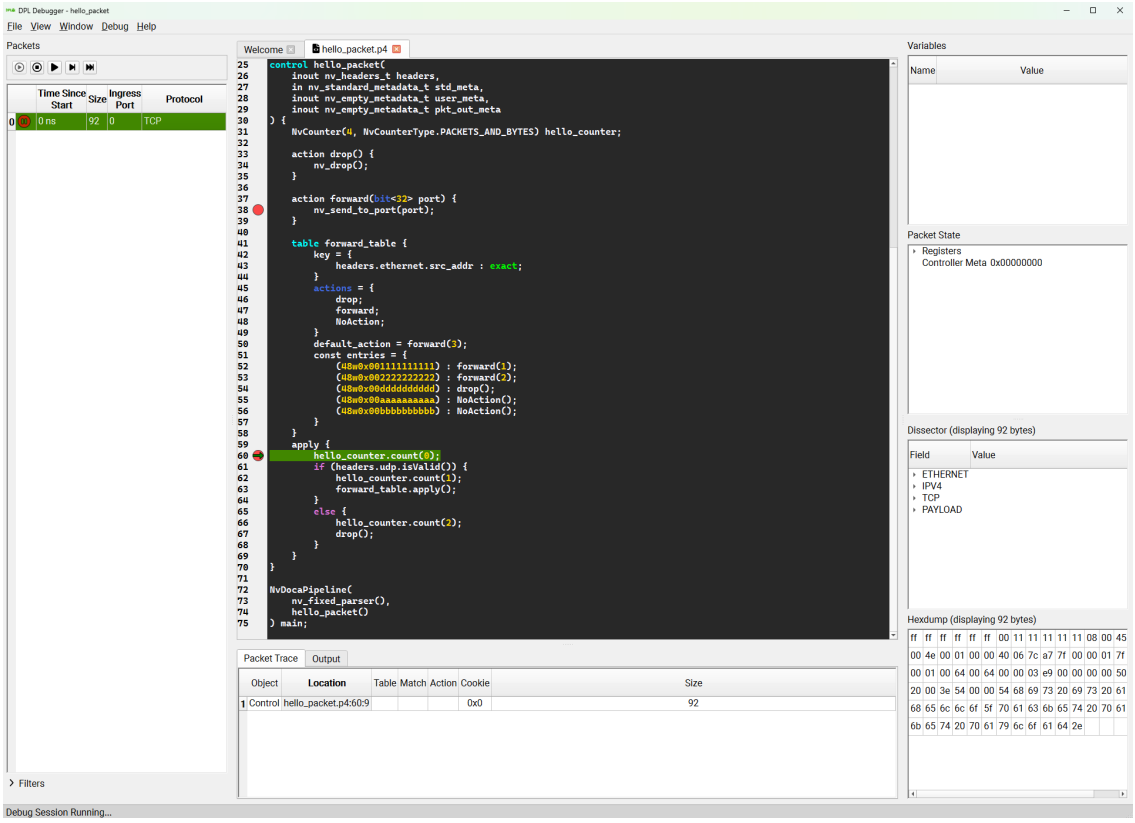
> The available debug points to enable depend on which debug mode you compiled the program with.

8.11.7.8.4.2 4.2. Inspecting and Editing Packets

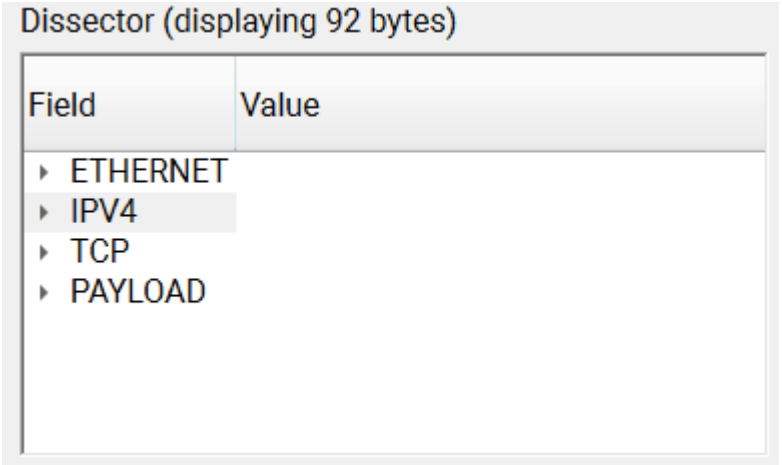
When a packet hits an enabled debug point, it is sent to the debugger.

- A red icon indicates the packet is currently paused by the debugger.

- A green arrow indicates the specific debug point where the packet is halted.



Once paused, you can edit the packet's content using either the Dissector or the Hexdump views. For example, you can edit the packet's protocol from TCP to UDP directly within the Dissector.



8.11.7.8.4.3 4.3. Advancing the Pipeline

After inspecting or editing a paused packet, inject it back into the pipeline using one of the following commands under the Debug menu:

- Step: Advances the packet to the very next line, even if no breakpoint is explicitly enabled there. This creates a new row in the Packet Trace widget.

- Continue: Advances the packet until it hits the next enabled breakpoint. If no further breakpoints exist, the packet finishes its pipeline and the pause icon disappears.
- Continue with trace: Functions like **Continue**, but records and displays the trace of all debug points the packet passed through along the way.

The packet in the **Packets** list correlates to the first row in the **Packet Trace** list. Changes made to the packet (e.g., size or protocol) will only be reflected in the subsequent row of the Packet Trace after reinjection.

8.11.7.8.4.4 4.4. Example Debugging Scenario

The following example demonstrates tracing a simple match/action pipeline.

4.4.1. The DPL Program (`hello_packet.p4`)

This program matches on a destination MAC address and either forwards or drops the packet. It was compiled with the `debug-mode=auto` flag.

```
#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>

control hello_packet(
    inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvCounter(4, NvCounterType.PACKETS_AND_BYTES) hello_counter;

    action drop() {
        nv_drop();
    }

    action forward(bit<32> port) {
        nv_send_to_port(port);
    }

    table forward_table {
        key = {
            headers.ethernet.src_addr : exact;
        }
        actions = {
            drop;
            forward;
            NoAction;
        }
        default_action = forward(3);
        const entries = {
            (48w0x001111111111) : forward(1);
            (48w0x002222222222) : forward(2);
            (48w0x00dddddddddd) : drop();
            (48w0x00aaaaaaaaaa) : NoAction();
            (48w0x00bbbbbbbbbb) : NoAction();
        }
    }
    apply {
        hello_counter.count(0);
        if (headers.udp.isValid()) {
            hello_counter.count(1);
            forward_table.apply();
        }
        else {
            hello_counter.count(2);
            drop();
        }
    }
}

NvDocaPipeline(
    nv_fixed_parser(),
    hello_packet()
) main;
```

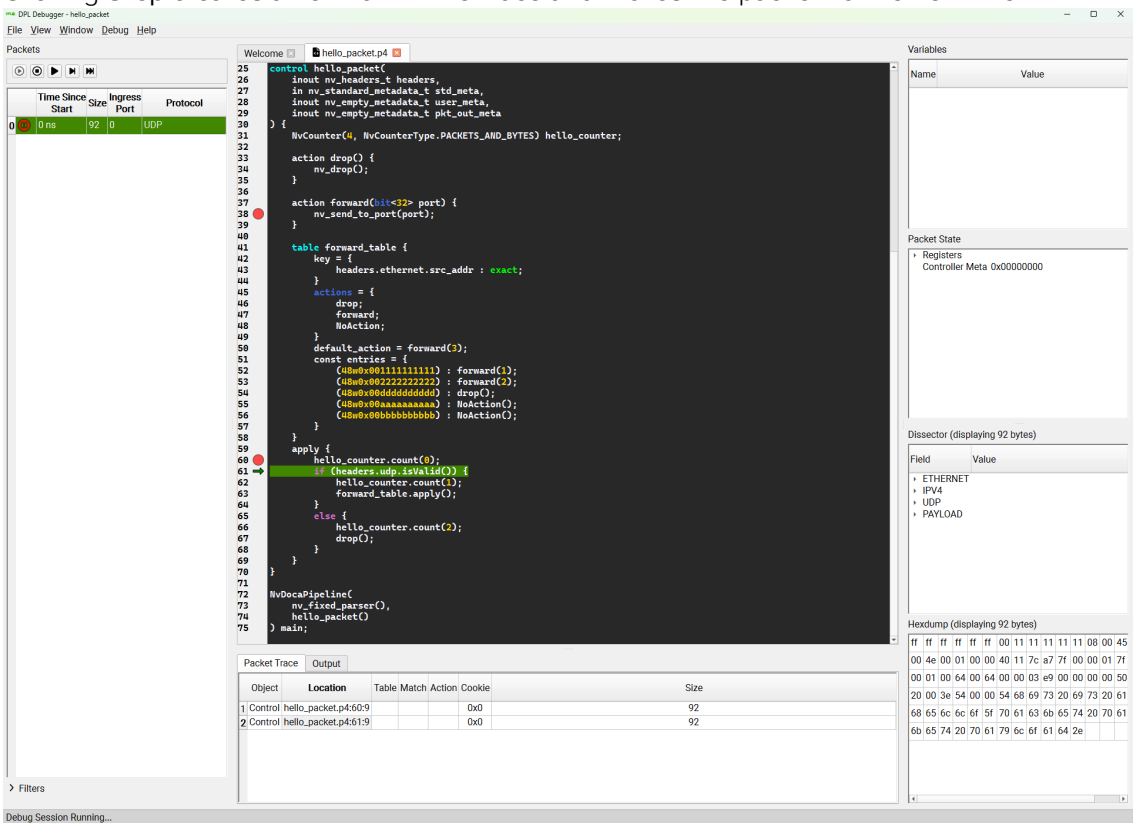
4.4.2. Sending Traffic (`send_hello_packet.py`)

Assuming port `p1` is connected via loopback to `p0`, this Python script sends a TCP packet to the pipeline.

```
from scapy.all import IP, TCP, Ether, Packet, Raw, sendp

packet = (Ether(src='00:11:11:11:11:11', dst='ff:ff:ff:ff:ff:ff') /
          IP() /
          TCP(sport=100, dport=100, seq=1001, flags='P') /
          Raw(load=b'This is a hello_packet packet payload.'))

sendp(packet, "p1", count=1)
```

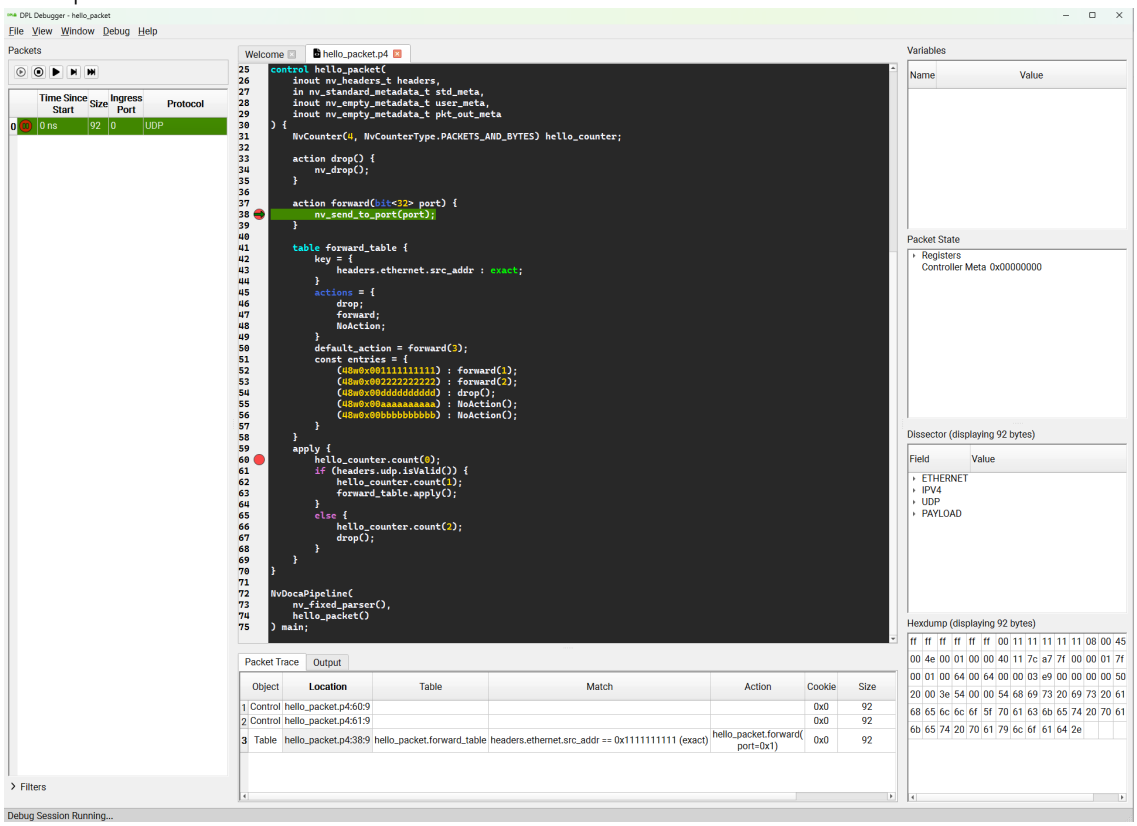
4.4.3. Execution & Trace

- The packet arrives at the debugger and pauses at the enabled breakpoint.
- Clicking **Step** creates a new row in the trace and moves the packet to the next line.



- If you edit the packet in the Dissector (e.g., changing TCP to UDP) and click **Continue**, the packet diverges based on your edit (satisfying the `if` statement) and pauses at the next

breakpoint.

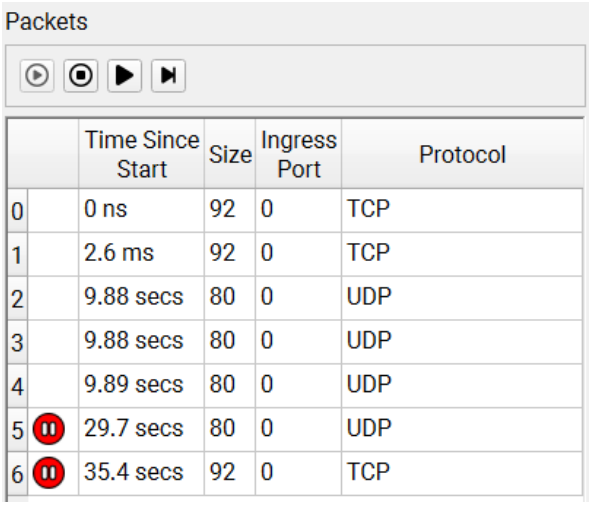


8.11.7.8.5 5. UI Reference: Widgets

The DPL Debugger provides multiple widgets to analyze packet states and system health:

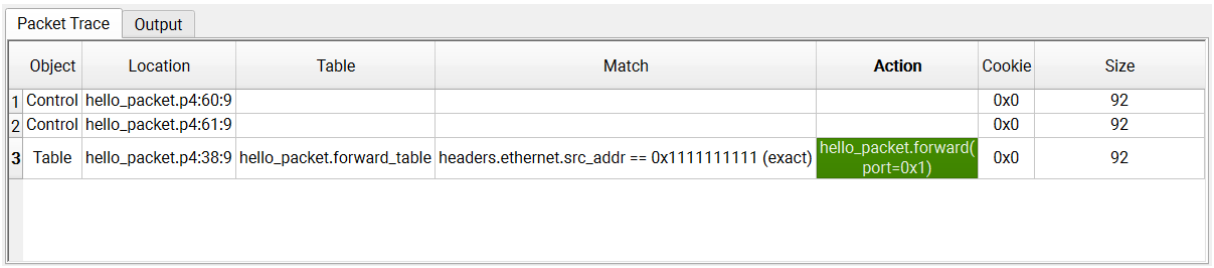
8.11.7.8.5.1 5.1. Packet List

Displays packets sent to the debugger. Selecting a packet reveals its trace. (Clear this list via File → Remove all packets).



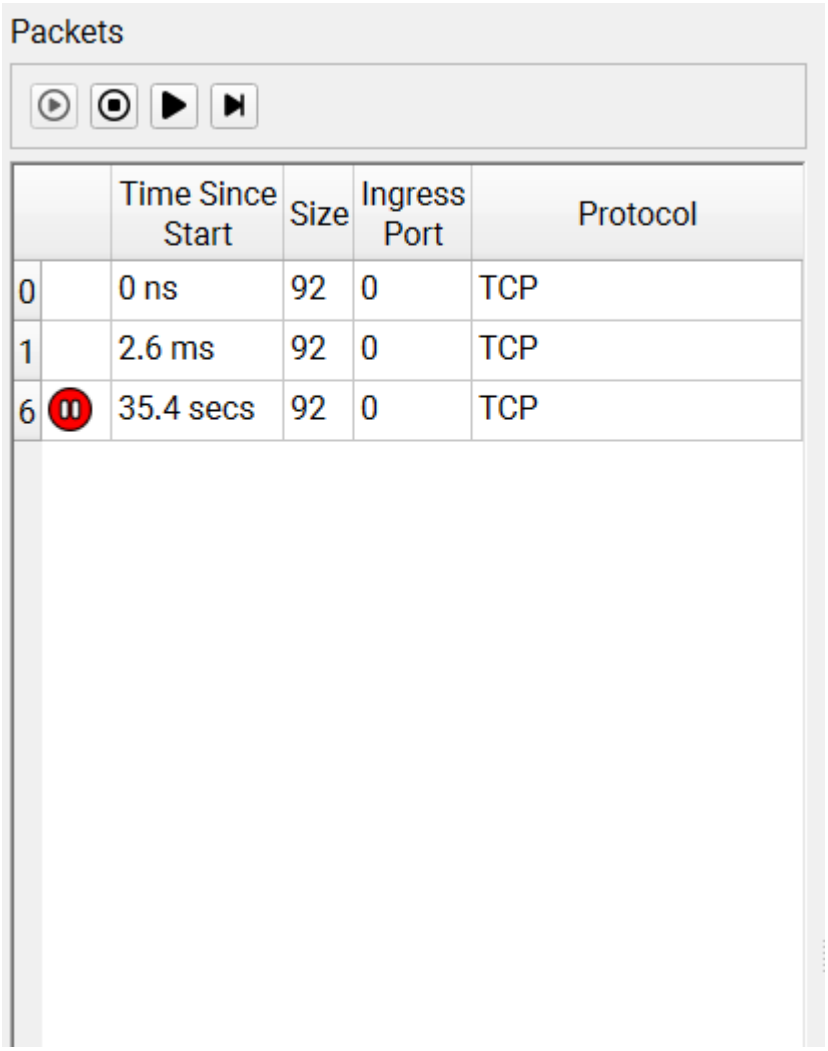
8.11.7.8.5.2 5.2. Packet Trace

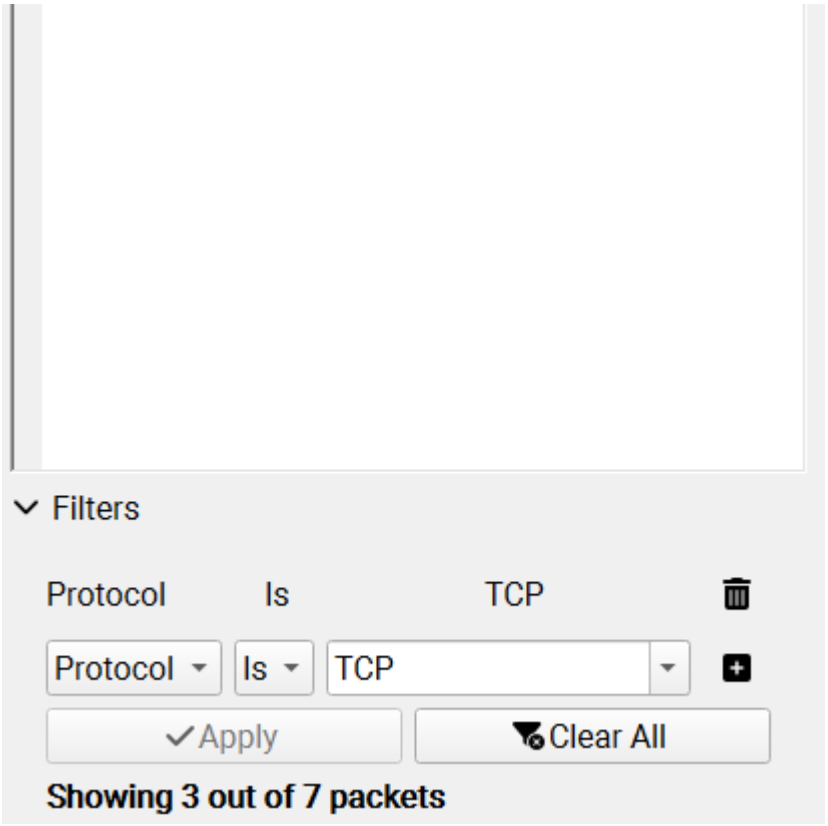
Each row represents a single debug point hit. Selecting a row reveals the packet data, state, and pipeline at that exact moment.



8.11.7.8.5.3 5.3. Filter

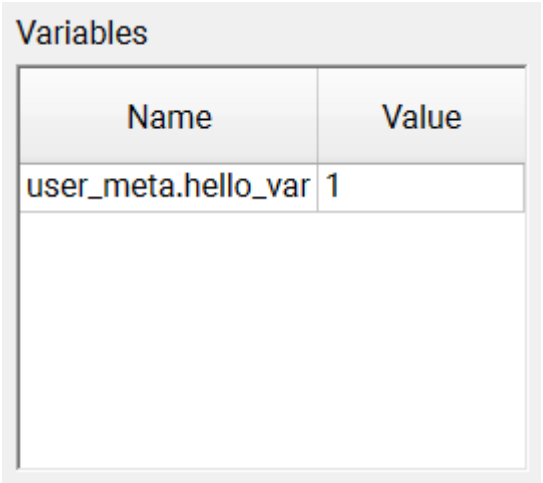
Allows packet filtering by Table Name, Action, Match Field, Match Type, Match Value, Ingress Port, Packet Size, or Protocol.





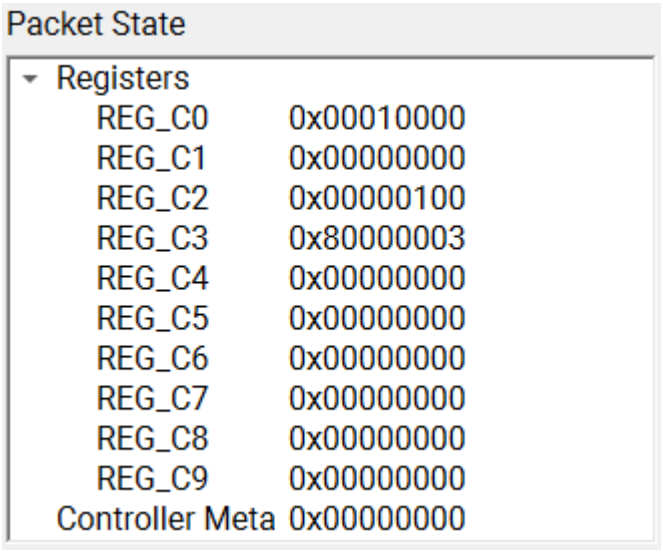
8.11.7.8.5.4 5.4. Variables

Displays the program's defined variables and their values at the selected debug point.



8.11.7.8.5.5 5.5. Packet State

Displays low-level variables, such as registers and controller metadata.



8.11.7.8.5.6 5.6. Packet Dissector

Displays the parsed packet. Paused packets can be edited by double-clicking values or pressing F2. (Values are shown/edited in decimal format and translated upon saving).

Dissector (displaying 80 bytes)

Field	Value
▾ ETHERNET	
dst_addr	ff:ff:ff:ff:ff:ff
src_addr	00:11:11:11:11:11
ether_type	Internet Protocol version 4 (IPv4)
▾ IPV4	
version	4
ihl	5
diffserv	CS0
ecn	Not-ECT
total_len	66
identification	1
flags	0
frag_offset	0
ttl	64
protocol	UDP
hdr_checksum	31912
src_addr	127.0.0.1
dst_addr	127.0.0.1
▾ UDP	
src_port	100
dst_port	100
length	46
checksum	45592
▾ PAYLOAD	
data	This is a hello_packet packet payload.

8.11.7.8.5.7 5.7. Hex Dump

Displays a raw hexadecimal dump of the packet. Double-click or press F2 to edit.

Hexdump (displaying 80 bytes)

ff	ff	ff	ff	ff	ff	00	11	11	11	11	11	08	00	45	00
00	42	00	01	00	00	40	11	7c	a8	7f	00	00	01	7f	00
00	01	00	64	00	64	00	2e	b2	18	54	68	69	73	20	69
73	20	61	20	68	65	6c	6c	6f	5f	70	61	63	6b	65	74
20	70	61	63	6b	65	74	20	70	61	79	6c	6f	61	64	2e

8.11.7.8.5.8 5.8. Source Code

Displays DPL source files. Clicking objects with source correlations (tables, actions, debug points) will automatically open the relevant file.

Welcome | hello_packet.p4

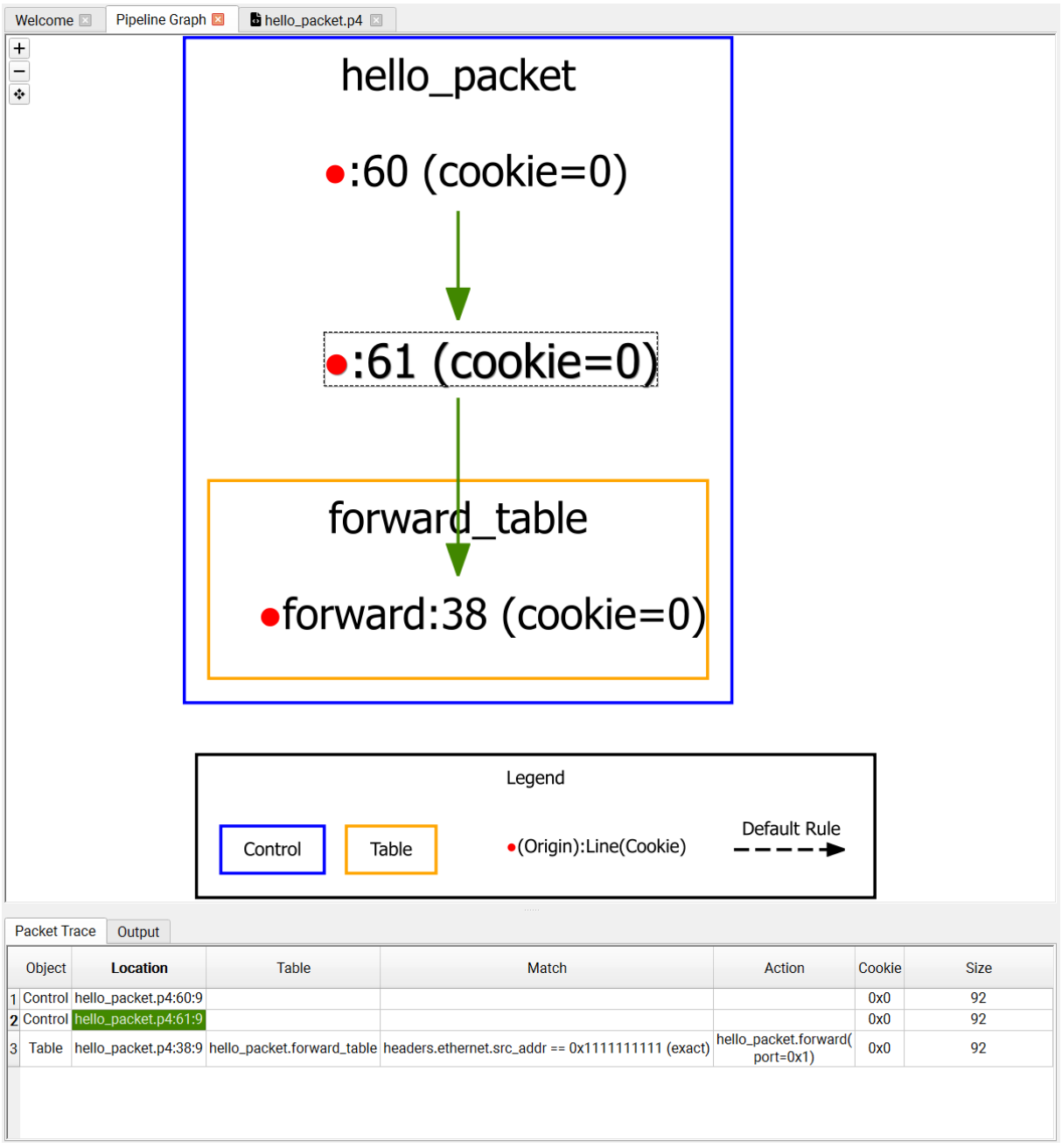
```
12
13  #include <doca_model.p4>
14  #include <doca_headers.p4>
15  #include <doca_externs.p4>
16  #include <doca_parser.p4>
17
18  /*
19   * This basic application demonstrates a simple match/action pipeline using NVIDIA
20   * DOCA P4 Target Architecture.
21   * - Match on destination MAC address
22   * - Forward or drop the packet
23   */
24
25  control hello_packet(
26      inout nv_headers_t headers,
27      in nv_standard_metadata_t std_meta,
28      inout nv_empty_metadata_t user_meta,
29      inout nv_empty_metadata_t pkt_out_meta
30  ) {
31      NvCounter(4, NvCounterType.PACKETS_AND_BYTES) hello_counter;
32
33      action drop() {
34          nv_drop();
35      }
36
37      action forward(bit<32> port) {
38          nv_send_to_port(port);
39      }
40
41      table forward_table {
42          key = {
43              headers.ethernet.src_addr : exact;
44          }
45          actions = {
46              drop;
47              forward;
48              NoAction;
49          }
50          default_action = forward(3);
51          const entries = {
52              (48w0x001111111111) : forward(1);
53              (48w0x002222222222) : forward(2);
54              (48w0x00dddddddddd) : drop();
55              (48w0x00aaaaaaaaaa) : NoAction();
56              (48w0x00bbbbbbbbbb) : NoAction();
57          }
58      }
59      apply {
60          hello_counter.count(0);
61          if (headers.udp.isValid()) {
62              hello_counter.count(1);
63              forward_table.apply();
```

Packet Trace | Output

	Object	Location	Table	Match	Action	Cookie	Size
1	Control	hello_packet.p4:60:9				0x0	92
2	Control	hello_packet.p4:61:9				0x0	92
3	Table	hello_packet.p4:38:9	hello_packet.forward_table	headers.ethernet.src_addr == 0x1111111111 (exact)	hello_packet.forward(port=0x1)	0x0	92

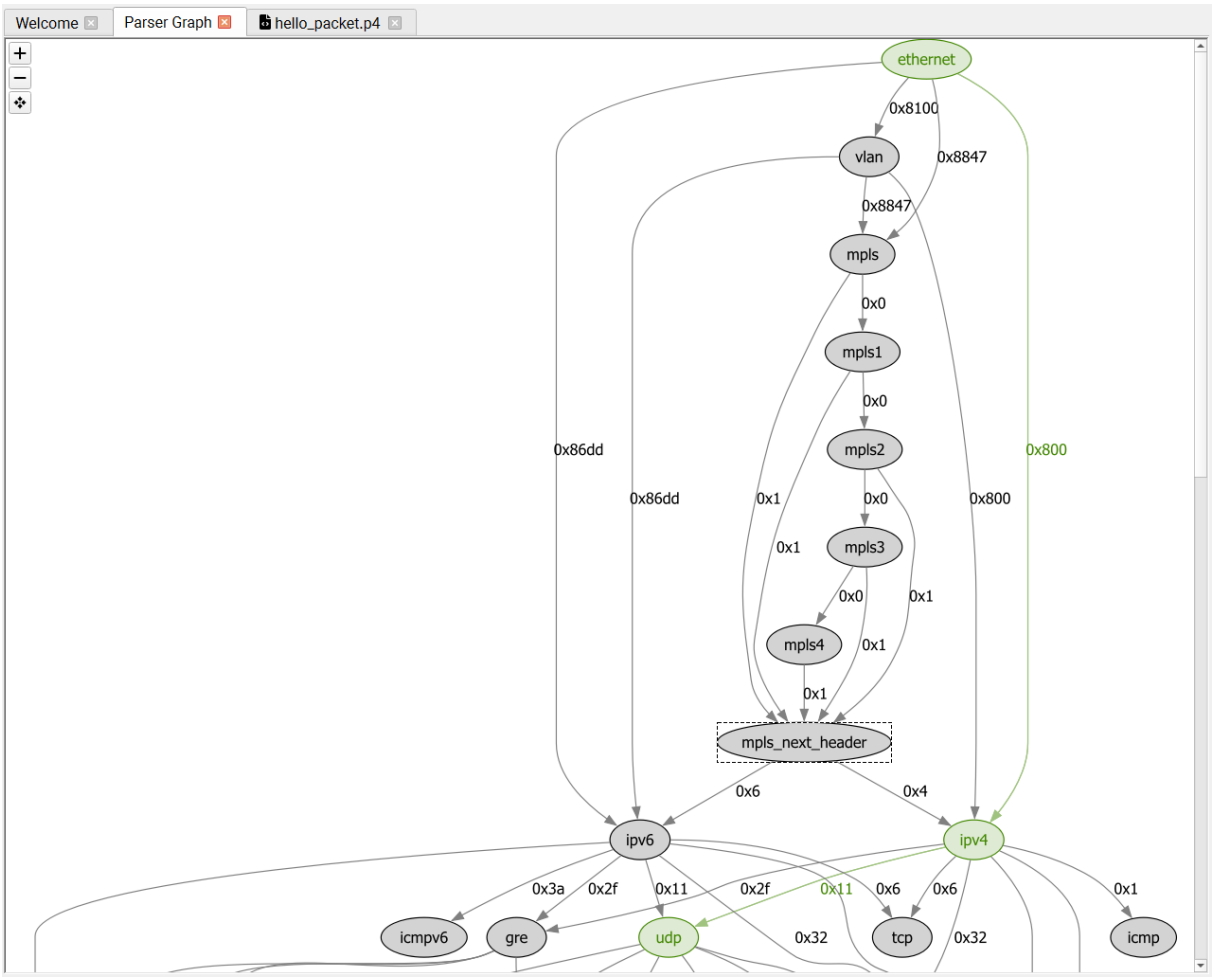
8.11.7.8.5.9 5.9. Pipeline

Displays the selected packet's pipeline (View → Pipeline Graph). Clicking a trace row highlights the corresponding debug point in the graph.



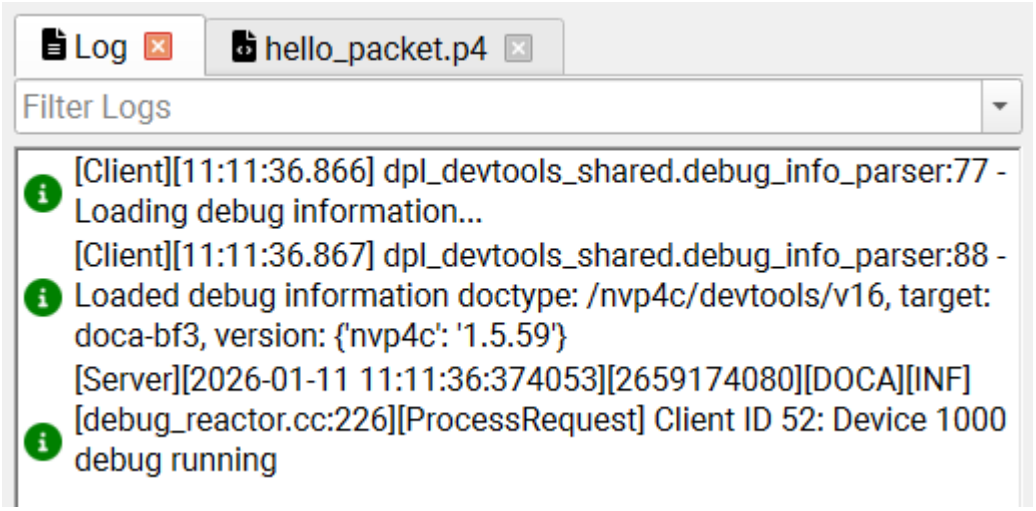
8.11.7.8.5.10 5.10. Parser

Displays the complete parser graph (both user-defined flex nodes and hardware-fixed nodes) and outlines how the selected packet was parsed (View → Parser Graph).

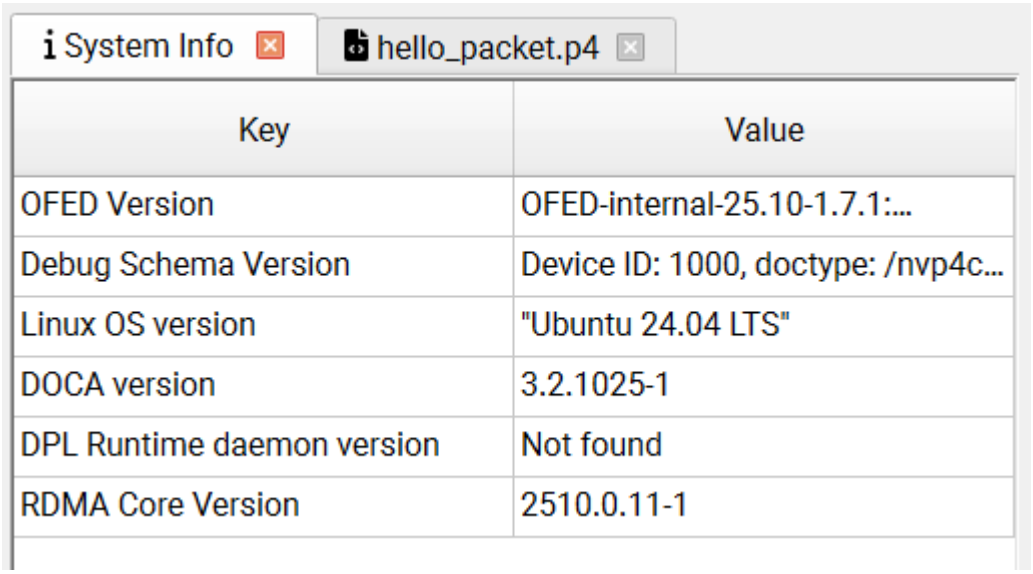


8.11.7.8.5.11 5.11. Log

Displays internal DPL Runtime daemon logs, filterable by severity (View → Log).



System Info: (View → System Info) Displays target system information.



8.11.7.8.6 6. Release Notes

8.11.7.8.6.1 6.1. New Features

Feature	Description
Correlated hexdump and dissector	Added highlighting, selecting a state or field in the dissector now highlights the corresponding bytes in the hexdump, and vice versa.
Source code keyboard controls	Enabled keyboard interaction in the code viewer. A caret is now visible and starts at the first line, it can be moved with the mouse or keyboard. Press `F9` to toggle a breakpoint on the caret line, matching the existing click-on-margin behavior. The breakpoint margin now has a tooltip.
Graph view buttons	Added zoom-in/zoom-out/fit-to-view buttons to the graph views.
Auto reinject checkbox	Added a checkbox to automatically reinject filtered-out packets as they arrive, so the debug session is not interrupted by filtered packets.
Paused packets checkbox	Added a checkbox to filter out unpaused packets from the packets table, so you can focus on paused packets only. When unchecked, both paused and unpaused packets are shown.
Externs column	Added an "Externs" column to the packet trace table, showing the externs associated with the table entry that the packet matched and their details.

8.11.7.8.6.2 6.2. Bug Fixes

- Code viewer syntax highlighting losing its bold weight after the window was resized.
- When a graph is not available, a placeholder message is now shown instead of a blank view.
- Packet filters using the "Is", "Is Not", "Starts With", or "Ends With" operators raise an exception when applied to fields that resolve to multiple values (such as match keys with several fields).
- Packets table's filter dropdowns not refreshing their available values after the packets table was cleared.
- Packet variables and packet details panes not keeping even proportions when their pane was resized.
- Packet state transitions during a debug session are more consistent and predictable.

8.11.7.9 DPL Program Metrics

8.11.7.9.1 Introduction

The DPL compiler produces metrics information for compiled DPL programs to provide developers with insight into how their programs are implemented in the target hardware architecture. This information is critical for identifying potential performance bottlenecks, such as expensive P4 tables, complex P4 actions, or critical processing paths through a complete program.

The information in this section describes what visibility the DPL compiler currently provides and how it can be used to guide DPL program development to achieve required performance.

8.11.7.9.2 Configuration and Output

By default, the DPL compiler outputs metrics information in the `p4-build` directory alongside other standard compiler outputs

8.11.7.9.2.1 Enable/Disable Metrics

- Default Behavior: Equivalent to including the compilation flag `--enable metrics`.
- Disable Metrics: Generation can be turned off using the flag `--disable metrics`.

8.11.7.9.2.2 Generated Files

- `program_name.metrics.log`: A text-based log summary of the compiled program's metrics.
- `program_name.metrics.json`: A JSON-structured representation, useful for tracking metrics programmatically over time.

8.11.7.9.3 Report Details (`program_name.metrics.log`)

The metrics log is based on a "lowered" representation of the DPL program, mapping high-level P4 constructs to hardware-aligned blocks called matchers and rules.

- Matcher: Generally correlates to a P4 Table.
- Rule: Generally correlates to a P4 Action.
- STE (Steering Table Entry): The fundamental hardware unit of work, consisting of a match phase and an action phase.

The log file contains three main sections:

8.11.7.9.3.1 DOCA HWS Metrics Report

This section summarizes resource usage and associated costs for matchers and rules. It begins with a summary of the total estimated memory usage and total unique STEs.

Column definitions:

Column	Description
ID	A unique identifier generated by the compiler, corresponding to ID values found in the `program_name.program.json` file
Table Name	The P4 table from which the matcher was derived (blank for synthesized matchers)
Domain	The hardware domain the matcher resides in. Currently allowed domains: FDB, FDB_RX, and FDB_TX.
Table Size	The capacity of rules (entries) the table can hold. This is expected to be a power of two due to memory allocation constraints.
Action Name	The P4 action from which the rule was derived.
Memory	Matcher: Bytes allocated to implement the matcher (enough for worst-case rules). Rule: Bytes required for a single rule of that type.

Column	Description
Packet Processing	A unitless estimate of execution cost. Higher numbers indicate more processing time. Matcher: Cost of processing a table miss (search + branching). Rule: Cost to execute the rule.
Insertion	A unitless estimate of the cost to insert a rule into a given matcher. Complex rules with many STEs take longer to insert.
Match STEs	The number of STEs required to perform the match operation (typically 1 or 2 for BlueField targets).
Action STEs	The number of STEs required to perform all of a rule's actions. A value of 0 indicates actions are performed in the same STE as the match.

8.11.7.9.3.2 DOCA HWS Metrics Report

This section summarizes the worst-case execution paths through the DPL program based on two different criteria:

1. Steps: The path requiring the highest number of STE steps.
2. Cost: The path with the highest cumulative packet processing cost.

Column definitions:

Column	Description
Step	The step order of the execution path, at the granularity of a hardware rule.
Table	The matcher ID visited at this step (includes P4 table name when possible).
Entry	The rule ID visited at this step (includes P4 action name when possible.
STEs	The number of STEs executed to perform the rule.
Pkt Processing	A unitless estimate of the cost of exercising the rule at this specific step.
Total	The sum of STE count and Packet Processing cost for the entire execution path.

8.11.7.9.3.3 Static entry hash distribution

When a program uses static entry tables (tables that have constant entries known at compile-time), the **metrics.log** will output a section that shows the hashing properties of each table. This section shows how the static entries are distributed across the allocated hash table. An example is shown below.

```
Static entry hash distribution:
Table 101 :
  0x00000000: x
  0x00000004: x
Table 102 :
  0x00000001: x
  0x00000003: x
  0x00000178: xxx
  0x00000563: x
```

In this example, two tables are implemented that have static entries, table 101 and table 102. For each table, a histogram of entries per (occupied) hash bucket is shown. For example, Table 102

has six entries (denoted by x's in the histogram). One entry is at bucket 0x1, one entry is at bucket 0x3, three entries are at bucket 0x178, and one entry is at bucket 0x563. We say that bucket 0x178 has hash collisions, because more than one entry is found within the hash bucket.

Hash collisions incur a cost in terms of performance, because more memory locations have to be examined to resolve the collision. The best performant table would have no hash collisions. While the compiler attempts to implement static entry tables to avoid hash collisions, it is not always possible. Refer to the Performance User Guide for tips on improving hashing performance.

8.11.7.9.4 Optimization Guidelines

The metrics output serves as a guide for optimizing DPL programs. Consider the following principles:

1. Minimize STE Execution: The total number of STEs executed has a direct adverse effect on packets per second (PPS). Fewer STEs result in faster processing times per packet.
2. Simplify Rules: Rule complexity impacts both memory footprint and insertion cost. Smaller, simpler rules execute faster and are faster to insert from the control plane.
3. Focus on Critical Paths: Complex tables or actions usually only impact performance when located on a critical program execution path. Optimizing rarely used paths (e.g., error handling) may not yield observable performance improvements.

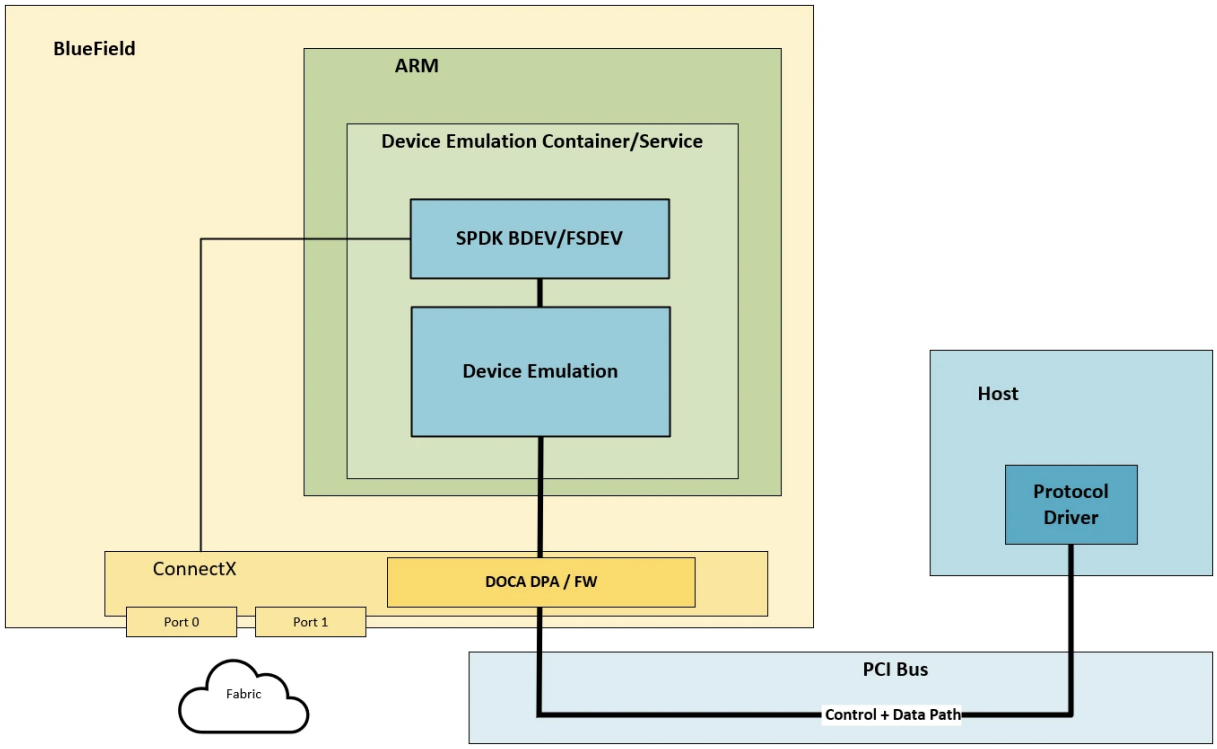
8.11.7.9.5 Disclaimers

- Estimates Only: The output provided in these metrics are estimates.
- Runtime Variables: The compiler cannot predict system caching performance or the specific characteristics of rules inserted at runtime.
- Memory Impact: Because fetch/retrieval times from system memory contain unknowns, actual packet processing costs may vary from these static estimates

8.12 DOCA SNAP Services

NVIDIA® DOCA™ Storage-Defined Network Accelerated Processing (SNAP) technology encompasses a family of services that enable hardware-accelerated virtualization of local storage running on NVIDIA® BlueField® products. The SNAP services present networked storage as local block or file system devices to the host, emulating local drives on the PCIe bus. The host OS or hypervisor utilizes its standard storage or file system driver, remaining unaware that communication is occurring not with a physical drive, but with a protocol-specific SNAP framework. This framework allows for any logic to be applied to the I/O requests or data before redirecting the requests and/or data over a fabric-based network to remote or local storage targets.

The DOCA SNAP services are built on BlueField technology, offering unique hardware-accelerated storage virtualization and isolation, combined with advanced networking and programmability. Together, SNAP and BlueField support a wide range of applications aimed at enhancing storage and networking efficiency, security, and performance.



NVIDIA offers the following DOCA SNAP services:

- NVIDIA DOCA SNAP Virtio-fs service – Virtio-fs for file system emulation on BlueField-3
- NVIDIA DOCA SNAP-4 service – NVMe and virtio-block for block device emulation on BlueField-3
- NVIDIA DOCA SNAP-3 – NVMe and virtio-block for block device emulation on BlueField-2

8.12.1 DOCA SNAP Virtio-fs Service Guide

8.12.1.1 1. Introduction

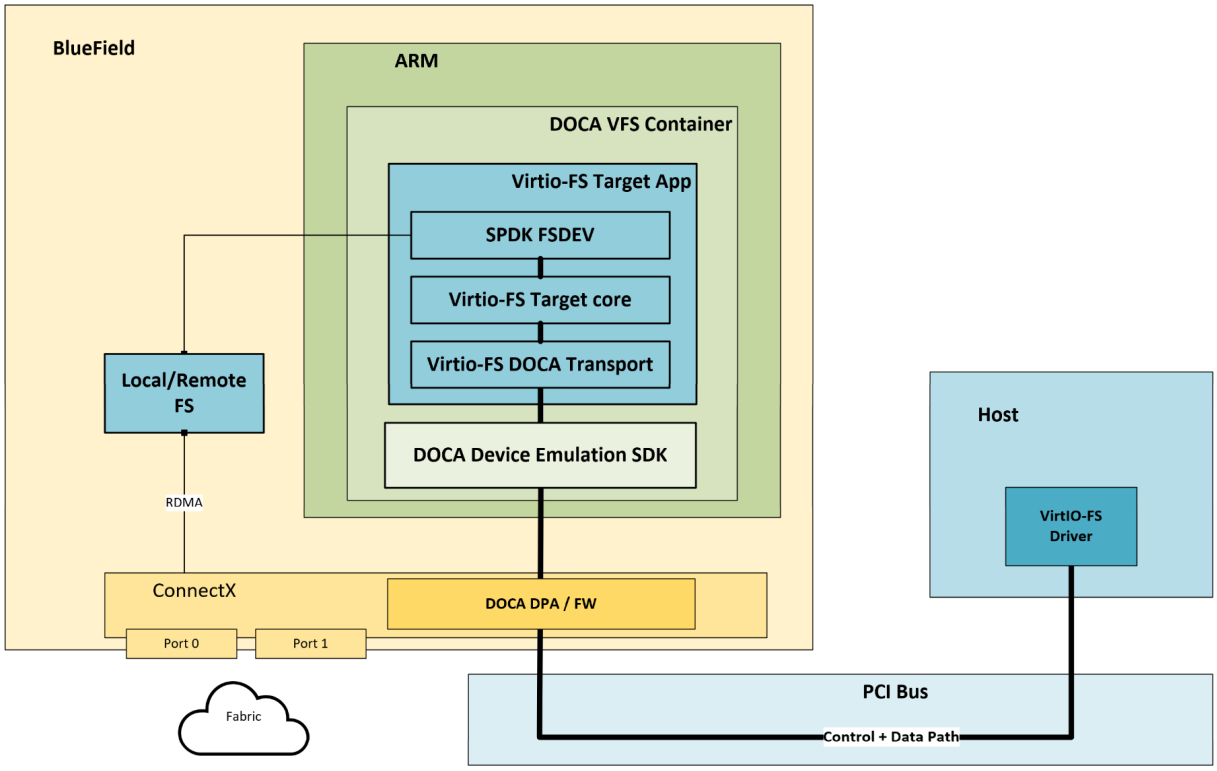
NVIDIA BlueField enables hardware-accelerated, software-defined virtio-fs PCIe device emulation. This leverages the power of BlueField networking platforms (DPUs or SuperNICs) to provide high-performance file system access in bare-metal and virtualized environments. Using BlueField, users can offload and accelerate networked file system operations from the host/guest, freeing up resources for other tasks and improving overall system efficiency. In this solution, the host/guest uses its own standard virtio-fs driver which is fully isolated from the networked filesystem mounted within the BlueField.

Built upon the DOCA and SPDK frameworks, virtio-fs device emulation on BlueField devices offers a comprehensive set of libraries for BlueField-based solutions and for storage solutions. This architecture consists of several key components:

- DOCA DevEmu subsystem and DOCA Virtio-fs library – These core libraries are responsible for the low-level hardware management and the translation of virtio descriptors carrying

FUSE (filesystem in userspace) requests into abstract virtio-fs requests, which are then processed by the SPDK virtio-fs DOCA transport component.

- SPDK virtio-fs transport – This component is responsible for the interaction with the low-level DOCA components and translating the incoming abstract DOCA SNAP Virtio-fs requests into generic virtio-fs request which are then processed by the virtio-fs target core.
- SPDK virtio-fs target – This component implements and manages the virtio-fs device, transports, and the interface with a backend file system. Upon arrival on a new generic virtio-fs request from the transport, it processes and translates the requests according to virtio-fs and FUSE specifications, translating FUSE-based commands into the generic filesystem protocol.
- SPDK fsdev – This component provides generic filesystem abstraction and interfaces with the low-level filesystem modules implementing a specific backend filesystem protocol.



8.12.1.1.1 1.1. DOCA SNAP Virtio-fs as Container

The DOCA SNAP Virtio-fs container image may be downloaded from NVIDIA NGC and easily deployed on the BlueField using a YAML file. The YAML file points to the docker image that includes DOCA SNAP Virtio-fs binaries aligned with the latest **spdk.nvda** version.

DOCA SNAP Virtio-fs is not pre-installed on the BFB but can be downloaded manually on demand. For instructions on how to install the DOCA SNAP Virtio-fs container, refer to DOCA SNAP Virtio-fs Container Deployment.

8.12.1.1.1.1 1.2.1. Data Path Providers

DOCA SNAP Virtio-fs facilitates user-configurable providers to assist in offloading data-path applications from the host. These include device emulation, I/O-intensive operations, and DMA operations.

Data Path Providers	Description
DPA provider	DPA (data path accelerator) is a cluster of multi-core and multi-execution-unit RISC-V processors embedded within the BlueField.
DPU provider	Hybrid mode that handles the data-path applications from the host using the BlueField CPU and the DPA hardware. This mode improves I/O latency and reduces SNAP downtime during crash recovery.

DPU is the default provider in DOCA SNAP Virtio-fs.

To switch provider, use the **provider** parameter in the **virtio_fs_transport_create** RPC. For DOCA transport, **provider=1** selects DPA and **provider=2** selects DPU.

To verify which data path provider is in use, check the service logs for the message **Data Path Provider 'DPA/DPU' set to be used by VirtioFS**, where **DPA** or **DPU** indicates the active provider.

8.12.1.2 2. DOCA SNAP Virtio-fs Deployment

This section describes how to deploy DOCA SNAP Virtio-fs as a container.

> ⚠ DOCA SNAP Virtio-fs does not come pre-installed with the BFB bundle.

> ⚠ The default virtio-blk emulation provider is set to **DPU**. In this mode, SNAP Virtio-fs and SNAP Virtio-blk can operate simultaneously only if they are assigned to different DPA execution units (EUs).
>
> This separation is achieved by setting the environment variable **dpu_helper_core_mask=0x1fffe**, which is configured in the **set_environment_variables.sh** script included in the SNAP Virtio-fs package.

8.12.1.2.1 2.1. Installing Full DOCA Image on BlueField

To install the BFB on BlueField:

```
[host] sudo bfb-install --rshim <rshimN> --bfb <image_path.bfb>
```

For more information, refer to "Installing Full DOCA Image on DPU" in the NVIDIA DOCA Installation Guide for Linux.

8.12.1.2.2 2.2. Firmware Installation

```
[dpu] sudo /opt/mellanox/mlnx-fw-updater/mlnx_fw_updater.pl --force-fw-update
```

For more information, refer to "Upgrading Firmware" in the NVIDIA DOCA Installation Guide for Linux.

8.12.1.2.3 2.3. Firmware Configuration

> **Warning**
>
> Firmware configuration may expose new emulated PCIe functions, which can be later used by the host's OS. As such, the user must make sure all exposed PCIe functions (static/hotplug) are backed by a supporting virtio-fs software configuration. Otherwise, these functions would malfunction and host behavior would be anomalous.

1. Clear the firmware configuration before implementing the required configuration:

   ```
   [dpu] mst start
   [dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 reset
   ```

2. Verify the firmware configuration:

   ```
   [dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 query
   ```

   Output example:

   ```
   mlxconfig -d /dev/mst/mt41692_pciconf0 -e query | grep VIRTIO_FS
   Configurations:                                 Default             Current             Next Boot
   *        VIRTIO_FS_EMULATION_ENABLE             False(0)            True(1)             True(1)
            VIRTIO_FS_EMULATION_NUM_VF             0                   0                   0
   *        VIRTIO_FS_EMULATION_NUM_PF             0                   2                   2
            VIRTIO_FS_EMU_SUBSYSTEM_VENDOR_ID      6900                6900                6900
            VIRTIO_FS_EMULATION_SUBSYSTEM_ID       4186                4186                4186
   *        VIRTIO_FS_EMULATION_NUM_MSIX           2                   3                   3
   ```

   The output provides 5 columns (listed from left to right):

   - Non-default configuration marker (`*`)
   - Firmware configuration name
   - Default firmware value
   - Current firmware value
   - Firmware value after reboot – shows configuration update pending system reboot

3. To enable storage emulation options, BlueField must be set to work in internal CPU model:

   ```
   [dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s INTERNAL_CPU_MODEL=1 PF_BAR2_ENABLE=0
   ```

4. To enable the firmware config with virtio-fs emulation PF:

   ```
   [dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s VIRTIO_FS_EMULATION_ENABLE=1 VIRTIO_FS_EMULATION_NUM_PF=1
   VIRTIO_FS_EMULATION_NUM_MSIX=3
   ```

> **Warning**
>
> For a complete list of the DOCA SNAP Virtio-fs firmware configuration options, refer to "Appendix – BlueField Firmware Configuration".

⚠ Power cycle is required to apply firmware configuration changes.

8.12.1.2.3.1 2.3.1. RDMA/RoCE Firmware Configuration

RoCE communication is blocked for the default interfaces of BlueField OS's (named ECPFs), **mlx5_0** and **mlx5_1** typically. If RoCE traffic is required, scalable functions (or SFs) must be added which are network functions which support RoCE transport.

To enable RDMA/RoCE:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s PER_PF_NUM_SF=1
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s PF_SF_BAR_SIZE=8 PF_TOTAL_SF=2
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0.1 s PF_SF_BAR_SIZE=8 PF_TOTAL_SF=2
```

⚠ This is not required when working over TCP or RDMA over InfiniBand.

⚠ When using 64KB page size OS on BlueField, replace **PF_SF_BAR_SIZE=8** with **PF_SF_BAR_SIZE=10 BAR_PAGE_ALIGNMENT=2** .

8.12.1.2.3.2 2.3.2. SR-IOV Firmware Configuration

DOCA SNAP Virtio-fs supports a maximum of 128 total VFs across the system. These VFs can be distributed across up to 4 virtio-fs PFs.

2.3.2.1. Standard SR-IOV Configuration

Use the following command to enable SR-IOV and set the standard link types and scalable functions:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s SRIOV_EN=1 PER_PF_NUM_SF=1 LINK_TYPE_P1=2 LINK_TYPE_P2=2
PF_TOTAL_SF=1 PF_SF_BAR_SIZE=8
```

2.3.2.2. Large-Scale Configuration (250 VFs)

Common example:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s SRIOV_EN=1 PER_PF_NUM_SF=1 LINK_TYPE_P1=2 LINK_TYPE_P2=2
PF_TOTAL_SF=1 PF_SF_BAR_SIZE=8
```

The following example configures a total of 128 VFs (64 VFs per PF across 2 PFs) with 2 queues allocated per VF:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s VIRTIO_FS_EMULATION_ENABLE=1 VIRTIO_FS_EMULATION_NUM_VF=64
VIRTIO_FS_EMULATION_NUM_PF=2 VIRTIO_FS_EMULATION_NUM_MSIX=5 VIRTIO_FS_EMULATION_NUM_VF_MSIX=6
```

8.12.1.2.3.3 2.3.3. Hot-plug Firmware Configuration

When PCIe switch emulation is enabled, BlueField can support **PCI_SWITCH_EMULATION_NUM_PORT** -1 hotplug virtio-fs function. These PCIe functions are

shared among all BlueField users and applications and may hold hot-plugged devices of type NVMe, virtio-blk, virtio-fs, and more (e.g., virtio-net).

To enable PCIe switch emulation and configure 31 hot-plugged ports to be used, run:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s PCI_SWITCH_EMULATION_ENABLE=1 PCI_SWITCH_EMULATION_NUM_PORT=32
```

PCI_SWITCH_EMULATION_NUM_PORT equals 1 plus the number of hot-plugged PCIe functions.

⚠ On AMD machines, hotplug is not guaranteed to work and enabling **PCI_SWITCH_EMULATION_ENABLE** may impact SR-IOV capabilities.

8.12.1.2.4 2.4. DPA Core Mask

The Data Path Accelerator (DPA) is an auxiliary processor designed to offload and accelerate data-path operations. It consists of a cluster of 16 cores, each containing 16 Execution Units (EUs).

- Total Capacity: 256 EUs (16 cores × 16 EUs)
- SNAP Allocation: 170 EUs are specifically available to SNAP (which runs a DPA application to accelerate the **virtio-fs** protocol).

The YAML-Based DPA Execution Unit Management Tool is the default mechanism for controlling DPA EUs. For the standard setup, refer to the DPA Resource Management Default Configuration.

If other DPA applications (e.g., **virtio-net**) are running concurrently with SNAP, you must explicitly configure the DPA resource YAML file to allocate specific EUs to each application.

⚠ For more details, see Single Point of Resource Distribution.

8.12.1.2.4.1 2.4.1. DPA EU Reservation

SNAP supports reserving DPA EUs for virtio-fs controllers. By default, all available EUs (0–170) are shared among all DPA applications on the system, including **virtio-fs** and **virtio-blk**.

8.12.1.2.4.2 2.4.2. YAML-based DPA Resource Management

DPA EU allocation is managed via a YAML-based resource file. For more details, see the documentation on Single Point of Resource Distribution.

This method centralizes and enforces consistent EU allocation across applications.

Requirements:

- Application name in the YAML file must match SNAP's DPA app:
  - **doca_devemu_virtio_dpa_app** → for **virtio-fs**
- At least one EU must be allocated for the **virtio-fs** DPA application.
- EU IDs must be in the range 1–170 (EU 0 is reserved).
- EU allocations must not overlap between applications.
- EU groups are not supported.

- SNAP's DPA application must run on the ROOT partition. EUs allocated to other partitions are not available.

8.12.1.2.4.3 2.4.3. DPA Resource Management Default Configuration

2.4.3.1. Default YAML Input Format for SNAP

```
---
version: 25.04
---
DPA_APPS:
  doca_devemu_virtio_dpa_app:
    - partition: ROOT
      affinity_EUs: [17-169]
  dpa_helper:
    - partition: ROOT
      affinity_EUs: [1-16]
  dpa_virtq_split:
    - partition: ROOT
      affinity_EUs: [1-169]
  dpa_nvme:
    - partition: ROOT
      affinity_EUs: [1-169]
```

8.12.1.2.4.4 2.4.4. Usage Instructions

1. Generate the output YAML file using the **dpa-resource-mgmt** tool:

   ```
   dpa-resource-mgmt config -d mlx5_0 -f ~/DPA_RESOURCE_INPUT.yaml
   ```

2. Set the **DEVEMU_DPA_RESOURCES_FILE_PATH** environment variable to point to the generated YAML file:

   ```
   export DEVEMU_DPA_RESOURCES_FILE_PATH=~/ROOT.yaml
   ```

Notes:

- Do not manually edit the YAML file generated by **dpa-resource-mgmt**.
- Each DPA EU supports up to 128 queues (threads).
- SNAP DPA applications only operate on the ROOT partition.

8.12.1.2.5 2.5. Virtio-fs DOCA Host Drive Installation (Optional)

The NVIDIA DOCA host package includes an optimized and extended version of the Virtio-fs driver; it provides better performance and additional features when compared to the upstream package.

- Optimized performance using Virtio-fs multi-queue with a better locking design.
- Support for a notification queue, allowing for a more accurate cached view of the filesystem by the host when remote changes happen.
- Support for GPU Direct Storage, allowing for zero-copy transfers between storage devices and GPUs.

Virtio-fs DOCA host package supports FLR only with Ubuntu 25.04

DOCA Host optimized virtiofs module support starts at kernel version 6.6

8.12.1.2.5.1 2.5.1. Supported OS

Optimized virtio-fs driver support is available for:

- Ubuntu 24.04
- Ubuntu 25.04 (with FLR)
- RHEL 10
- CentOS 10 (same package as RHEL10)
- OpenEuler 24.03

> ⚠ Debian does not support the optimized driver. Debian users have to fallback on the inbox drivers (no FLR, no notification queue).

8.12.1.2.5.2 2.5.2. Installation

To install DOCA host, install the following package:

1. Install the doca repo

```
[host] apt install doca-host_<version>-ubuntu2404_amd64.deb
```

2. Update the package cache list:

```
[host] apt update
```

3. Install NVIDIA virtio-fs:

```
[host] apt install virtiofs-dkms
```

> ⚠ Install OFED version 25.04-0.2.3.0 or later.

The virtio-fs DOCA host driver takes over your system version of the virtio-fs driver. It does not affect your system version of FUSE. Existing FUSE-based application should run unaffected.

8.12.1.2.5.3 2.5.3. Configuration

Once installed, configure your DPU to run a SNAP Virtio-fs service and reboot the host. The host system will be stuck during the boot process until the SNAP Virtio-fs service becomes available.

You can confirm you are running Virtio-fs DOCA Host driver by looking in your kernel log for mentions of it:

```
[host] dmesg | grep 'virtio-fs' virtio-fs: Loading NVIDIA-virtiofs +mq +lockless +nvq
```

You can now mount your Virtio-fs drive as usual:

```
[host] mount -t virtiofs <tagname> /mnt/virtiofs/
```

8.12.1.2.5.4 2.5.4. Advanced Configuration to Optimize Virtiofs Performance

Beginning with package **virtiofs-dkms version 25.07-OFED.25.07.0.2.3.1**, the sysfs file used to display CPU mappings has become writable. This means user can now manually assign CPUs to queues to adjust the default mapping.

Additionally, a new file called irq_affinity is now available under procfs for each queue. This file allows the user to specify one or more CPUs to direct that queue's interrupts accordingly. Note that user can only write to this file; to verify its impact, the user will need to manually check the system's interrupts.

These features allow users to experiment with and optimize CPU-to-queue mappings.

Example usage:

Assign CPUs 4, 5, and 6 to queue 4 (removing them from any previous assignment)

```
[host] echo 4,5,6 > /sys/fs/virtiofs/5/mqs/4/cpu_list
```

Direct queue 4's interrupts to CPUs 4, 5, and 6

```
[host] echo 4,5,6 > /sys/fs/virtiofs/5/mqs/4/irq_affinity
```

8.12.1.2.5.5 2.5.5. Uninstallation

If you wish to remove the Virtio-fs DOCA Host driver, you can do so with the following command. It will also remove packages automatically installed as dependencies:

```
[host] apt purge --autoremove virtiofs-dkms
```

8.12.1.2.6 2.6. DOCA SNAP Virtio-fs Container Deployment

DOCA SNAP Virtio-fs container is available on the DOCA SNAP Virtio-fs NVIDIA NGC page.

To deploy DOCA SNAP Virtio-fs container on top of BlueField, the following procedure is required:

1. Setup preparation and DOCA SNAP Virtio-fs resource download for container deployment. See section "Preparation Steps" for details.
2. Adjust the **doca_vfs.yaml** for advanced configuration if needed according to section "Adjusting YAML Configuration".
3. Deploy the container. The image is automatically pulled from NGC. See section "Spawning DOCA SNAP Virtio-fs Container" for details.

8.12.1.2.6.1 2.6.1. Preparation Steps

2.6.1.1. Step 1: Allocate Hugepages

Allocate **8GiB** hugepages for the DOCA SNAP Virtio-fs container according to the DPU OS's **Hugepagesize** value:

1. Query the **Hugepagesize** value:

```
[dpu] grep Hugepagesize /proc/meminfo
```

In Ubuntu22 and Ubuntu24, the value should be 2048KB. In Ubuntu24 with 64k page size, the value should be 524288KB.

2. For OS with 2048KB hugepage, use the **doca-hugepages** tool to configure the requested hugepages:

```
[dpu] doca-hugepages config --app snap --size 2048 --num 4096
```

3. For OS with 524288KB hugepage, use the **doca-hugepages** tool to configure the requested hugepages:

```
[dpu] doca-hugepages config --app snap --size 524288 --num 16
```

4. Reload the hugepages configuration for all applications based on the current database settings:

```
[dpu] doca-hugepages reload
```

If other applications are running concurrently within the setup and are consuming hugepages, make sure to allocate a quantity of hugepages appropriate to accommodate all applications.

2.6.1.2. Step 2: Create /etc/virtiofs Folder

The folder **/etc/virtiofs** is used by the container for automatic configuration after deployment.

The default YAML configuration only mounts the **/etc/virtiofs** folder for exposure and sharing between the container and the BlueField. This folder is used to expose configuration files or local file backends (e.g., AIO fsdev) from the DPU to the container.

8.12.1.2.6.2 2.6.2. Downloading YAML from NGC

The **.yaml** configuration file for the DOCA SNAP Virtio-fs container, **doca_vfs.yaml**, is uploaded to DOCA NGC.

Internet connectivity is necessary to download DOCA SNAP Virtio-fs resources.

8.12.1.2.6.3 2.6.3. Adjusting YAML Configuration

The **.yaml** file can easily be edited for advanced configuration.

- The DOCA SNAP Virtio-fs **.yaml** file is configured by default to support Ubuntu setups (i.e., **Hugepagesize** = 2048 kB) by using hugepages-2Mi.

To support other setups, edit the hugepages section according to the relevant **Hugepagesize** value for the BlueField OS. For example, to support CentOS 8.x or configure **Hugepagesize** to 512MB:

```
 limits:
    hugepages-512Mi: "<number-of-hugepages>Gi"
```

- The following example edits the **.yaml** file to request 6G memory for the DOCA SNAP Virtio-fs container:

```
    resources:
      requests:
        memory: "6Gi"
      limits:
        memory: "6Gi"
```

- The following example edits the **.yaml** file to request 8 CPU cores for the DOCA SNAP Virtio-fs container:

```
resources:
    cpu: "8"
  limits:
    cpu: "8"
env:
  - name: APP_ARGS
    value: "-m 0xff"
```

- To automatically configure the DOCA SNAP Virtio-fs container upon deployment:
  a. Add the **spdk_rpc_init.conf** file under **/etc/virtiofs/** . File example:

```
fsdev_ aio0 /etc/virtiofs/test
virtio_fs_transport_create -t DOCA
virtio_fs_transport_start -t DOCA
virtio_fs_device_create --transport-name DOCA --dev-name vfsdev0 --tag docatag --fsdev aio0 --num-
request-queues 1 --queue-size 32 --driver-platform x86_64
virtio_fs_doca_device_modify --dev-name vfsdev0 --manager mlx5_0 --vuid "MT2251XZ02WZVFSS0D0F3"
virtio_fs_device_start --dev-name vfsdev0
```

  b. Edit the **.yaml** file accordingly (uncomment):

```
env:
  - name: SPDK_RPC_INIT_CONF
    value: "/etc/virtiofs/spdk_rpc_init.conf"
```

8.12.1.2.6.4 2.6.4. Spawning DOCA SNAP Virtio-fs Container

Run the Kubernetes tool:

```
[dpu] systemctl restart containerd
[dpu] systemctl restart kubelet
[dpu] systemctl enable kubelet
[dpu] systemctl enable containerd
```

Copy the updated **doca_vfs.yaml** file to the **/etc/kubelet.d** directory.

Kubelet automatically pulls the container image from NGC described in the YAML file and spawns a pod executing the container.

```
cp doca_vfs.yaml /etc/kubelet.d/
```

The DOCA SNAP Virtio-fs Service starts initialization immediately, which may take a few seconds.

To verify whether DOCA SNAP Virtio-fs is running, send **spdk_rpc.py spdk_get_version** to confirm whether DOCA SNAP Virtio-fs is operational or still initializing.

8.12.1.2.6.5 2.6.5. Debug and Log

View currently active pods, and their IDs (it might take up to 20 seconds for the pod to start):

```
crictl pods
```

Example output:

```
POD ID              CREATED              STATE         NAME
0379ac2c4f34c       About a minute ago   Ready         virtiofs
```

View currently active containers, and their IDs:

```
crictl ps
```

View existing containers and their ID:

```
crictl ps -a
```

Examine the logs of a given container (virtio-fs logs):

```
crictl logs <container_id>
```

Examine the kubelet logs if something does not work as expected:

```
journalctl -u kubelet
```

The container log file is saved automatically by Kubelet under **/var/log/containers**.

8.12.1.2.6.6 2.6.6. Stop, Start, Restart DOCA SNAP Virtio-fs Container

- To stop the container, remove the **.yaml** file form **/etc/kubelet.d/**.
- To start the container, copy the **.yaml** file to the same path:

  ```
  cp doca_vfs.yaml /etc/kubelet.d
  ```

- To restart the container (with sig-term), use the **-t** (timeout) option:

  ```
  crictl stop -t 10 <container-id>
  ```

- To restart the SNAP service without restarting the entire container, the user can either use the **supervictl** tool or manually terminate the SNAP service process on the DPU. Different termination signals trigger different behaviors. For example, using **pkill** with the **-9** option sends a **SIGKILL**, which forcefully stops the process:

  ```
  pkill -9 -f virtiofs
  ```

After containers in a pod exit, the Kubelet restarts them using an exponential back-off strategy (e.g., 10s, 20s, 40s), with the delay capped at five minutes. If a container runs successfully for 10 minutes, the Kubelet resets the restart back-off timer for that container.

The termination of the **virtiofs** service may take time, as it must release all allocated resources. The duration depends on the scale of the use case and whether other applications are sharing resources with SNAP. Kubelt may display errors if the container termination timeout is shorter than the actual time required for cleanup.

8.12.1.2.7 2.8. DOCA SNAP Virtio-fs with SNAP Support

The DOCA SNAP virtio-fs container, along with its associated packages, natively supports DOCA SNAP-4, which is implemented as an SPDK subsystem module. This design enables the concurrent operation of both virtio-fs, virtio-blk and NVMe as a unified service. Additionally, DOCA SNAP is integrated as part of the DOCA SNAP virtio-fs deployment.

DOCA SNAP deployment sets **snap**, **snap_nvme**, and **snap_vblk** as SPDK subsystems which can be disabled as needed.

Refer to *DOCA SNAP-4 Service Guide* documentation for more information.

DOCA SNAP RPCs can be used as an SPDK plugin, the recommended method for running RPCs with the SPDK RPC script. Users may need to set the **PYTHONPATH** environment variable to include the path to **snap_rpc.py**.

This command creates a Virtio block controller using the DOCA SNAP RPC plugin, specifying **--pf_id 0** and using **Null0** as the block device:

```
spdk_rpc.py  --plugin snap_rpc virtio_blk_controller_create --pf_id 0 --bdev Null0
```

For further details on using RPC plugins, refer to the SPDK official documentation.

8.12.1.3 3. RPC Commands

Like other standard SPDK applications, the remote procedure call (RPC) protocol is used to control the DOCA SNAP Virtio-fs Service and supports JSON-based RPC protocol commands to control any resources and create, delete, query, or modify commands easily from the CLI.

DOCA SNAP Virtio-fs supports all standard SPDK RPC commands in addition to an extended DOCA SNAP Virtio-fs command set. Standard SPDK commands are executed by the **spdk_rpc.py** tool.

Virtio-fs-specific command set is implemented as a separate RPC plugin. When working with container, this RPC plugin is loaded automatically on **spdk_rpc.py** execution. When working with

sources, users must explicitly load the plugin by adding the **--plugin rpc_virtio_fs_tgt** flag to the SPDK's **spdk_rpc.py** command.

The following is an example of an RPC when using DOCA SNAP Virtio-fs from the source:

```
/opt/nvidia/spdk-subsystem/src/spdk/install-$(hostname)/bin/spdk_rpc.py --plugin rpc_virtio_fs_tgt --help
```

You may need to define the path to the folder with **rpc_virtio_fs_tgt.py** using the **PYTHONPATH** environment variable. It is normally located in the root of virtio-fs sources folder. More details on the RPC plugins can be found in SPDK's official documentation.

Full **spdk_rpc.py** command set documentation can be found in the SPDK official documentation site.

DOCA SNAP Virtio-fs extended commands are detailed in the following subsections.

8.12.1.3.1 3.1. Using JSON-based RPC Protocol

The JSON-based RPC protocol can be used with the **spdk_rpc.py** script inside the DOCA SNAP Virtio-fs container and **crictl** tool.

The DOCA SNAP Virtio-fs container is CRI-compatible.

- To query the active container ID:

  ```
  crictl ps -s running -q --name virtiofs
  ```

- To post RPCs to the container using **crictl**:

  ```
  crictl exec <container-id> spdk_rpc.py -v <RPC-method>
  ```

  The flag **-v** controls verbosity. For example:

  ```
  crictl exec 0379ac2c4f34c spdk_rpc.py -v virtio_fs_doca_get_functions
  ```

  Alternatively, an alias can be used:

  ```
  crictl exec -it $(crictl ps -s running -q --name virtiofs) spdk_rpc.py -v virtio_fs_doca_get_functions
  ```

- To open a bash shell to the container that can be used to post RPCs:

  ```
  crictl exec -it <container-id> bash
  ```

8.12.1.3.2 3.2. PCIe Function Management

Emulated PCIe functions are managed through DOCA devices called emulation managers. Emulation managers have special privileges to control, manipulate, and expose the emulated PCIe devices towards the host PCIe subsystem.

To operate a virtio-fs device/function by the DOCA transport, it is necessary to locate the appropriate emulation manager for it. The emulation manager maintains a list of the emulated PCIe functions it controls. Each of those functions is assigned a globally unique serial called a vendor unique identifier or VUID (e.g., MT2251XZ02WZVFSS0D0F2), which serves as unambiguous reference for identification and tracking purposes.

8.12.1.3.2.1 3.2.1. virtio_fs_doca_get_managers

List existing emulation managers for virtio-fs. This method has no input parameters.

Example response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": [
    {
      "name": "mlx5_0"
    }
  ]
}
```

8.12.1.3.2.2 3.2.2. virtio_fs_doca_get_functions

List functions for virtio-fs with their characteristics. The user may specify no parameters to list all emulated virtio-fs functions managed by any emulation manager device, or specify an emulation manager device name to list virtio-fs functions managed by that emulation manager device.

Example response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": [
    {
      "manager": "mlx5_0",
      "Function List": [
      {
        "hot pluggable": "false",
        "pci_address": "0000:86:00.2",
        "vuid": "MT2306XZ00A7VFSS0D0F2",
        "function_type": "PF",
        "total_vfs": 2
      },
      {
        "hot pluggable": "false",
        "pci_address": "0000:86:04.5",
        "vuid": "MT2306XZ00A7VFSS0D0F2VF1",
        "function_type": "VF",
        "pf_vuid": "MT2306XZ00A7VFSS0D0F2"
      },
      {
        "hot pluggable": "false",
        "pci_address": "0000:86:04.6",
        "vuid": "MT2306XZ00A7VFSS0D0F2VF2",
        "function_type": "VF",
        "pf_vuid": "MT2306XZ00A7VFSS0D0F2"
      }
      ]
    }
  ]
}
```

Parameter Name	Optional/ Mandatory	Type	Description
manager	Optional	String	Emulation manager device name to list emulated virtio-fs functions specific to it

8.12.1.3.2.3 3.2.3. virtio_fs_doca_get_possible_managers

List possible emulation managers for virtio-fs. This method has no input parameters.

Example response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": [
    {
      "name": "mlx5_0",
    }
  ]
}
```

8.12.1.3.2.4 3.2.4. virtio_fs_doca_manager_create

Create a virtio-fs emulation manager.

Parameter Name	Optional/Mandatory	Type	Description
manager	Mandatory	String	Emulation manager device name

8.12.1.3.2.5 3.2.5. virtio_fs_doca_manager_destroy

Destroy a virtio-fs emulation manager.

Parameter Name	Optional/Mandatory	Type	Description
manager	Mandatory	String	Emulation manager device name

> ⚠ The RPC may take a long time to complete or even timeout when many queues (>1000) are created.

8.12.1.3.3 3.3. Hot-pluggable PCIe Functions Management

Hotplug PCIe functions are configured dynamically at runtime using RPCs.

The commands outlined in the following subsections hot plug a new PCIe function to the system.

8.12.1.3.3.1 3.3.1. virtio_fs_doca_get_functions

List DOCA transport functions for virtio-fs with their characteristics.

Users may specify no parameters to list all emulated virtio-fs functions managed by any emulation manager device, or an emulation manager device name to list virtio-fs functions managed by a specific emulation manager device.

Parameter Name	Optional/ Mandatory	Type	Description
manager	Mandatory	String	Emulation manager device name for creating a new Virtio FS function

8.12.1.3.3.2 3.3.2. virtio_fs_doca_function_create

Create a DOCA virtio FS function. The return value of this method is a VUID. This is not needed for static functions as the VUID can be retrieved from virtio_fs_doca_get_functions.

Parameter Name	Optional/ Mandatory	Type	Description
manager	Mandatory	String	Emulation manager device name for creating a new virtio-fs function

8.12.1.3.3.3 3.3.3. virtio_fs_doca_function_destroy

Destroy a DOCA SNAP Virtio-fs function.

> This function should not be associated to any virtio-fs device.

Parameter Name	Optional/ Mandatory	Type	Description
manager	Mandatory	String	Emulation manager device name for destroying a virtio-fs function
vuid	Mandatory	String	VUID of the function to destroy

8.12.1.3.3.4 3.3.4. virtio_fs_doca_device_hotplug

Hot plug a DOCA SNAP Virtio-fs device. The virtio-fs device must be started.

Parameter Name	Optional/Mandatory	Type	Description
dev_name	Mandatory	String	Virtio-fs device name to hot plug
wait-for-done	Optional	Flag	If used, the method waits until the device is visible by the host PCIe subsystem. Otherwise, only issue hot-plug operation and exit.

8.12.1.3.3.5 3.3.5. virtio_fs_doca_device_hotunplug

Hot unplug a DOCA virtio FS device. The virtio FS device must be started.

Parameter Name	Optional/Mandatory	Type	Description
dev_name	Mandatory	String	Virtio-fs device name to hot unplug
wait-for-done	Optional	Flag	If exists, the method waits until the device is non-visible by the host PCIe subsystem. Otherwise, only issue hot-unplug operation and exit.

8.12.1.3.4 3.5. SPDK Fsdev Management

DOCA SNAP Virtio-fs uses the SPDK file system (fsdev) device framework as a backend for its virtio-fs controllers. Therefore, an SPDK fsdev must created and configured in advance.

Although the SPDK fsdev framework is generic and allows different types of the backend file system devices to be implemented. Currently, the only available backend devices are AIO and NFS.

8.12.1.3.4.1 3.5.1. fsdev_get_fsdevs

Get information about the SPDK filesystem devices (fsdevs). The user may specify no parameters to list all filesystem devices, or a filesystem device may be specified by name.

Parameter Name	Optional/Mandatory	Type	Description
name	Optional	string	Name of the fsdev of interest

8.12.1.3.4.2 3.5.2. fsdev_get_iostat

Get I/O statistics of filesystem devices (fsdevs).

Parameter Name	Optional/Mandatory	Type	Description
name	Optional	string	Fsdev name
per_channel	Optional	bool	Display per channel data.

If **name** is not specified, the RPC shows stats for all filesystem devices. If **per_channel** is not specified, the RPC shows the cumulative per-fsdev stat.

8.12.1.3.4.3 3.5.3. fsdev_reset_iostat

Reset I/O statistics of filesystem devices (fsdevs).

Parameter Name	Optional/Mandatory	Type	Description
name	Optional	string	Fsdev name

If `name` is not specified, the RPC resets the stats for all filesystem devices.

8.12.1.3.4.4 3.5.4. fsdev_aio_set_options

Set SPDK AIO fsdev module options. This method can be executed only in SPDK STARTUP state, i.e. it requires the service to be started with `--wait-for-rpc` CLI parameter.

Parameter Name	Optional/ Mandatory	Type	Description
`max_io_depth`	Optional	int	Maximum I/O depth per channel (Default: 256)
`enable_io_uring`	Optional	Flag	`true` to enable IO uring

> ⚠ `max_io_depth` maximum is dictated by the Linux kernel's `/proc/sys/fs/aio-max-nr` parameter. Setting this value higher than the default typically has little if any performance impact.

8.12.1.3.4.5 3.5.5. fsdev_aio_create

Create an SPDK AIO fsdev. This is the file system device that provides passthrough access to a local folder using either the Linux-native async I/O or POSIX async I/O. It can also use Linux io_uring APIs for read and write operations. IO uring is disabled by default and can be enabled with `fsdev_aio_set_options` RPC.

> ⚠ AIO fsdev IO uring implementation requires a compatible kernel version and is only supported with Ubuntu 24.04 and above.

Parameter Name	Optional/ Mandatory	Type	Default	Description
`name`	Mandatory	string		Name of the AIO fsdev to create
`root_path`	Mandatory	string		Path on the system directory to be exposed as an SPDK filesystem
`enable-xattr`	Optional	bool	`false`	Enable extended attributes if set to `true`
`enable-writeback-cache`	Optional	bool	`false`	Enable the writeback cache if set to `true`
`max-xfer-size`	Optional	int	`0x00020000`	Max data transfer size in bytes (minimum 4KB, maximum 1MB)
`max-readahead`	Optional	int	`0x00020000`	Max readahead size in bytes (minimum 4KB, maximum 128KB)

Parameter Name	Optional/ Mandatory	Type	Default	Description
`skip-rw`	Optional	bool	`false`	Enable skipping read/ write IOs if set to `true` ⚠ For debug purposes only.
`enable-notifications`	Optional	bool	`false`	Enable notifications if set to `true` ⚠ `fsdev_aio` relies on the Linux fanotify API to implement event notifications and therefore inherits all its limitations. For instance, notifications are not supported on filesystems that report a zero FSID, such as FUSE and NFS. For a complete list of limitations and behaviors, refer to the official fanotify documentation.
`attr-valid-ms`	Optional	int	`0`	File attribute validity time in milliseconds. Used for entry cache. A value of `0` disables entry caching entirely (minimum 0, maximum `INT32_MAX`).

⚠ AIO fsdev does not support recovery.

⚠ When using an `aio` fsdev backed by a Linux kernel NFS mount, certain file operations may result in 'silly-renamed' files (prefixed with `.nfs`) that are not automatically removed. These files can only be cleared by dropping caches on the host: `echo 3 > /proc/sys/vm/drop_caches`.

⚠ `aio` fsdevs currently do not support NFS kernel mounts with NFS targets that have `root_squash` enabled. It is recommended to use the native `nfs` fsdev instead.

8.12.1.3.4.6 3.5.6. fsdev_aio_delete

Delete an AIO fsdev.

Parameter Name	Optional/Mandatory	Type	Description
`name`	Mandatory	string	Name of the AIO fsdev to delete

8.12.1.3.4.7 3.5.7. fsdev_nfs_set_options

Set SPDK NFS fsdev module options. This method can be executed only in SPDK STARTUP state (i.e., it requires the service to be started with `--wait-for-rpc` CLI parameter).

Parameter Name	Optional/ Mandatory	Type	Description
`min_srcport`	Optional	int	Minimum source TCP port number
`max_srcport`	Optional	int	Maximum source TCP port number
`max_fobjects`	Optional	int	Maximum number of file objects allowed across all NFS fsdevs (default: 40M, maximum: 40M)

⚠ TCP Source Port Configuration

Defining the minimum and maximum source TCP port numbers is important depending on your specific NFS target:

- Required: If the NFS target strictly accepts TCP connections from reserved privileged ports (0-1023). Keep in mind that MOUNT connections always use TCP, even when operating over NFS RDMA.
- Recommended: For NFS TCP targets that require replayed requests to originate from the exact same TCP source port during recovery scenarios.

8.12.1.3.4.8 3.5.8. fsdev_nfs_create

Create an SPDK NFS fsdev:

Parameter Name	Optional/ Mandatory	Type	Description
name	Mandatory	string	Name of the NFS fsdev to create
server	Mandatory	string	NFS server's IP
exportname	Mandatory	string	NFS export's name
nfsport	Optional	int	Use this port for NFS instead of using the portmapper
protocol	Optional	string	**tcp** (default) or **rdma**
max_fobjects	Optional	int	Maximum number of file objects (default: 100K)

⚠ Only NFSv3 is supported.

⚠ NFS Target Port Configuration

To establish a successful connection, the NFS **fsdev** requires one of the following two port configurations:

- Allow non-privileged ports (target-side) – The NFS server must be configured to accept connections from non-privileged TCP ports. On Linux kernel NFS targets, you can enable this by adding the **insecure** flag to your export options.
- Enforce privileged ports (client-side) – You can configure the **fsdev** itself to use privileged local ports. To enforce this, specify the exact port range using the **min-srcport** and **max-srcport** parameters within the **fsdev_nfs_set_options** RPC.

⚠ FUSE Notifications and File Object Limits

To actively manage capacity, NFS **fsdev** uses FUSE notifications. Whenever the size of the file object map exceeds 80% of the defined **max_fobjects** limit, these notifications automatically trigger the host to "forget" (evict) older file objects.

⚠ Maximum NFS fsdev Capacity

The system supports a maximum of 34 NFS **fsdev** instances. Keep in mind that reaching this maximum concurrently depends heavily on your underlying storage performance; you may need to distribute these instances across multiple NFS servers to avoid bottlenecks.

⚠ Connection Timeouts with NFS RDMA at Scale

When operating at higher scales (e.g., attaching multiple hotplug PFs or VFs), the Linux Kernel NFS RDMA server may establish incoming connections slowly, which can result in timeouts. To prevent this, add a staggered delay between mounting different hotplug PFs on the host. A 10-second delay is typically sufficient, though you may need to adjust this depending on your specific NFS server's performance and load.

Note that this also applies to fsdev_aio_create when its backing filesystem is an NFS mount to a Linux Kernel NFS RDMA server.

Device Recovery Time

Recovery of 10 virtio-fs/NFS fsdev devices can take up to 20 seconds, with additional devices further increasing recovery time.

Number of File Objects

The total number of file objects across all NFS fsdevs cannot exceed the maximum amount specified in `fsdev_nfs_set_options` RPC.

8.12.1.3.4.9 3.5.9. fsdev_nfs_delete

Delete an NFS fsdev.

Parameter Name	Optional/Mandatory	Type	Description
`name`	Mandatory	string	Name of the NFS fsdev to delete

8.12.1.3.5 3.6. Virtio-fs Emulation Management

Virtio-fs emulation is a protocol belonging to the virtio family of devices. These mount points are found in virtual environments yet by design look like physical mount points to the user within the virtual machine. Each virtio-fs mount point (e.g., virtio-fs PCIe entry) exposed to the host, whether it is PF or VF, must be backed by a virtio-fs controller.

Probing a virtio-fs driver on the host without an already functioning virtio-fs controller may cause the host to hang until such controller is opened successfully (no timeout mechanism exists).

8.12.1.3.5.1 3.6.1. virtio_fs_transport_create

Create a virtio-fs transport. This RPC includes all the common parameters/options for all transports. The transport becomes operational once it is started.

Parameter Name	Optional/ Mandatory	Type	Description
`transport_name`	Mandatory	String	Transport type name. For DOCA SNAP Virtio-fs, `transport_name` should be DOCA.
`max_io_size_kb`	Optional	Number	The maximal IO size that the transport supports, in KB
`min_page_size_kb`	Optional	Number	The minimal page size that the transport supports, in KB.

Parameter Name	Optional/ Mandatory	Type	Description
provider	Optional	Number	DOCA transport datapath provider: 1-DPA, 2-DPU (default: 2-DPU); only valid when transport is DOCA
debug-level	Optional	Number	Log verbosity level for this transport. 1-ERROR, 2-WARN, 3-INFO, 4-DEBUG, default: 1 (ERROR)

8.12.1.3.5.2 3.6.2. virtio_fs_transport_destroy

Destroy a virtio-fs transport.

> ⚠ The transport must be stopped for destruction.

Parameter Name	Optional/ Mandatory	Type	Description
transport_name	Mandatory	String	Transport type name. For DOCA SNAP Virtio-fs, transport_name should be DOCA.

8.12.1.3.5.3 3.6.3. virtio_fs_transport_start

Start a virtio-fs transport. This RPC finalizes the transport configuration. From this point, the transport is fully operational and can be used to create new devices.

Parameter Name	Optional/ Mandatory	Type	Description
transport_name	Mandatory	String	Transport type name. For DOCA SNAP Virtio-fs, transport_name should be DOCA.

8.12.1.3.5.4 3.6.4. virtio_fs_transport_stop

Stop a virtio-fs transport. This RPC makes the transport configurable again.

> ⚠ A transport cannot be stopped if any devices are associated to it.

Parameter Name	Optional/ Mandatory	Type	Description
transport_name	Mandatory	String	Transport type name. For DOCA SNAP Virtio-fs, transport_name should be DOCA.

8.12.1.3.5.5 3.6.5. virtio_fs_get_transports

Display virtio-fs transports or requested transport.

Parameter Name	Optional/ Mandatory	Type	Description
`transport_name`	Optional	String	Transport type name. For DOCA SNAP Virtio-fs, `transport_name` should be DOCA.

8.12.1.3.5.6 3.6.6. virtio_fs_device_create

Create a virtio-fs device. This RPC creates a device with common parameters which are acceptable to all the transport types. To configure transport-specific parameters, users should use the `virtio_fs_doca_device_modify` command. The device becomes operational once it is started.

Parameter Name	Optional/Mandatory	Type	Description
`transport_name`	Mandatory	String	Transport type name. For DOCA SNAP Virtio-fs, `transport_name` should be DOCA.
`dev_name`	Mandatory	String	Virtio-fs device name to use
`tag`	Optional	String	Virtio-fs tag according to the virtio specification. ⚠ Must be provided during the `virtio_fs_device_create` RPC before the `virtio_fs_device_start` RPC.
`num_request_queues`	Optional	Number	Virtio-fs `num_request_queues` according to the virtio specification (default 31, range 1-254)

Parameter Name	Optional/Mandatory	Type	Description
`queue_size`	Optional	Number	The maximal queue size for all virtio queues (default 64; range 8-512) ⚠ The maximum supported I/O size is influenced by the `queue_size` value. Each virtio descriptor maps to a 4KB region of host memory, and each queue entry references one descriptor. For example, a 128KB I/O request requires 32 descriptors (one per 4KB block). If `queue_size` is set to 16, the queue cannot accommodate the full I/O in a single operation. In such cases, the host driver splits the I/O into smaller segments to fit within the available queue entries.

Parameter Name	Optional/Mandatory	Type	Description
`fsdev`	Optional	String	The name of the SPDK filesystem backend device ⚠ Must be provided during the `virtio_fs_device_create` RPC before the `virtio_fs_device_start` RPC. ⚠ RPC does not verify if fsdev is valid. If a wrong fsdev is attached to the device, the user would experience failure during mount of the FS on the host.
`packed_vq`	Optional	Bool	Expose packed virtqueues feature to the driver for negotiation.
`driver_platform`	Optional	String	Set the driver's platform architecture. Possible values: `native` ; `x86` ; `x86_64` ; `aarch32` ; `aarch64` . Using the `native` platform option sets the driver platform to be identical to the device platform.
`notification_vq`	Optional	Bool	Expose notification virtqueue feature to the driver for negotiation
`indirect_desc`	Optional	Bool	Enables/Disables the use of indirect descriptors for the device. False by default.
`driver_page_size_kb`	Optional	Number	Set the driver's page size (in KB). Value of 0 will set the driver page size to be identical to the device page size.

8.12.1.3.5.7 3.6.7. virtio_fs_device_start

Start a virtio-fs device. This RPC finalizes the device configuration. From this point, the transport is fully operational.

Parameter Name	Optional/Mandatory	Type	Description
dev_name	Mandatory	String	Virtio-fs device name

8.12.1.3.5.8 3.6.8. virtio_fs_device_stop

Stop a virtio-fs device.

Parameter Name	Optional/Mandatory	Type	Description
dev_name	Mandatory	String	Virtio-fs device name

> ⚠ The RPCs **virtio_fs_device_stop** and **virtio_fs_device_start** are supported during traffic. The user can stop the device while traffic is ongoing using **virtio_fs_device_stop**, then restart it using **virtio_fs_device_start**, and the device would continue handling I/O without any errors.

> ⚠ If the RPC returns an ETIMEDOUT (-110) error, the user is expected to retry the RPC.

8.12.1.3.5.9 3.6.9. virtio_fs_device_destroy

Destroy a virtio-fs device.

> ⚠ The device must be stopped before destruction.

Parameter Name	Optional/Mandatory	Type	Description
dev_name	Mandatory	String	Virtio-fs device name

8.12.1.3.5.10 3.6.10. virtio_fs_device_modify

Modify a virtio-fs device. This RPC is used to modify/set common properties of the device which are acceptable to all the transports.

Parameter Name	Optional/Mandatory	Type	Description
dev_name	Mandatory	String	Virtio-fs device name to use

Parameter Name	Optional/Mandatory	Type	Description
`tag`	Optional	String	Virtio-fs tag according to the virtio specification ⚠ Must be provided during `virtio_fs_device_create` or `virtio_fs_device_modify` RPCs, before `virtio_fs_device_start` RPC.
`num_request_queues`	Optional	Number	Virtio-fs `num_request_queues` according to the virtio specification (default 31; range 1-254)
`queue_size`	Optional	Number	The maximal queue size for all virtio queues (default 64; range 8-256)
`fsdev`	Optional	String	The name of the SPDK filesystem backend device ⚠ Must be provided during `virtio_fs_device_create` or `virtio_fs_device_modify` RPCs, before `virtio_fs_device_start` RPC. ⚠ RPC does not verify if fsdev is valid. If a wrong fsdev is attached to the device, the user would experience failure during mount of the FS on the host.
`packed_vq`	Optional	Bool	Expose packed virtqueues feature to the driver for negotiation

Parameter Name	Optional/Mandatory	Type	Description
`driver_platform`	Optional	String	Set the driver's platform architecture. Possible values: `native` ; `x86` ; `x86_64` ; `aarch32` ; `aarch64` . Using the `native` platform option sets the driver platform to be identical to the device platform.

8.12.1.3.5.11 3.6.11. virtio_fs_get_devices

Display virtio-fs devices with their characteristics.

- The user may specify no parameters to list the virtio-fs devices associated with all transports
- The user may specify the name of a transport to list the virtio-fs devices associated with it
- The user may specify the name of a virtio-fs device to display its characteristics

Transport name and device name parameters should be mutually exclusive.

Example response:


```
[
  {
    "name": "vfsdev0",
    "transport_name": "DOCA",
    "state": "running",
    "fsdev": "aio0",
    "tag": "doca_pf0",
    "queue_size": 256,
    "num_request_queues": 8,
    "packed_ring": false,
    "notification_vq": false,
    "manager": "mlx5_0",
    "vuid": "MT2306XZ00A7VFSS0D0F2",
    "num_msix": 0,
    "total_vfs": 128,
    "num_vfs": 2,
    "free_msix": 12,
    "free_queues": 138
  },
  {
    "name": "vfsdev0vf0",
    "transport_name": "DOCA",
    "state": "running",
    "fsdev": "aio1",
    "tag": "doca_pf0_vf0",
    "queue_size": 256,
    "num_request_queues": 1,
    "packed_ring": false,
    "notification_vq": false,
    "manager": "mlx5_0",
    "vuid": "MT2306XZ00A7VFSS0D0F2VF1",
    "num_msix": 2
  },
  {
    "name": "vfsdev0vf1",
    "transport_name": "DOCA",
    "state": "running",
    "fsdev": "aio2",
    "tag": "doca_pf0_vf1",
    "queue_size": 256,
    "num_request_queues": 1,
    "packed_ring": false,
    "notification_vq": false,
    "manager": "mlx5_0",
    "vuid": "MT2306XZ00A7VFSS0D0F2VF2",
    "num_msix": 2
  }]
```

Parameter Name	Optional/ Mandatory	Type	Description
transport_name	Optional	String	Name of transport whose associated virtio-fs devices to list
dev_name	Optional	String	Virtio-fs device name

8.12.1.3.5.12 3.6.12. virtio_fs_doca_device_modify

Modify a virtio-fs device created from DOCA transport.

> ⚠ Validation of params is not done. If an invalid param is set, **virtio_fs_device_start** RPC fails.

> ℹ This RPC is for configuring DOCA target specific parameters.

Parameter Name	Optional/Mandatory	Type	Description
dev-name	Mandatory	String	Virtio-fs device name
manager	Optional (must be provided before start)	String	Emulation manager
vuid	Optional (must be provided before start)	String	Vendor unique identifier
num-msix	Optional	Number	Total number of MSI-X vectors allocated for the device. • For hotplug devices: If unspecified, the system assigns a default value of 2. • For VFs: This parameter can only be modified only after the parent PF has reclaimed VF dynamic resources using virtio_fs_doca_device_reset_dynamic_resources.

8.12.1.3.5.13 3.6.13. virtio_fs_doca_device_reset_dynamic_resources

Reset dynamic MSI-X and doorbell/queue resources assigned to VFs for a started SR-IOV PF.

> ⚠ The PF device must already be started.

This RPC is for configuring DOCA target specific parameters.

Parameter Name	Optional/Mandatory	Type	Description
dev-name	Mandatory	String	Virtio-fs PF device name
skip_msix	Optional	Bool	Skip reset of dynamic MSI-X resources
skip_queues	Optional	Bool	Skip reset of queue resources
num-msix	Optional	Number	Total number of MSI-X vectors allocated for the device. • For hotplug devices: If unspecified, the system assigns a default value of 2. • For VFs: This parameter can only be modified only after the parent PF has reclaimed VF dynamic resources using virtio_fs_doca_device_reset_dynamic_resources`.

8.12.1.3.5.14 3.6.14. virtio_fs_device_get_stat

Displays stats for a specified **virtio-fs** device, or for all devices if none is specified.

Parameter Name	Optional/ Mandatory	Type	Description
dev_name	Optional	String	Virtio-fs device name The statistics of a virtio-fs device are valid only when the device is in "started" state.

This RPC command cannot be executed in parallel. Only one instance of the command can run at a time.

Example response:

```
{
jsonrpc": "2.0",
  "id": 1,
  "result": {
      "devices": [
        {
          "name": "vfsdev0",
          "fsdev": "aio0",
          "state": "running",
          "transport_name": "DOCA",
          "transport_specific": {
            "queues": [
              {
                "queue_id": 0,
                "queue_enabled": 1,
                "queue_depth": 256,
```

```
                "queue_inflights": 0,
                "hw_available_index": 8,
                "driver_available_index": 8,
                "hw_used_index": 8,
                "driver_used_index": 8
              },
              {
                "queue_id": 1,
                "queue_enabled": 1,
                "queue_depth": 256,
                "queue_inflights": 0,
                "hw_available_index": 104,
                "driver_available_index": 104,
                "hw_used_index": 104,
                "driver_used_index": 104
              }
            ]
          }
        }
      ]
    }
}
```


8.12.1.3.6 3.7. Configuration Example

8.12.1.3.6.1 3.7.1. Static Function – Bring up

The following is an example of creating virtio-fs DOCA transport and associating it to a virtio-fs device using a static physical function.

- In BlueField:
  a. Create an AIO fsdev backend:

```
spdk_rpc.py fsdev_aio_create aio0 /etc/virtiofs
```

  b. List possible emulation managers:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_get_possible_managers
```

  c. Create DOCA transport, emulation manager and start transport:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_transport_create -t DOCA
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_manager_create -m mlx5_0
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_transport_start -t DOCA
```

  d. Get transport information:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_get_transports
```

  e. Get managers information:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_get_managers
```

  f. Get function information, including their VUIDs:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_get_functions
```

  g. Create the virtio-fs device associated with DOCA transport:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_create --transport-name DOCA --dev-name
vfsdev0 --tag docatag --fsdev aio0 --num-request-queues 8 --queue-size 256 --driver-platform x86_64
```

  h. Set and modify virtio-fs parameters (VUID must be provided before calling **virtio_fs_device_start** RPC):

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_device_modify --dev-name vfsdev0 --manager
mlx5_0 --vuid MT2333XZ0VJQVFSS0D0F2
```

i. Start the virtio-fs device:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_start --dev-name vfsdev0
```

j. Get device information:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_get_devices
```

- In VM/host:
  - To mount a device with the tag **docatag** and load **virtio_pci** driver if not loaded:

```
mkdir "/tmp/test"
modprobe -v virtiofs
mount -t virtiofs docatag /tmp/test
```

8.12.1.3.6.2 3.7.2. Static Function – Teardown

- In BlueField:
  a. Get device information:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_get_devices
```

b. Stop and destroy the virtio-fs device:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_stop --dev-name vfsdev0
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_destroy --dev-name vfsdev0
```

c. Stop the DOCA transport and destroy emulation manager and transport:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_transport_stop -t DOCA
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_manager_destroy -m mlx5_0
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_transport_destroy -t DOCA
```

- In VM/host:
  - To unmount the device:

```
umount /tmp/test
modprobe -rv virtiofs
```

8.12.1.3.6.3 3.7.3. Hotplug Function

The following is an example of creating virtio-fs DOCA transport, creating a virtio-fs function, associating it to a virtio-fs device, and hot-plugging it:

- In BlueField:
  a. Create AIO fsdev backend:

```
spdk_rpc.py fsdev_aio_create aio0 /etc/virtiofs
```

b. List possible emulation managers:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_get_possible_managers
```

c. Create the DOCA transport and emulation manager and start the transport:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_transport_create -t DOCA
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_manager_create -m mlx5_0
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_transport_start -t DOCA
```

d. Get transport information:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_get_transports
```

e. Get managers information:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_get_managers
```

Some managers would show hotplug capability.

f. Get functions information:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_get_functions
```

g. Create virtio-fs function:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_function_create --manager mlx5_0
```

Returns VUID **MT2333XZ0VJQVFSS0D0F2**.

h. Get functions information:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_get_functions
```

Returns the function that has been created with the appropriate VUID.

i. Create the virtio-fs device associated with DOCA transport:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_create --transport-name DOCA --dev-name
vfsdev0 --tag docatag --fsdev aio0 --num-request-queues 8 --queue-size 256 --driver-platform x86_64
```

j. Set and modify virtio-fs parameters:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_device_modify --dev-name vfsdev0 --manager
mlx5_0 --vuid MT2333XZ0VJQVFSS0D0F2
```

The VUID must be provided before calling the **virtio_fs_device_start** RPC.

k. Start the virtio FS device:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_start --dev-name vfsdev0
```

l. Get device information:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_get_devices
```

The output for **vfsdev0** would show it is not yet plugged.

m. Hot plug the DOCA device to the host and wait until it becomes visible by the host:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_device_hotplug --dev-name vfsdev0 --
wait-for-done
```

- In VM/host:

- To mount a device with the tag **docatag** and load **virtio_pci** driver if not loaded:

```
mkdir "/tmp/test"
modprobe -v virtiofs
mount -t virtiofs docatag /tmp/test
```

8.12.1.3.6.4 3.7.4. Hot-unplug Function

The following is an example of cleaning up and destroying the flow described under section "Hotplug Function":

- In VM/host:
  - To unmount the device:

```
umount /tmp/test
modprobe -rv virtiofs
```

- In BlueField:
  a. Get device information:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_get_devices
```

  b. Hot unplug the DOCA device from the host and wait until it becomes non-visible by the host:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_device_hotunplug --dev-name vfsdev0 --
wait-for-done
```

  c. Get device information:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_get_devices
```

  d. Stop and destroy the virtio-fs DOCA device:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_stop --dev-name vfsdev0
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_destroy --dev-name vfsdev0
```

  e. Destroy the virtio-fs function:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_function_destroy --manager mlx5_0 --vuid
MT2333XZ0VJQVFSS0D0F2
```

Stop the DOCA transport and destroy the emulation manager and transport:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_transport_stop -t DOCA
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_manager_destroy -m mlx5_0
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_transport_destroy -t DOCA
```

8.12.1.4 4. RPC Log History

Supported from SPDK version **spdk23.01.2-12** and above.

RPC log history (enabled by default) records all RPC requests and their responses in **/var/log/virtiofs/rpc-log**. This file is also available outside the container at **/var/log/snap-log/rpc-log** on the DPU.

The file is continuously updated in append mode. Monitor its size, and delete it before launching the SNAP pod if it becomes too large.

8.12.1.5 5. Saving and Loading Configuration

To recover and reuse configurations for virtio-fs, you can save the desired configuration and load it later using a saved configuration file. This process allows for quick recovery and ensures consistent device/application setup.

To save the configuration of virtio-fs devices:

1. Use RPCs to configure your virtio-fs device/application as needed.
2. Save the configuration after each configuration cycle using the **save_config** RPC. This creates a JSON configuration file for future use:

```
spdk_rpc.py save_config -o <json_config>.json
```

To initiate a configuration cycle, create a new configuration file using the same format as **spdk_rpc_init.conf** files:

```
cat <new_config>.conf | spdk_rpc.py
```

The last line of the **<new_config>.conf** file must include a call to save the RPC configuration. Example:

```
--plugin rpc_virtio_fs_tgt -v fsdev_aio_create aio0 /tmp/fsdev-test
--plugin rpc_virtio_fs_tgt -v virtio_fs_transport_create -t DOCA
--plugin rpc_virtio_fs_tgt -v virtio_fs_transport_start -t DOCA
--plugin rpc_virtio_fs_tgt -v virtio_fs_device_create --transport-name DOCA --dev-name vfsdev0 --tag
docatest --fsdev aio0 --num-request-queues 1 --queue-size 256 --driver-platform x86_64
--plugin rpc_virtio_fs_tgt -v virtio_fs_doca_device_modify --dev-name vfsdev0 --manager mlx5_0 --vuid
MT2306XZ00HKVFSS0D0F4
--plugin rpc_virtio_fs_tgt -v virtio_fs_device_start --dev-name vfsdev0
save_config -o <json_config>.json
```

Saving configuration at the end of the script ensures the configuration is saved only after all the script changes are successfully applied.

To load a saved configuration for virtio-fs, use the following methods based on your environment:

- When using virtio-fs from a container, set the following in the YAML configuration file:

```
SPDK_RPC_INIT_CONF_JSON: <json_config>.json
SPDK_APP_OPTS: "--shm-id 1"
```

- When using virtio-fs from source, re-run the DOCA SNAP Virtio-fs application, recovering the configuration from the saved file:

```
/opt/nvidia/spdk-subsystem/src/virtio-fs-target/app/virtio_fs_tgt -c <json_config>.json --json-ignore-init-errors
```

When modifying controller or function configurations, ensure the host driver remains unloaded until the configuration change is fully applied. If the driver loads before completion, the system may fail to recover from a partial or interrupted configuration.

When using **SPDK_RPC_INIT_CONF_JSON**, disable other initialization methods such as **SPDK_RPC_INIT_CONF** to avoid conflicts.

Hotplugged emulation functions persist across process restarts (but not BlueField resets). These functions should be configured once during the process's initial setup. Only controllers created on these functions are saved in the configuration state.

If you specify **--wait-for-rpc** in your YAML file to enable initial interactive RPC bring-up, you must remove it before enabling automatic recovery from a generated configuration file. Leaving it in place will stall the automated recovery process.

8.12.1.6 6. Disabling/enabling SPDK Subsystem

Virtio-fs is defined as an SPDK subsystem and can be disabled or enabled as required.

The Virtio-fs subsystem is enabled by default. To disable it:

1. Start the Virtio-fs application in a waiting state for RPC commands:

```
/opt/nvidia/spdk-subsystem/src/virtio-fs-target/app/virtio_fs_tgt --wait-for-rpc
```

2. Use the following RPC command to disable the **virtio_fs** subsystem:

```
spdk_rpc.py framework_disable_subsystem  virtio_fs
```

3. Restart the SPDK framework initialization:

```
spdk_rpc.py framework_start_init
```

To view the status of all subsystems:

```
spdk_rpc.py framework_get_subsystems
```

This subsystem configuration can be saved in SPDK configuration for future use with the command **spdk_rpc.py save_config**. This ensures the configuration persists across application restarts.

8.12.1.7 7. SR-IOV

SR-IOV configuration depends on the kernel version and must be handled carefully to ensure device visibility and system stability across both the hypervisor and DPU orchestrators.

8.12.1.7.1 7.1. General Setup Procedure

To ensure a safe and stable SR-IOV setup, follow these steps in order:

8.12.1.7.1.1 7.1.1. Step 1: Preconfigure VF Controllers (DPU Side)

Before configuring SR-IOV on the host, ensure that the DPU is properly configured with all required VF devices created and started.

- Verify Visibility: VF functions are always visible and configurable on the DPU side. Verify the current state:

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_get_functions
```

- Lock-in: Once host-side configuration begins, further modifications to these resources may not be possible.

8.12.1.7.1.2 7.1.2. Step 2: Disable Autoprobing

In deployments with many virtual devices (large-scale), autoprobing must be disabled to ensure stable device discovery. Failing to do so may result in incomplete device visibility, system hangs during initialization, or unreliable behavior (specifically with >100 VFs).

Configuration:

```
# Recommended for large-scale deployments
echo 0 > /sys/bus/pci/devices/<BDF>/sriov_drivers_autoprobe
```

8.12.1.7.1.3 7.1.3. Step 3: Configure SR-IOV (Host Side)

Enable the Virtual Functions on the host. Note that hot-plugged PFs do not support SR-IOV.

- Small-scale deployments (<100 VFs):

```
echo <number_of_vfs> > /sys/bus/pci/devices/<BDF>/sriov_totalvfs
```

- Newer drivers:

```
echo <number_of_vfs> > /sys/bus/pci/devices/<BDF>/sriov_numvfs
```

8.12.1.7.2 7.2. Dynamic MSI-X and Queue Management

Message Signaled Interrupts eXtended (MSI-X) provides efficient interrupt handling for high-performance devices. By default, BlueField distributes MSI-X vectors evenly between all virtual PCIe functions (VFs). However, this is often suboptimal for heterogeneous environments where different VMs require different resource footprints.

Dynamic MSI-X Management allows users to manually control the number of MSI-X vectors provided per VF independently.

8.12.1.7.2.1 7.2.1. Behavior & Persistence

Once configured, the MSI-X link to the VFs remains persistent. Configuration changes only when:

1. The PF explicitly reclaims all VF MSI-X vectors back to the pool (after VF destruction).
2. An Arm reboot (FE reset/cold boot) occurs.

> ⚠ Application restarts, crashes, or closing/reopening PFs without dynamic MSI-X support will not change the configuration.

8.12.1.7.2.2 7.2.2. Configuration Workflow

1. Reset Resources to Free Pool: After creating a **`virtio-fs`** PF device, reclaim all MSI-X vectors from the VFs into the PF's free pool.

```
spdk_rpc.py --plugin --plugin rpc_virtio_fs_tgt -v virtio_fs_device_create \
  --transport_name DOCA \
  --dev_name vfsdev0 \
  --tag doca_test \
  --fsdev aio0 \
  --num_request_queues 1 \
  --queue_size 256 \
  --driver_platform x86_64


spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_device_modify \
  --dev_name <PF_DEV_NAME> \
  --manager mlx5_0 \
  --vuid <PF_DEV_VUID> \

spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_start --dev-name vfsdev0

spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_device_reset_dynamic_resources \
    --dev-name vfsdev0
```

   - *Prerequisite:* No VF devices can be open on the host.

2. Assess Available Resources: Query the controller list to determine the constraints for the PF (**`free_msix`** and **`free_queues`**).

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_get_devices
```

3. Create VF with Specific Allocation: Distribute MSI-X and queues to the VF during creation.
   a. Create the device with specific queues.

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_create \
  --transport_name DOCA \
  --dev_name vfsdev0vf1 \
  --tag doca_test_vf \
  --fsdev aio1 \
  --num_request_queues 4 \
  --queue_size 128 \
  --driver_platform x86_64
```

   b. Modify the device to assign MSI-X vectors.

```
Modify the device to assign MSI-X vectors
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_device_modify \
  --dev_name vfsdev0vf1 \
  --manager mlx5_0 \
  --vuid <VF_DEV_VUID> \
  --num_msix 8
```

   c. Start the device.

```
spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_start --dev-name vfsdev0vf1
```

4. VF Teardown and Reclamation: When destroying a VF, you may choose to release its resources back to the global pool. This can only be done once all VFs are destroyed.
   a. Destroy the VF.

   ```
   spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_device_destroy --dev_name vfsdev0vf1
   ```

   At this point all VF devices are destroyed.
   b. Reclaim resources.

   ```
   spdk_rpc.py --plugin rpc_virtio_fs_tgt -v virtio_fs_doca_device_reset_dynamic_resources \
         --dev-name <PF_DEV_NAME>
   ```

5. Enable on Host: Finally, enable SR-IOV on the host driver.

```
echo <N> > /sys/bus/pci/devices/<BDF>/sriov_numvfs
```

8.12.1.8 8. NFS/TCP XLIO

XLIO is a high-performance user-space TCP stack that offloads traffic handling from the kernel to achieve significant throughput improvements. NFS over TCP leverages XLIO to deliver enhanced performance, utilizing an XLIO socket module integrated directly into SPDK.

> ⚠ The minimum supported version of **libxlio** is 3.60.4.

8.12.1.8.1 8.1. Configuration and Resource Management

XLIO is enabled by default in the container.

8.12.1.8.1.1 8.1.1. Memory Implications

Because XLIO consumes additional memory, you may wish to disable it in non-TCP configurations to conserve resources.

If XLIO is disabled, the required hugepage allocation for the default configuration can be decreased to 4GiB.

8.12.1.8.1.2 8.1.2. Disabling XLIO

To disable XLIO, set the **XLIO_ENABLED** environment variable to **"0"** within the **doca_vfs.yaml** configuration file:

```
env:
  - name: XLIO_ENABLED
    value: "0"
```

8.12.1.9 9. Advanced Configuration

8.12.1.9.1 9.1. NFS/RDMA

8.12.1.9.1.1 9.1.1. Enabling large I/O with accel_mlx5

By default, NFS/RDMA can only transmit up to 40 segments per operation due to the SunRPC protocol limitation. This means that a single I/O can transfer up to 160KB of data.

It is possible to get around this limitation by coalescing the buffers into a single memory key via UMR. This increases the maximum size of an I/O request to 1MB. To achieve this, accel_mlx5 platform driver must be enabled and configured with the support for UMR memory buffer (`umr_memory_buffer`).

Of course, the virtio-fs transport must be created with `max_io_size_kb` configured to handle the larg I/O sizes (see section “Virtio-fs Emulation Management”) .

accel_mlx5 is enabled with the `mlx5_scan_accel_module` RPC, which must run in SPDK STARTUP state (service started with `--wait-for-rpc`).

`mlx5_scan_accel_module` — configures the accel_mlx5 module.

Parameter Name	Optional/ Mandatory	Type	Description
`enable_driver`	Optional	Boolean	Enables the accel_mlx5 platform driver
`umr_memory_buffer`	Optional	Boolean	Enable UMR memory buffer support allowing for coalescing scattered buffers up 1 MB
`allowed_devs`	Optional	String	Comma-separated list of allowed devices (e.g. `mlx5_2,mlx5_3`). If unset, all devices with matching capabilities will be used.

Example:

```
# STARTUP state (service launched with --wait-for-rpc)
mlx5_scan_accel_module --enable-driver --umr-memory-buffer
framework_start_init
```

It is also possible to configure it with JSON configuration file by adding the following section to `accel` subsystem.

```
{
  "subsystem": "accel",
  "config": [
    {
      "method": "mlx5_scan_accel_module",
      "params": {
        "enable_driver": true,
        "umr_memory_buffer": true
      }
    }
  ]
}
```

8.12.1.10 10. Appendix – NVIDIA GPUDirect Storage (GDS) Support in VirtioFS

NVIDIA GPUDirect Storage (GDS) is a technology that enables direct data transfer between storage devices and NVIDIA GPUs. By bypassing the CPU and system memory, GDS significantly reduces data transfer latency and improves bandwidth efficiency for I/O-intensive applications. GDS is particularly beneficial for workloads such as deep learning, high-performance computing (HPC), and large-scale data analytics.

See Getting Started with NVIDIA GPUDirect Storage for more information.

8.12.1.10.1 10.1. Prerequisites

Make sure the following are available:

- NVIDIA GPU with GPUDirect storage support (via P2P DMA)
- Linux kernel with **CONFIG_PCI_P2PDMA** enabled.
- NVIDIA-optimized virtio-fs drivers. Refer to Virtio-fs DOCA Host Drive Installation (Optional).
- NVIDIA CUDA repository (version 13.1 or above). Refer to the CUDA Installation Guide.
- Installed NVIDIA GPU drivers. Refer to the NVIDIA GPUDirect Storage Troubleshooting Guide for additional details.
- Properly configured virtio-fs target

8.12.1.10.2 10.2. Enabling GDS Support in Virtio-fs

1. Configure the system for GDS support:
    a. Enable PCIe peer-to-peer DMA in **libcufile**:

```
// Edit /usr/local/cuda/gds/cufile.json:

// Enable use_pci_p2pdma property
"properties": {
  ...
  "use_pci_p2pdma": true
  ...
}

// Add "virtiofs" and enable use_pci_p2pdma for it under "fs"
"fs": {
   ...
   "virtiofs": { "use_pci_p2pdma": true },
   ...
 }
```

    b. Configure NVIDIA driver parameters – Create a modprobe configuration file to enable the required parameters for the NVIDIA driver:

```
echo 'options nvidia
NVreg_RegistryDwords="RMForceStaticBar1=1;RmForceDisableIomapWC=1;FORCE_P2P=1"' | sudo tee /etc/
modprobe.d/nvidia-gds.conf
```

    c. Configure kernel boot parameters – Edit the **GRUB_CMDLINE_LINUX_DEFAULT** variable in **/etc/default/grub** to add:
        - **nokaslr** to disable Kernel Address Space Layout Randomization.
        - Depending on your CPU:
            - For AMD CPUs: **amd_iommu=off**
            - For Intel CPUs: **intel_iommu=off**

```
#For example:
GRUB_CMDLINE_LINUX_DEFAULT="console=tty0 nokaslr intel_iommu=off"
#After saving the file, update GRUB:
sudo update-grub
```

Then, reboot your system for the changes to take effect.

2. Verify system compatibility:
    a. After rebooting, run the following command to check for GDS support:

```
/usr/local/cuda/gds/tools/gdscheck.py -p | grep -e VIRTIOFS -e properties.use_pci  -e virtiofs.use_
```

You should see an output similar to:

```
VIRTIOFS           : p2pdma, compat
 properties.use_pci_p2pdma : true
 fs.virtiofs.use_pci_p2pdma : true
```

3. Mount the virtio-fs drive with GDS:
    a. To enable GDS workloads on your virtio-fs drive, mount it with the **gds** option.

```
sudo mount -t virtiofs -o gds <tagname> /mnt/vfs/
```

    b. Replace **<tagname>** with the appropriate identifier for your setup. Once mounted, test GDS performance using the **gdsio** tool:

```
/usr/local/cuda/gds/tools/gdsio -T 1 -D /mnt/vfs/ -d0 -I1 -s10m -i1m -w1 -x0
```

This command outputs performance metrics and generates a **cufile.log** in the current working directory. Check the log for any errors or issues.

8.12.1.11 11. Appendix – BlueField Firmware Configuration

Before configuring DOCA SNAP Virtio-fs, the user must ensure that all firmware configuration requirements are met. By default, virtio-fs is disabled and must be enabled by running both common DOCA SNAP Virtio-fs configurations and additional protocol-specific configurations depending on the expected usage of the application (e.g., hot-plug, SR-IOV, UEFI boot, etc).

After configuration is finished, the host must be power cycled for the changes to take effect.

To verify that all configuration requirements are satisfied, users may query the current/next configuration by running the following:

```
mlxconfig -d /dev/mst/mt41692_pciconf0 -e query
```

Note: On some BlueField firmware versions, emulation parameters use the **EMU_MNG** prefix instead of **EMULATION** (for example, **VIRTIO_FS_EMU_MNG_ENABLE** instead of **VIRTIO_FS_EMULATION_ENABLE**). Use the following command to see which names your firmware exposes:

```
mlxconfig -d /dev/mst/mt41692_pciconf0 -e query | grep -E 'VIRTIO_FS|VIRTIO_BLK|NVME'
```

8.12.1.11.1 11.1. System Configuration Parameters

Parameter	Description	Possible Values
INTERNAL_CPU_MODEL	Enable BlueField to work in internal CPU model ⚠ Must be set to 1 for storage emulations.	0/1
PCI_SWITCH_EMULATION_ENABLE	Enable PCIe switch for emulated PFs	0/1
PCI_SWITCH_EMULATION_NUM_PORT	The maximum number of hotplug emulated PFs which equals PCI_SWITCH_EMULATION_NUM_PORT minus 1. For example, if PCI_SWITCH_EMULATION_NUM_PORT=32, then the maximum number of hotplug emulated PFs would be 31. ⚠ One switch port is reserved for all static PFs.	[0-32]

8.12.1.11.2 11.2. RDMA/RoCE Configuration

BlueField's RDMA/RoCE communication is blocked for BlueField's default OS interfaces (nameds ECPFs, typically mlx5_0 and mlx5_1). If RoCE traffic is required, additional network functions (scalable functions) must be added which support RDMA/RoCE traffic.

⚠ The following is not required when working over TCP or even RDMA/IB.

To enable RoCE interfaces, run the following from within the BlueField device:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s PER_PF_NUM_SF=1
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s PF_SF_BAR_SIZE=8 PF_TOTAL_SF=2
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0.1 s PF_SF_BAR_SIZE=8 PF_TOTAL_SF=2
```

8.12.1.11.3 11.3. Virtio-fs Configuration

Before configuring Virtio-fs emulation, ensure the following prerequisites are met:

- A reliable communication channel (e.g., RShim, serial) is available for accessing the Arm subsystem even if the host fails to boot.
- The controller must be created on the static virtio-fs PF using the initial configuration file.

Parameter	Description	Possible Values
`VIRTIO_FS_EMULATION_ENABLE`	Enables virtio-fs device emulation	0/1
`VIRTIO_FS_EMULATION_NUM_PF`	Number of static emulated virtio-fs PFs Misconfiguration may cause the host server OS to fail during boot. Passthrough of PFs to the VM (static hot-plugged) is not allowed.	[0–30]
`VIRTIO_FS_EMULATION_NUM_VF`	Number of vitrual functions per PF Currently up to 128 total VFs across all PFs are supported	[0–128]
`VIRTIO_FS_EMULATION_NUM_MSIX`	Number of MSI-X vectors assigned to the emulated virtio-fs PFs or VFs I/O queue requires one MSI-X vector. Additional vectors are required if the following features are enabled: • Config interrupt: +1 (enabled by default) • High-priority queue: +1 (enabled by default) • Notification queue: +1 (disabled by default) For example, to support 32 I/O queues: • With notification data disabled, set `MSIX` to 34 • With notification data enabled, set `MSIX` to 35	[0–256]

8.12.1.12 12. Appendix – Host OS Configuration

With Linux environment on host OS, additional kernel boot parameters may be required to support DOCA SNAP Virtio-fs related features:

- To use PCIe hotplug, `pci=realloc` must be added
- `modprobe.blacklist=virtio_pci,virtiofs` for the virtio-fs driver which is not built-in
- `modprobe.blacklist=virtio_pci` for the `virtio_pci` driver which is not built-in

To view boot parameter values, run:

```
cat /proc/cmdline
```

It is recommended to use the following command with virtio-fs:

```
[dpu] cat /proc/cmdline BOOT_IMAGE … pci=realloc modprobe.blacklist=virtio_pci,virtiofs
```

8.12.1.12.1 12.1. Intel Server Performance Optimizations

```
cat /proc/cmdline
BOOT_IMAGE=(hd0,msdos1)/vmlinuz-5.15.0_mlnx root=UUID=91528e6a-b7d3-4e78-9d2e-9d5ad60e8273 ro crashkernel=auto
resume=UUID=06ff0f35-0282-4812-894e-111ae8d76768 rhgb quiet pci=realloc modprobe.blacklist=virtio_pci,virtiofs
```

8.12.1.12.2 12.2. AMD Server Performance Optimizations

```
cat /proc/cmdline
cat /proc/cmdline BOOT_IMAGE=(hd0,msdos1)/vmlinuz-5.15.0_mlnx root=UUID=91528e6a-b7d3-4e78-9d2e-9d5ad60e8273 ro
crashkernel=auto resume=UUID=06ff0f35-0282-4812-894e-111ae8d76768 rhgb quiet pci=realloc
modprobe.blacklist=virtio_pci,virtiofs
```

8.12.1.13 13. Appendix – Custom Fsdev Modules

This section explains how to extend the Virtio-fs service with custom fsdev modules. Two methods are provided:

- Building a new production container (recommended for production)
- Manual build and run inside a development container (ideal for development and testing)

The Dockerfiles and commands provided in this chapter are intended as examples and are designed to work with the external fsdev module example included with SPDK, located at **spdk/test/external_code/fsdev_passthru**. These examples demonstrate the integration process and should be adapted to suit real-world use cases, including adjustments for custom fsdev modules, deployment environments, and specific application requirements.

8.12.1.13.1 13.1. Building a New Production Container

This option allows to create a new container that includes both the original service and the custom fsdev module. The resulting container will run Virtio-fs service with custom functionality built-in.

8.12.1.13.1.1 13.1.1. Environment Preparation

This example uses **/opt/build** as working directory but any other directory can be used.

The working directory should contain the **spdk** sources and the **Dockerfile**.

See an example of the Dockerfile in the next section.

```
root@bf:/# mkdir -p /opt/build
root@bf:/# cd /opt/build
root@bf:/opt/build# git clone https://github.com/Mellanox/spdk --branch v25.01.1.nvda
root@bf:/opt/build# ll
total 16
drwxr-xr-x  3 root root 4096 Dec  2 14:16 ./
```

```
drwxr-xr-x  6 root root 4096 Dec  2 13:30 ../
-rw-r--r--  1 root root  723 Dec  2 14:15 Dockerfile
drwxr-xr-x 37 root root 4096 Dec  2 14:01 spdk/
```

8.12.1.13.1.2 13.1.2. Dockerfile Example

```
FROM nvcr.io/nvstaging/doca/doca_vfs:1.2.0-2-doca2.10.0 as builder

RUN apt-get update && apt-get install -y autoconf libtool python3-pyelftools libaio-dev libncurses-dev libfuse3-dev
patchelf libcmocka-dev make

COPY spdk/test/external_code /external_code

WORKDIR /external_code
ENV SPDK_HEADER_DIR=/opt/nvidia/spdk-subsystem/virtiofs/include
ENV SPDK_LIB_DIR=/opt/nvidia/spdk-subsystem/virtiofs/lib
ENV DPDK_LIB_DIR=/opt/nvidia/spdk-subsystem/virtiofs/include
RUN make fsdev_passthru_shared

FROM nvcr.io/nvstaging/doca/doca_vfs:1.2.0-2-doca2.10.0

COPY --from=builder /external_code/fsdev_passthru/libfsdev_passthru_external.so /opt/nvidia/spdk-subsystem/
virtiofs/lib/
ENV VIRTIOFS_LD_PRELOAD=/opt/nvidia/spdk-subsystem/virtiofs/lib/libfsdev_passthru_external.so

COPY --from=builder /external_code/fsdev_passthru/fsdev_passthru.py /usr/lib/python3/dist-packages/spdk/rpc/
ENV SPDK_RPC_PLUGIN="spdk.rpc.rpc_virtio_fs_tgt:spdk.rpc.fsdev_passthru"
```

8.12.1.13.1.3 13.1.3. Building and Running the Container

1. Build the new container:

```
docker build -t doca_vfs_custom_fsdev:latest -f Dockerfile .
```

2. Run the new container with Kubelet service:

```
docker save doca_vfs_custom_fsdev:latest > doca_vfs_custom_fsdev.tar
ctr -n=k8s.io images import doca_vfs_custom_fsdev.tar

# Copy doca_vfs.yaml into /etc/kubelet.d/
# Open doca_vfs.yaml and change "image" section:
    image: doca_vfs_custom_fsdev:latest
```

3. To configure running container with custom fsdev module:

```
# DPU
mkdir /etc/virtiofs/AIO0
crictl exec $(crictl ps -s running -q --name virtiofs) spdk_rpc.py fsdev_aio_create AIO0 /etc/virtiofs/AIO0
crictl exec $(crictl ps -s running -q --name virtiofs) spdk_rpc.py fsdev_passthru_ext_create --name
CUSTOM_FSDEV_PT0 --base-fsdev-name AIO0
crictl exec $(crictl ps -s running -q --name virtiofs) spdk_rpc.py virtio_fs_transport_create -t DOCA
crictl exec $(crictl ps -s running -q --name virtiofs) spdk_rpc.py virtio_fs_doca_manager_create -m mlx5_0
crictl exec $(crictl ps -s running -q --name virtiofs) spdk_rpc.py virtio_fs_transport_start -t DOCA
crictl exec $(crictl ps -s running -q --name virtiofs) spdk_rpc.py virtio_fs_device_create --transport-name
DOCA --dev-name vfsdev0 --tag CUSTOM_FSDEV_PT0 --fsdev CUSTOM_FSDEV_PT0 --num-request-queues 62 --queue-
size 64 --driver-platform x86_64
crictl exec $(crictl ps -s running -q --name virtiofs) spdk_rpc.py virtio_fs_doca_device_modify --dev-name
vfsdev0 --manager mlx5_0 --vuid MT2334XZ0LEDVFSS0D0F3
crictl exec $(crictl ps -s running -q --name virtiofs) spdk_rpc.py virtio_fs_device_start --dev-name
vfsdev0

# x86
modprobe -v virtio_pci
modprobe -v virtiofs
test -d /tmp/CUSTOM_FSDEV_PT0 || mkdir -p /tmp/CUSTOM_FSDEV_PT0
mount -t virtiofs CUSTOM_FSDEV_PT0 /tmp/CUSTOM_FSDEV_PT0
```

8.12.1.13.2 13.2. Manual Build and Run in a Development Container

This approach is suitable for development workflows where frequent updates, builds, and tests are required. It sets up a development container with the necessary environment to manually build and run the Virtio-fs service with a custom fsdev module.

8.12.1.13.2.1 13.2.1. Environment Preparation

> This example uses **/opt/build** as working directory but any other directory can be used.

The working directory should contain the **spdk** sources and the **Dockerfile**.

> See an example of the Dockerfile in the next section.

```
root@bf:/# mkdir -p /opt/build
root@bf:/# cd /opt/build
root@bf:/opt/build# git clone https://github.com/Mellanox/spdk --branch v24.01.4.nvda
root@bf:/opt/build# ll
total 16
drwxr-xr-x  3 root root 4096 Dec  2 14:16 ./
drwxr-xr-x  6 root root 4096 Dec  2 13:30 ../
-rw-r--r--  1 root root  723 Dec  2 14:15 Dockerfile
drwxr-xr-x 37 root root 4096 Dec  2 14:01 spdk/
```

8.12.1.13.2.2 13.2.2. Dockerfile Example

```
FROM nvcr.io/nvstaging/doca/doca_vfs:1.2.0-2-doca2.10.0 as builder

RUN apt-get update && apt-get install -y autoconf libtool python3-pyelftools libaio-dev libncurses-dev libfuse3-dev
patchelf libcmocka-dev make

ENTRYPOINT /bin/bash
```

8.12.1.13.2.3 13.2.3. Building and Running the Development Container

1. Build the container using the provided **Dockerfile**:

   ```
   docker build -t doca_vfs_custom_fsdev_dev:latest -f Dockerfile .
   ```

2. Start the container and enter the bash shell:

   ```
   docker run -ti --privileged --net=host \
       --volume /dev/hugepages:/dev/hugepages \
       --volume /dev/shm:/dev/shm \
       --volume /dev/infiniband:/dev/infiniband \
       --volume /etc/virtiofs:/etc/virtiofs \
       --volume ${PWD}/spdk/test/external_code:/external_code \
       doca_vfs_custom_fsdev_dev:latest
   ```

3. Once inside the container, one can manually build and test custom module:

   ```
   cd /external_code/
   export SPDK_HEADER_DIR=/opt/nvidia/spdk-subsystem/virtiofs/include
   export SPDK_LIB_DIR=/opt/nvidia/spdk-subsystem/virtiofs/lib
   export DPDK_LIB_DIR=/opt/nvidia/spdk-subsystem/virtiofs/include
   make fsdev_passthru_shared
   ```

4. Start the virtio-fs service with the custom module for testing:

   ```
   cp /external_code/fsdev_passthru/fsdev_passthru.py /usr/bin/
   /opt/nvidia/spdk-subsystem/doca_snap/bin/set_environment_variables.sh
   LD_PRELOAD=/external_code/fsdev_passthru/libfsdev_passthru_external.so \
       /opt/nvidia/spdk-subsystem/virtiofs/bin/virtio_fs_tgt -m 0xff &
   ```

5. To configure the service with custom fsdev module:

   ```
   # DPU
   mkdir /etc/virtiofs/AIO0
   ```

```
spdk_rpc.py fsdev_aio_create AIO0 /etc/virtiofs/AIO0
spdk_rpc.py --plugin fsdev_passthru fsdev_passthru_ext_create --name CUSTOM_FSDEV_PT0 --base-fsdev-name
AIO0
spdk_rpc.py virtio_fs_transport_create -t DOCA
spdk_rpc.py virtio_fs_doca_manager_create -m mlx5_0
spdk_rpc.py virtio_fs_transport_start -t DOCA
spdk_rpc.py virtio_fs_device_create --transport-name DOCA --dev-name vfsdev0 --tag CUSTOM_FSDEV_PT0 --fsdev
CUSTOM_FSDEV_PT0 --num-request-queues 62 --queue-size 64 --driver-platform x86_64
spdk_rpc.py virtio_fs_doca_device_modify --dev-name vfsdev0 --manager mlx5_0 --vuid MT2334XZ0LEDVFSS0D0F2
spdk_rpc.py virtio_fs_device_start --dev-name vfsdev0

# x86
modprobe -v virtio_pci
modprobe -v virtiofs
test -d /tmp/CUSTOM_FSDEV_PT0 || mkdir -p /tmp/CUSTOM_FSDEV_PT0
mount -t virtiofs CUSTOM_FSDEV_PT0 /tmp/CUSTOM_FSDEV_PT0
```

8.12.1.14 14. Appendix – Custom Bdev Modules

With DOCA SNAP-4 block emulation subsystems enabled (as described in DOCA SNAP Virtio-fs with SNAP Support), DOCA SNAP virtio-fs service allows to load custom bdev modules.

> Refer to *DOCA SNAP-4 Service Guide* documentation for more information.

8.12.1.15 15. Appendix – Linux NFS Target

When using the SPDK NFS fsdev module to access shares from Linux NFS target, it's important to ensure that the target is up-to-date. Some versions of the kernel have a bug in the SunRPC code that can cause the NFS target to stop acknowledging incoming TCP packets causing the receive window to get to zero making the connection unusable. It's prone to happen on specific workloads and causes the NFS fsdev to report PDU timeouts and initiate reconnects to recover from this situation. This results in IO stalls and severely degraded performance. The faulty change was introduced in Linux v6.6 and has later been reverted in v6.14. The fix has been also cherry-picked in Ubuntu's 6.8.0-64-generic kernel. It is strongly advised to use kernels that include the fix.

When using the SPDK NFS **fsdev** module to access shares from a Linux NFS target, you must ensure the target kernel is up to date to avoid severe performance degradation.

- Issue – A known bug in the Linux kernel's SunRPC code causes the NFS target to abruptly stop acknowledging incoming TCP packets. This shrinks the TCP receive window to zero, rendering the connection unusable.
- Impact – Under specific workloads, this bug causes the NFS **fsdev** to report PDU timeouts and initiate forced reconnections in an attempt to recover. This results in significant I/O stalls and severely degraded storage performance.
- Resolution – The faulty commit (originally introduced in Linux v6.6) was reverted in Linux v6.14. This fix has also been cherry-picked into Ubuntu's **6.8.0-64-generic** kernel. It is strongly advised to upgrade your NFS target to a kernel containing this fix before deploying.

8.12.1.16 16. References

Title	Description
NVIDIA DOCA	NVIDIA DOCA™ SDK enables developers to rapidly create applications and services on top of NVIDIA® BlueField® networking platform, leveraging industry-standard APIs
NVIDIA BlueField BSP	BlueField Board Support Package includes the bootloaders and other essentials for loading and setting software components
BlueField DPU Hardware User Manual	This document provides details as to the interfaces of the BlueField DPU, specifications, required software and firmware for operating the device, and a step-by-step plan for bringing the DPU up
NVIDIA BlueField BSP Documentation	This document provides product release notes as well as information on the BlueField software distribution and how to develop and/or customize applications, system software, and file system images for the BlueField platform
DOCA Device Emulation	DOCA Device Emulation library documentation. The DOCA Device Emulation subsystem provides a low-level software API for users to develop PCIe devices and their controllers.
DOCA DevEmu Virtio-fs	DOCA Device Emulation Virtio-fs library documentation. The DOCA DevEmu Virtio-fs library is part of the DOCA DevEmu Virtio subsystem. It provides low-level software APIs that provide building blocks for developing and manipulating virtio filesystem devices using the device emulation capability of BlueField platforms.
DOCA DevEmu PCI	DOCA Device Emulation PCI library documentation. DOCA DevEmu PCI is part of the DOCA Device Emulation subsystem. It provides low-level software APIs that allow management of an emulated PCIe device using the emulation capability of NVIDIA® BlueField® networking platforms.

8.12.1.17 SNAP Virtio-fs Service Release Notes

The release notes provide information for the DOCA SNAP Virtio-fs Service such as changes and new features, software known issues, and bug fixes.

8.12.1.17.1 Key Features in Version 1.6.0-doca3.3.0

- Various bug fixes to enhance system stability.

8.12.1.17.2 Limitations

The following features are not supported in this version:

- Live update and live migration

8.12.1.17.3 Known Issues

8.12.1.17.3.1 DOCA SNAP Virtio-fs Issues

Ref #	Issue
–	Description: The NFS **fsdev** module does not garbage collect its file objects. Once referenced, internal file objects persist unless the associated file is deleted, leading to stale object accumulation. When the per-**fsdev** limit of 1 million objects is reached, new file creation and directory listing operations fail.
	Workaround: N/A
	Keywords: File deletion/creation
	Discovered in version: 1.4.0-doca3.1.0
–	Description: For hotplugged devices, mounting the device immediately after loading the driver may fail.
	Workaround: Introduce a minimum delay of 3 seconds between driver load and mount operations to ensure proper device readiness.
	Keywords: Wrong fs type; bad option; bad superblock; tag not found
	Discovered in version: 1.4.0-doca3.1.0
–	Description: Repeatedly running **virtio_fs_doca_device_hotplug** and **virtio_fs_doca_device_hotunplug** RPCs in a loop can lead to recovery-related issues.
	Workaround: Insert a minimum delay of 2 seconds between the **hotplug** and **hotunplug** operations to ensure system stability.
	Keywords: Recovery failed, DOCA error
	Discovered in version: 1.4.0-doca3.1.0
–	Description: Invoking the **virtio_fs_device_hotunplug** RPC while the device is still mounted on the host, followed by a **virtio_fs_device_hotplug** call, may result in I/O errors on the mount point.

Ref #	Issue
	Workaround: Before issuing the `virtio_fs_device_hotunplug` RPC, ensure that the emulated file system is no longer mounted on the host. Use the `umount` command to cleanly unmount the device's mount point.
	Keywords: Hotunplug; mount
	Discovered in version: 1.3.0-doca3.0.0
4346308	Description: Under high-scale scenarios, the `virtio_fs_doca_manager_destroy` RPC may time out.
	Workaround: Run the RPC with an extended timeout value. For example: `rpc.py -t 120 virtio_fs_doca_manager_destroy -m mlx5_0`.
	Keywords: Timeout
	Discovered in version: 1.3.0-doca3.0.0
–	Description: The Linux Kernel NFS RDMA server is known to handle incoming connections slowly. At high scale (e.g., when using multiple hotplug functions), this can result in connection timeouts.
	Workaround: Introduce a delay between mount operations of different hotplug functions on the host to avoid overwhelming the NFS RDMA server.
	Keywords: Timeout, hotplug, mount
	Discovered in version: 1.3.0-doca3.0.0
4422332	Description: The combination of virtio-fs, virtio-blk, and InfiniBand in a single use case is not supported.
	Workaround: N/A
	Keywords: IB
	Discovered in version: 1.3.0-doca3.0.0
–	Description: I/O operations larger than 255 KB are not supported due to a data pool size limitation in the VirtioFS process running on the DPU.
	Workaround: Set the `max_xfer_size` parameter in the `fsdev_aio_create` RPC to a value less than 255 KB. This parameter is used as `max_write` in the FUSE layer of the host kernel. If an I/O operation exceeds `max_xfer_size`, the FUSE layer automatically splits the request into smaller chunks. The default `max_xfer_size` value is 128 KB. For example, if an I/O request is 1 MB and `max_xfer_size` is set to 64 KB, the FUSE layer on the host splits the operation into 16 separate 64 KB chunks, which are then individually handled by the virtio-fs process on the DPU.
	Keywords: IO error; host error; io_size
	Discovered in version: 1.2.0-doca2.10.0
–	Description: Due to the lack of recovery support, negative or resilience operations (e.g., destroy and restore) are not supported during active I/O traffic. Restarting the device using `virtio_fs_device_stop` followed by `virtio_fs_device_start` is supported in such cases.
	Workaround: N/A
	Keywords: Recovery; negative/resilience operations

Ref #	Issue
	Discovered in version: 1.0.0-doca2.8.0
4862565	Description: The following FUSE commands are unsupported: `BMAP`, `SETUPMAPPING`, `REMOVEMAPPING, STATX`
	Workaround: N/A
	Keywords: FUSE
	Discovered in version: 1.0.0-doca2.8.0
–	Description: Application restart is not supported if the application controller has already processed FUSE commands.
	Workaround: Unload the virtio-fs driver on the host before restarting the application.
	Keywords: FUSE
	Discovered in version: 1.0.0-doca2.8.0
–	Description: The following operations are not supported when using Linux's virtio-fs inbox/upstream kernel driver: FLR and the virtio-fs notification queue.
	Workaround: N/A
	Keywords: FLR; virtio-fs; inbox/upstream kernel driver
	Discovered in version: 1.0.0-doca2.8.0
4687235	Description: Driver load on host or "echo <count> > /sys/bus/pci/devices/0000\:07\:00.2/sriov_numvfs" can get stuck forever due to resource constraint in DPU
	Workaround: Restart the virtio_fs_tgt application
	Keywords: flexio, process stuck
	Discovered in version: 1.5.0-doca3.2.0
4692130	Description: Triggering an FLR using `echo 1 > /sys/bus/pci/devices/0000:07:00.2/reset` on a PF with active VFs causes the VFs to become unusable.
	Workaround: N/A
	Keywords: FLR, VF
	Discovered in version: 1.5.0-doca3.2.0
4552213	Description: Certain file operation sequences when using an aio fsdev backed by a Linux kernel NFS mount may result in silly-renamed files (with `.nfs` prefix) that are not automaticall removed.
	Workaround: They can be removed by dropping caches (`echo 3 > /proc/sys/vm/drop_caches`) from the host.
	Keywords: AIO
	Discovered in version: 1.5.0-doca3.2.0
4961367	Description: Enabling ASAN in debug mode while using the source package can lead to a heap-use-after-free issue.
	Workaround: N/A
	Keywords: source package, ASAN, debug
	Discovered in version: 1.7.0-doca3.4.0

Ref #	Issue
4990959	Description: Recovery may fail if the process is terminated before shared memory (shm) initialization completes.
	Workaround: N/A
	Keywords: Recovery
	Discovered in version: 1.7.0-doca3.4.0
5000400	Description: Repeated events such as driver load/unload or device hotplug/hot-unplug that create and destroy virtqueues can eventually lead to resource exhaustion, causing virtqueue failures. This occurs when the process is running more than 1 ARM core.
	Workaround: Restart the virtiofs process
	Keywords: resource exhaustion, virtqueue fail
	Discovered in version: 1.7.0-doca3.4.0
5037109	Description: The maximum supported scale for virtio-blk controllers is 1000.
	Workaround: N/A
	Keywords: virtio-blk
	Discovered in version: 1.80-doca3.5.0
5210707	Description: During large-scale SR-IOV FLR/reset storms, virtio-fs device stop/destroy RPCs may time out if the target device is still being reset.
	Workaround: etry the RPC after the FLR/reset activity completes, or wait until device state is idle before issuing stop/destroy.
	Keywords: SRIOV, FLR, VF Reset, RPC timeout
	Discovered in version: 1.80-doca3.5.0

8.12.1.17.3.2 OS or Vendor Issues

Ref #	Issue
–	Description: Triggering a Function Level Reset (FLR) from the host by writing `1` to the reset file under `/sys/bus/pci/devices/<bdf>/` (e.g., `echo 1 > /sys/bus/pci/devices/0000:29:00.2/reset`) causes the host driver to skip virtqueue creation. As a result, the mount point becomes unresponsive.
	Workaround: Avoid initiating FLR while a virtio-fs mount is active on the host. To resume I/O after FLR, unload and reload the virtio-fs driver on the host.
	Keywords: Driver; FLR
	Discovered in version: 1.0.0-doca2.8.0
–	Description: On the host, when the virtio-fs mount is idle (i.e., no I/O operations), the `dmesg` log is repeatedly filled with AppArmor DENIED messages. These messages indicate that the `ntpd` service is being denied access to specific files, such as `/snap/bin/` and `/etc/ssl/openssl.cnf`, due to restrictions in its AppArmor profile.

Ref #	Issue

	Workaround: Modify the AppArmor profile for **ntpd** to grant the necessary read permissions.
	1. Locate the AppArmor profile for **ntpd** (typically located at **/etc/apparmor.d/usr.sbin.ntpd**).
	2. Edit the profile and add the permissions using a text editor. For example:
	a. Run:
	`sudo nano /etc/apparmor.d/usr.sbin.ntpd`
	b. Add the following lines to allow ntpd to read the necessary files:
	`/snap/bin/ r,`
	`/etc/ssl/openssl.cnf r,`
	3. Reload the AppArmor profile to apply the changes:
	`sudo apparmor_parser -r /etc/apparmor.d/usr.sbin.ntpd`

	Keywords: AppArmor, ntpd
	Discovered in version: 1.0.0-doca2.8.0
3949202	Description: On kernels older than version 6.10, repeatedly loading and unloading the **virtio_pci** and **virtiofs** drivers (or mounting and unmounting filesystems using these drivers) can cause the drivers to hang or become unresponsive.
	Workaround: Insert a delay of at least 1 second between load and unload operations to reduce the risk of driver instability.
	Keywords: Virtio_pci; virtiofs
	Discovered in version: 1.0.0-doca2.8.0
-	Description: FUSE_STATX was introduced in FUSE version 7.39. If the negotiated version is less than this, kernel can still send FUSE_STATX.
	Workaround: N/A
	Keywords: Unsupported opcode
	Discovered in version: 1.4.0-doca3.1.0
-	Description: On modern Linux distributions, unplugging a PCIe function from the host while there are inflight I/Os can cause the virtiofs driver to hang and virtio_fs_doca_device_hotunplug RPC will timeout
	Workaround: N/A
	Keywords: Hotplug, Linux
	Discovered in version: 1.5.0-doca3.2.0

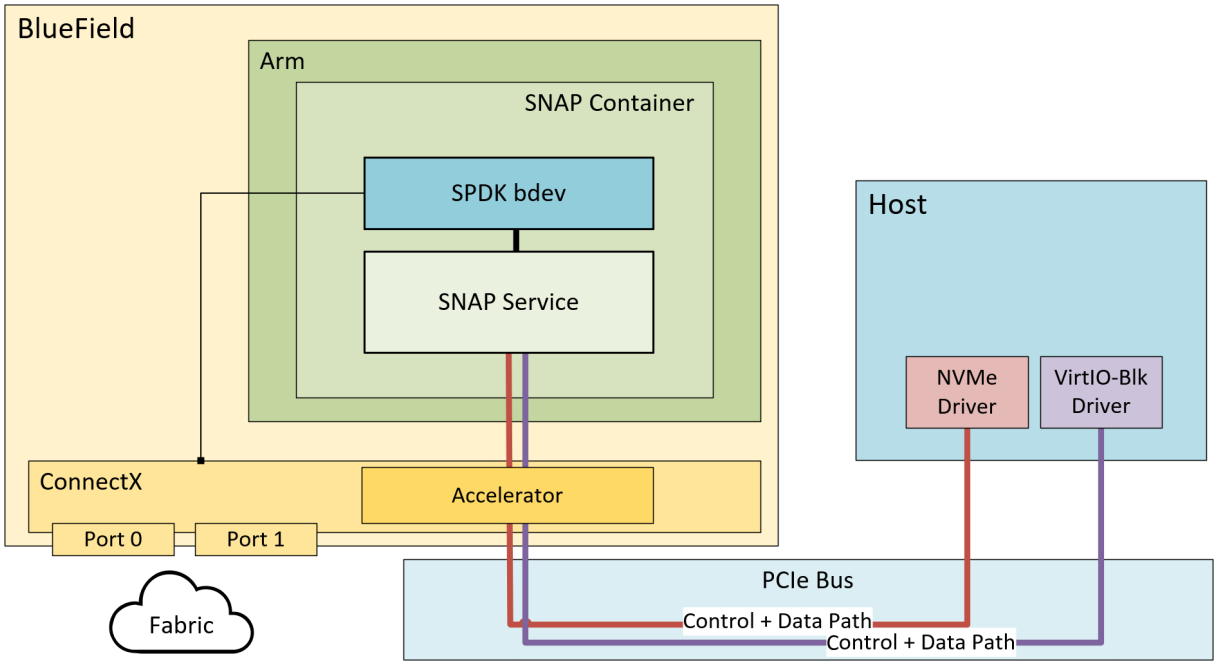
8.12.2 DOCA SNAP-4 Service Guide

This guide provides instructions on using the DOCA SNAP-4 service on top of the NVIDIA® BlueField®-3 DPU and later.

8.12.2.1 Introduction

NVIDIA® BlueField® SNAP and virtio-blk SNAP (storage-defined network accelerated processing) technology enables hardware-accelerated virtualization of local storage. NVMe/virtio-blk SNAP presents networked storage as a local block-storage device (e.g., SSD) emulating a local drive on the PCIe bus. The host OS or hypervisor uses its standard storage driver, unaware that communication is done, not with a physical drive, but with NVMe/virtio-blk SNAP framework. Any logic may be applied to the I/O requests or to the data via the NVMe/virtio-blk SNAP framework prior to redirecting the request and/or data over a fabric-based network to remote or local storage targets.

NVMe/virtio-blk SNAP is based on the NVIDIA® BlueField® DPU family technology and combines unique software-defined hardware-accelerated storage virtualization with the advanced networking and programmability capabilities of the DPU. NVMe/virtio-blk SNAP together with the BlueField DPU enable a world of applications addressing storage and networking efficiency and performance.



The traffic arriving from the host towards the emulated PCIe device is redirected to its matching storage controller opened on the **mlnx_snap** service.

The controller implements the device specification and may expose backend device accordingly (in this case, SPDK is used as the storage stack that exposes backend devices). When a command is received, the controller executes it.

Admin commands are mostly answered immediately, while I/O commands are redirected to the backend device for processing.

The request-handling pipeline is completely asynchronous, and the workload is distributed across all Arm cores (allocated to SPDK application) to achieve the best performance.

The following are key concepts for SNAP:

- Full flexibility in fabric/transport/protocol (e.g. NVMe-oF/iSCSI/other, RDMA/TCP, ETH/IB)
- NVMe and virtio-blk emulation support
- Programmability
- Easy data manipulation
- Allowing zero-copy DMA from the remote storage to the host
- Using Arm cores for data path

> ⚠ BlueField SNAP for BlueField-2 DPU is licensed software. Users must purchase a license per BlueField-2 DPU to use them. BlueField-3 and BlueField-4 DPU does not have license requirements to run BlueField SNAP.

8.12.2.1.1 SNAP as Container

In this approach, the container could be downloaded from NVIDIA NGC and could be easily deployed on the DPU.

The yaml file includes SNAP binaries aligned with the latest **spdk.nvda** version. In this case, the SNAP sources are not available, and it is not possible to modify SNAP to support different SPDK versions (SNAP as an SDK package should be used for that).

> ⚠ SNAP 4.x is not pre-installed on the BFB but can be downloaded manually on demand.

For instructions on how to install the SNAP container, please see "SNAP Container Deployment".

8.12.2.1.2 SNAP as a Package

The SNAP development package (custom) is intended for those wishing to customize the SNAP service to their environment, usually to work with a proprietary bdev and not with the **spdk.nvda** version. This allows users to gain full access to the service code and the lib headers which enables them to compile their changes.

8.12.2.1.2.1 SNAP Emulation Lib

This includes the protocols libraries and the interaction with the firmware/hardware as well as:

- Plain shared objects (***.so**)
- Static archives (***.a**)
- pkgconfig definitions (***.pc**)
- Include files (***.h**)

8.12.2.1.2.2 SNAP Service Sources

This includes the following managers:

- Emulation device managers:
- - Emulation manager – manages the device emulations, function discovery, and function events
  - Hotplug manager – manages the device emulations hotplug and hot-unplug

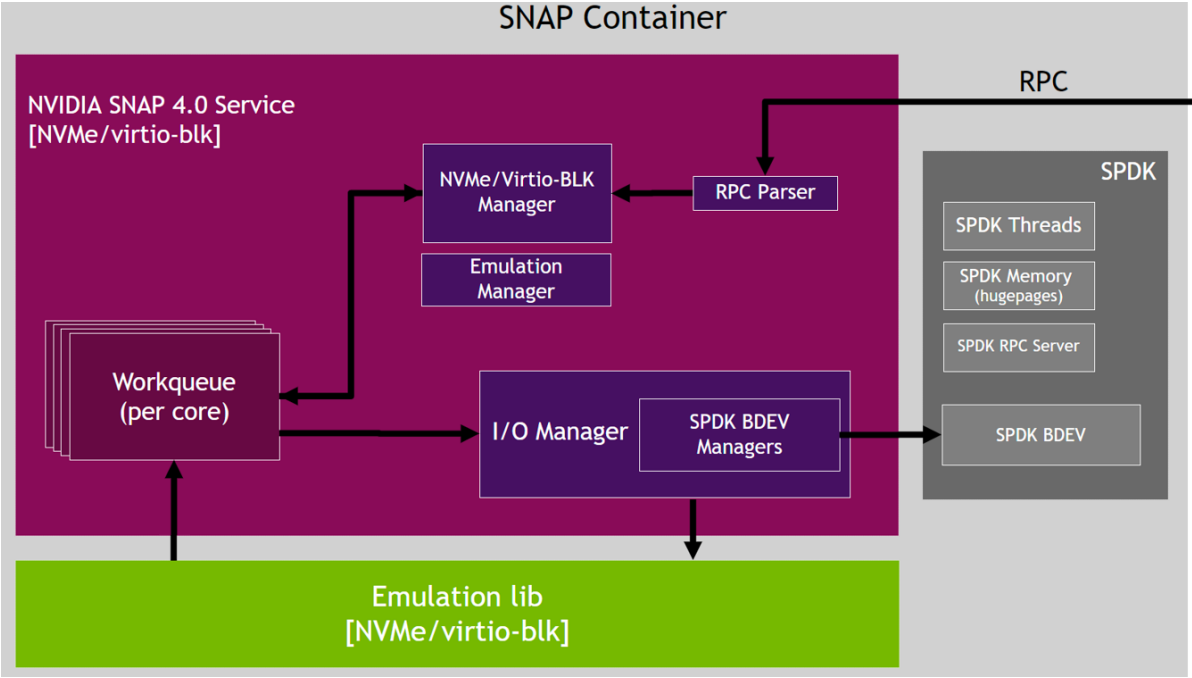
    - Config manager – handles common configurations and RCPs (which are not protocol-specific)
- Service infrastructure managers:
    - Memory manager – handles the SNAP mempool which is used to copy into the Arm memory when zero-copy between the host and the remote target is not used
    - Thread manager – handles the SPDK threads
- Protocol specific control path managers:
    - NVMe manager – handles the NVMe subsystem, NVMe controller and Namespace functionalities
    - VBLK manager – handles the virtio-blk controller functionalities
- IO manager:
    - Implements the IO path for regular and optimized flows (RDMA ZC and TCP XLIO ZC)
    - Handles the bdev creation and functionalities

SNAP Service Dependencies

SNAP service depends on the following libraries:

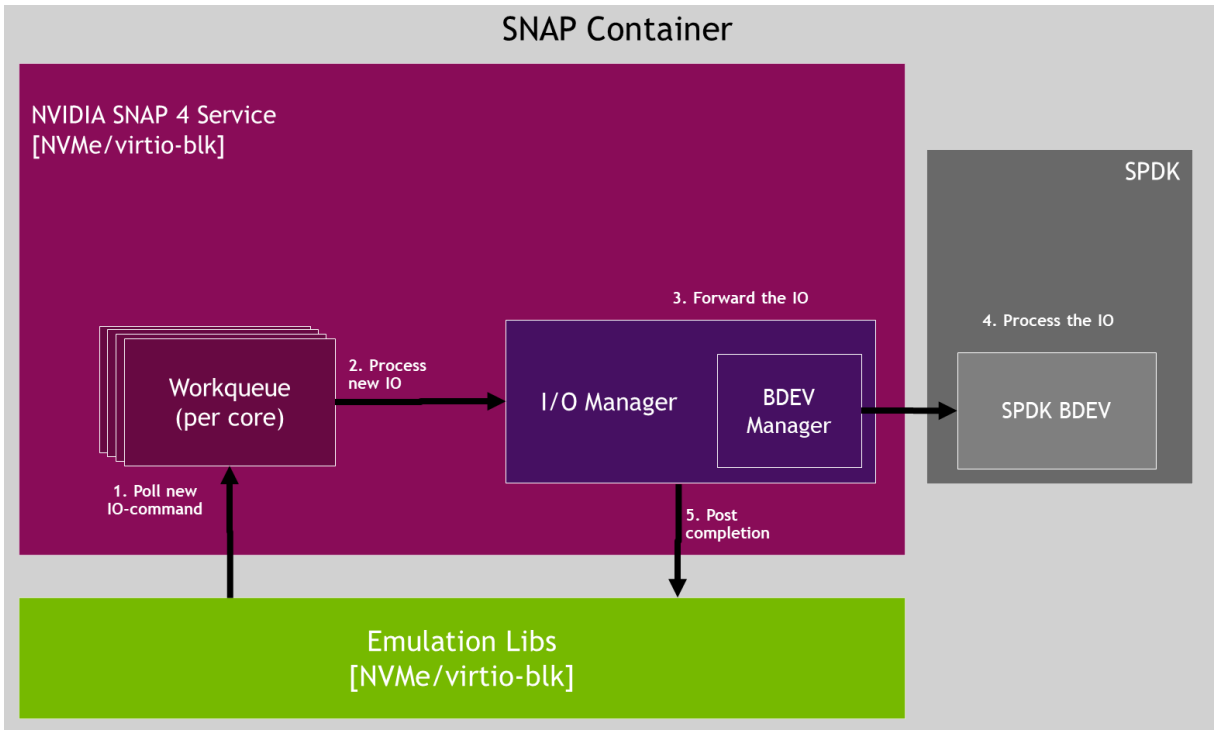
- SPDK – depends on the bdev and the SPDK resources, such as SPDK threads, SPDK memory, and SPDK RPC service
- XLIO (for NVMeTCP acceleration)

SNAP Service Flows

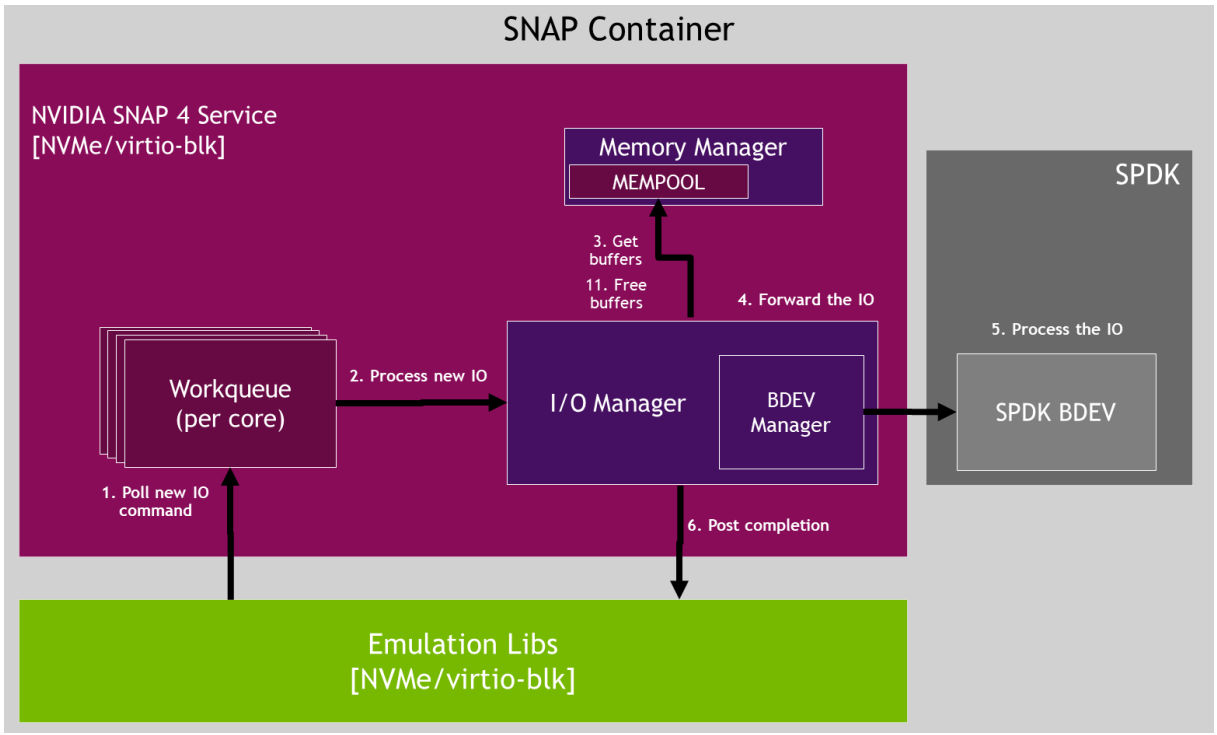


IO Flows

Example of RDMA zero-copy read/write IO flow:



Example of RDMA non-zero-copy read IO flow:



8.12.2.1.2.3 Data Path Providers

SNAP facilitates user-configurable providers to assist in offloading data-path applications from the host. These include: Device emulation, IO-intensive operations, and DMA operations.

Data Path Providers	Description	Support
DPA provider	DPA (data path accelerator) is a cluster of multi-core and multi-execution-unit RISC-V processors embedded within the BlueField.	NVMe, Virtio-blk
DPU provider	Hybrid mode that handles the data-path applications from the host using the BlueField CPU and the DPA hardware. This mode improves IO latency and reduces SNAP downtime during crash recovery.	Virtio-blk

DPA is the default provider in SNAP for NVMe and virtio-blk.

To switch provider, use the environment variable `VIRTIO_EMU_PROVIDER=dpu` to modify the the variable on the YAML. Refer to the "SNAP Environment Variables" page for more information.

8.12.2.2 SNAP-4 Service Release Notes

This page describes new features, known issues, and bug fixes for NVIDIA® BlueField®-3 SNAP software.

8.12.2.2.1 Changes and New Features in NVIDIA SNAP 4.12.0

New features:

Bug fixes:

- Various bug fixes to enhance system stability.

Live update Compatibility notes:

- Live update is supported only between consecutive versions for example between 4.11.0 and 4.12.0.

8.12.2.2.2 Bug Fixes

Ref #	Issue Details
4949600	Description: SNAP supports live downgrade (moving from newer to older SNAP version) from any version since 4.12.0 to any version since 4.12.0.
	Keywords: Live update; NVMe; virtio-blk
	Fixed in version: 4.12.0

8.12.2.2.3 Known Issues

8.12.2.2.3.1 SNAP Issues

Ref #	Issue Details
5149438	Description: When the host **nvme** driver is loaded or unloaded during live update with a high scale (100+) of NVMe SR-IOV VFs, admin commands may time out, resulting in a missing device in **nvme list** or in the host driver getting stuck, requiring a host reboot.
	Workaround: increase the host NVMe admin timeout (**options nvme_core admin_timeout=180**).
	Keywords: NVMe, live update, SR-IOV
	Discovered in version: 4.12.0
4964014	Description: When the dynamic MSI-X feature is enabled, if an FLR is triggered and the VF controller is created before its PF controller, the **num_msix** modify operation may fail on the VF controller. In this case, the commands **nvme_controller_modify --num_msix** and **nvme_controller_destroy --release_msix** may fail.
	Workaround: Run **nvme_controller_destroy** (without **--release_msix** flag) and re-create controller for the VF.
	Keywords: Dynamic msix; **num_msix** ; VF; FLR
	Discovered in version: 4.11.0
4953443	Description: Live update for NVMe VFs is not supported from older snap version to 4.11.0 when using 4.11.0 firmware.
	Workaround: Use 4.11.0 SNAP when updating to 4.11.0 firmware.
	Keywords: NVMe; live update
	Discovered in version: 4.11.0
4829857	Description: **emulation_resource_list** is not updated following NVMe emulation hotplug/ unplug/modify operations.
	Workaround: create all desired hotplug functions (**nvme_function_create** RPC) in advance before exposing it to host, to make sure there are sufficient resources (and gradually decrease if allocation fails)
	Keywords: NVMe; hotplug
	Discovered in version: 4.10.0
4836095	Description: SNAP does not support live downgrade (moving from newer to older SNAP version)
	Workaround: Stop existing (new) SNAP process, then start the desired one (old) instead
	Keywords: Live update; NVMe; virtio-blk
	Discovered in version: 4.9.0
4796121	Description: SNAP configuration recovery is unsupported in combination with live update. If it crashes config file may contain partial configuration.
	Workaround: Use different **SNAP_RPC_INIT_CONF_JSON** for src and dest. Save a copy of json config file before LU. If snap crashes during LU, kill both src and dest and replace the config file with the saved version. Recover with the saved config file and retry the LU flow.

Ref #	Issue Details
	Keywords: Configuration recovery; live update
	Discovered in version: 4.9.0
4691998	Description: On systems running Linux Kernel 6.14 or higher, triggering an FLR (**`echo 1 >sys/bus/pci/devices/0000:07:00.2/reset`**) on a PF while VFs are active causes the VFs to become unusable.
	Workaround: Choose one of the following: • Destroy or unbind all active VFs associated with the PF before triggering the reset. • After performing the PF FLR, triggering FLRs on all associated VFs restores VF functionality.
	Keywords: Virtio-blk; FLR; SR-IOV
	Discovered in version: 4.9.0
4561583	Description: NVMe drivers may experience I/O timeouts if **`queue_depth`** is configured to a value that is not a power of two. This applies to both the driver parameter (**`io_queue_depth`**) and the queue depth set during controller creation (**`nvme_controller_create`**).
	Workaround: N/A
	Keywords: NVMe; queue_depth
	Discovered in version: 4.9.0
4636221	Description: If SNAP crashes during live migration (LM), running recovery may cause subsequent LM commands to fail.
	Workaround: N/A
	Keywords: Live migration; recovery
	Discovered in version: 4.9.0
4499506	Description: Live update fails when multiple remote bdevs are configured, due to a QP creation timeout.
	Workaround: N/A
	Keywords: Live update; remote bdev
	Discovered in version: 4.8.0
4561599	Description: NVMe UEFI expansion ROM driver requires at least 2 IO queues to be supported by SNAP controller.
	Workaround: Use **`--num_queues=2`** (or higher) with **`nvme_controller_create`** RPC for NVMe boot devices.
	Keywords: NVMe; UEFI boot
	Discovered in version: 4.8.0
4499506 4502937	Description: When working with high scale (>=16) of virtio-blk or NVMe functions, backed by SPDK's NVMe-oF/RDMA bdevs as a backend, live update process may fail.
	Workaround: Use "regular" update process (close + reopen SNAP application).
	Keywords: NVMe; virtio-blk; live update; SPDK; NVMe-oF/RDMA
	Reported in version: 4.8.0
4513104	Description: If the virtio-blk driver in the VM is unprobed or there is an FLR on the VM function during virtio-blk VFIO live migration, the driver may get stuck.

Ref #	Issue Details
	Workaround: N/A
	Keywords: Virtio-blk; live migration; FLR; reset; driver unprobe
	Reported in version: 4.8.0
-	Description: Linux kernel's **virtio-blk** driver expects **--size_max** to be at least the size of the of OS page. If it is not, a driver warning will be prompted on the host side.
	Workaround: Use the **--size_max** option with a value equal to. the OS page size (typically 4096).
	Keywords: Virtio_blk; Linux; kernel; size_max
	Reported in version: 4.7.0
4396707	Description: SPDK's **virtio-blk** driver uses 4K segment sizes, even though **size_max** technically allows for larger segments. This might cause large logical block size IO handling to fail.
	Workaround: When using the SPDK **virtio-blk** driver, always set the **--size_max** option with a value of 4096.
	Keywords: Virtio_blk; size_max; SPDK
	Reported in version: 4.7.0
4409344	Description: When performing live update too fast (using automated script), destination process might not yet create all necessary resources, when prompted to handshake with source process.
	Workaround: Add a **sleep(3)** between creation of controller on destination process (with **--live_update_listener** flag) and the succeeding request to suspend source controller (with **--live_update_notifier** flag). That will increase the overall automated script runtime but should not affect the host driver's downtime.
	Keywords: Virtio_blk; NVMe; live update
	Reported in version: 4.7.0
4412341	Description: When using high scale (512<=) of virtio-blk VFs on a single PF, sudden hypervisor crash (or brutal warm reboot) may result in hypervisor hang, due to the long FLR processing time.
	Workaround: Split the opened VFs among more PFs; gracefully shutdown VMs before performing hard OS reset
	Keywords: Virtio-blk; FLR; SR-IOV
	Reported in version: 4.7.0
4104709	Description: Some legacy operating systems (e.g., RockyLinux with kernel 4.18) issues virtio-blk zero-length I/Os during boot (e.g., during EDD probing).
	Workaround: Set **VIRTIO_BLK_ZERO_LEN_IO_FAIL=1** environment variable to configure SNAP to fail zero-length I/Os in virtio-blk.
	Keywords: Virtio-blk; zero-length I/Os
	Reported in version: 4.6.0
-	Description: When using both NVMe and Virtio-blk protocols in SNAP, their data providers may share the same DPA HARTS, potentially causing NVMe configuration or performance issues. This is especially relevant when Virtio-blk is in DPU mode and NVMe is in DPA mode.

Ref #	Issue Details
	Workaround: Use `dpa_helper_core_mask` and `dpa_nvme_core_mask` environment variable as DPA core masks, for example `dpa_helper_core_mask=0x0000FFFF` and `dpa_nvme_core_mask=0xFFFF0000`
	Keywords: Virtio-blk and NVMe; DPA core mask.
	Reported in version: 4.6.0
4110943	Description: Hot-unplugging a hotplugged Virtio BLK device is not allowed unless a Virtio BLK controller has previously been created for the device.
	Workaround: Create a Virtio BLK controller on the device which need to do hotunplug
	Keywords: hotplug; hotunplug
	Discovered in version: 4.5.0
3631346	Description: When using dynamic MSIX with NVMe protocol, the `free_queues` PF controller property is not valid and always shows `0`.
	Workaround: Ignore the value and assume the `free_queues` pool is large enough.
	Keywords: NVMe; MSIX
	Discovered in version: 4.4.1
-	Description: The SPDK `bdev_uring` is not supported.
	Workaround: N/A
	Keywords: NVMe
	Discovered in version: 4.4.0
3817040	Description: When running `nvme_controller_suspend` RPC with the `--timeout` parameter, the device is no longer operational and cannot be resumed if the timeout expires.
	Workaround: Destroy and re-create the controller.
	Keywords: NVMe
	Discovered in version: 4.4.0
3809646	Description: When working with a new DPA provider, if an interrupt is sent to DPA immediately after a DMA operation, DPA may wake up before the DMA is fully written to the buffer, causing it to miss events.
	Workaround: Add a software-based periodic wake-up mechanism.
	Keywords: NVMe
	Discovered in version: 4.4.0
3773346	Description: When configuring the virtio-blk controller, using an unaligned `size_max` value with the SPDK NVMe-oF initiator as the backend can lead to memory corruption.
	Workaround: `size_max` and `seg_max` values must be a power of 2.
	Keywords: Virtio-blk; NVMe-oF; spdk
	Discovered in version: 4.3.1
3745842	Description: When using the NVMe/TCP SPDK block device as a backend, SNAP is limited to working with no more than 8 cores.
	Workaround: Work with Arm core mask which uses only 8 cores.

Ref #	Issue Details
	Keywords: NVMe; TCP; SPDK
	Discovered in version: 4.3.1
-	Description: Container images may become corrupted, resulting in a container status of **exited** with the error message **/usr/bin/supervisord: exec format error** .
	Workaround: Remove the YAML from kubelet, use **crictl images** to list the images and **crictl rmi <image-id>** to remove the images. Run **systemctl restart containerd** and **systemctl restart kubelet** , then copy the YAML file again to kubelet.
	Keywords: NGC; container image
	Discovered in version: 4.3.1
3757171	Description: When running virtio-blk emulation with large IOs (>128K) and SPDK's NVMe-oF initiator as a backend, IOs might fail in the SPDK layer due to poor alignment.
	Workaround: **size_max** value of **virtio_blk_controller_create** RPC must be set and be a power of 2.
	Keywords: SPDK; virtio-blk; size_max
	Discovered in version: 4.3.1
3689918 3753637	Description: The SNAP container takes a long time to start up when configured with a large number of emulations, potentially exceeding the default NVMe driver timeout.
	Workaround: Increase the NVMe driver IO timeout from 30 to 300 seconds.
	Keywords: NVMe; recovery; kernel driver
	Discovered in version: 4.3.0
-	Description: NVMeTCP XLIO is not supported when running 64K page size kernels on the DPU Arm cores (as is the case for CentOS 8.x, Rocky 8.x, or openEuler 20.x).
	Workaround: N/A
	Keywords: 64K page size; NVMeTCP XLIO
	Discovered in version: 4.1.0
3264154	Description: NVMeTCP XLIO is not supported when running 64K page size kernels on the DPU Arm cores (such is the case with CentOS 8.x, Rocky 8.x, or openEuler 20.x).
	Workaround: N/A
	Keywords: Page size; NVMeTCP XLIO
	Discovered in version: 4.1.0
-	Description: NVMe over RDMA full offload is not supported.
	Workaround: N/A
	Keywords: NVMe over RDMA; support
	Discovered in version: 4.0.0

8.12.2.2.3.2 OS or Vendor Issues

Ref #	Issue Details
4583923	Description: Windows OS virtio-blk driver sends corrupted commands incompatible with virtio-blk specification conventions, making it impossible for the device to provide any reply (even with error status).
	Workaround: N/A
	Keywords: Windows, Windows OS, virtio-blk
	Discovered in version: 4.9.0
4408109	Description: When using a Windows OS on the host side, the controller must be created with a **`--num_queues`** value greater than or equal to the number of CPU cores on the Windows server. Failing to do so may result in undefined behavior, and the disk might fail to initialize properly.
	Workaround: N/A
	Keywords: Windows, Windows OS
	Discovered in version: 4.8.0
4573304	A call trace with the message “Bad FPU state detected” may appear in the host dmesg during consecutive host reboots when using hotplug functions. This is a known Linux kernel issue and has been fixed starting from Linux v6.1 and later.
	Workaround: disable AMX/XSAVE/XRSTOR feature to CPU, Hotplug
	Keywords: Linux Kernal, Call Trace, reboot, FPU
	Discovered in Version: 4.8.0
4540848	Description: Windows NVMe driver ignores MQEs value set on the PCI BAR, and forcefully trying to open queues with the size of 256.
	Workaround: Set `NVME_EMULATION_MAX_QUEUE_DEPTH` in mlxconfig to be >= 8 (default is 12), to support MQES >= 256 queue depth.
	Keywords: NVMe, Windows
	Discovered in version: 4.8.0
-	Description: Windows OS assumes NVMe devices support at least 2 IO CQs (CQ ID 2 exists), even when the controller declares it only supports 1 IO queue.
	Workaround: Open NVMe controller with **`–num_queues`** value >= 2
	Keywords: Windows, NVMe
	Discovered in version: 4.7.0
4418372	Description: On Windows OS, hot-unplugging a virtio-blk PCIe function can cause unexpected behavior, and a host reboot might be necessary to recover. This is because the Windows OS does not support online PCIe rescan, unlike Linux.
	Workaround: Before unplugging a PCIe function, disable its storage controller in **`Device Manager`**
	Keywords: Windows, virtio-blk, hotplug
	Discovered in version: 4.7.0

Ref #	Issue Details
4206444	Description: The Linux kernel driver does not restrict `seg_max` based on the maximum `queue_size` reported by the device. This can lead to a `WARN_ON_ONCE()` trigger in the kernel, resulting in driver misbehavior and potential system hangs.
	Workaround: Ensure that all virtio-blk controllers are configured with a `seg_max` value smaller than `(queue_size-2)`.
	Keywords: Virtio-blk; kernel driver; `seg_max`; `queue_size`
	Reported in version: 4.6.0
-	Description: When using hotplugged PCIe devices, after all devices are plugged, the host must be rebooted for Windows to detect all devices (some Windows versions may perform reboot automatically). This is requires as Windows OS does not support online PCIe rescan (as in Linux).
	Workaround: N/A
	Keywords: Hotplug, Windows
	Reported in version: 4.5.0
3748674	Description: On most modern Linux distributions, unplugging a PCIe function from the host while there are inflight I/Os can cause the virtio-blk driver to hang. To clarify, inflights I/Os might also be implicitly generated by other actions (e.g. driver load/unload).
	Workaround: N/A
	Keywords: Hotplug, Linux
	Reported in version: 4.5.0
-	Description: Some old Windows OS NVMe drivers have buggy usage of SGL support.
	Workaround: Disable SGL support when using Windows OS by setting the `--quirks` bit 4 to 1 in `snap_rpc.py nvme_controller_create` RPC.
	Keywords: Windows; NVMe
	Reported in version: 4.4.0
2879262	Description: When the virtio-blk kernel driver cannot find enough MSI-X vectors to satisfy all its opened virtqueues, it failovers to assign a single MSI-X vector to all virtqueues which negatively impacts performance. In addition, when a large number (e.g., 64) of virtqueues are associated with a single MSI-X, the kernel may enter a soft-lockup (kernel bug) and the IO will hang.
	Workaround: Always keep `num_queues` < `num_msix`.
	Keywords: Virtio-blk; kernel driver; MSI-X; soft-lockup
	Reported in version: 4.3.0
-	Description: If PCIe devices are inserted before the hot-plug driver is loaded on the host, the hot-plug driver in kernel versions less than 4.19 does not enable the slot, even if the slot is occupied (i.e., presence detected in the slot status register). This means that only the presence state of the slot is updated by the firmware, but the PCIe slot is not enabled by the kernel after the host boots up. As a result, the PCIe device will not be visible when using `lspci` on the host side, and the bus, device, and function (BDF) identifier will show as 0 on the controller.
	Workaround: Add `pciehp.pciehp_force=1` to the boot command line on host.
	Keywords: Virtio-blk; kernel driver; hot-plug

Ref #	Issue Details
	Reported in version: 4.2.1
-	Description: RedHat/Centos 7.x does not handle "online" (post driver probe) namespace additions or removals correctly.
	Workaround: Use **--quirks=0x2** option in **snap_rpc.py nvme_controller_create**.
	Keywords: NVMe; CentOS; RedHat; kernel
	Reported in version: 4.1.0
-	Description: Some Windows drivers have experimental support for "online" (post driver probe) namespace additions/removal, although such support is not communicated with the device.
	Workaround: Use **--quirks=0x1** option in **snap_rpc.py nvme_controller_create**.
	Keywords: NVMe; Windows
	Reported in version: 4.1.0
-	Description: VMWare ESXi supports "online" (post driver probe) namespace additions/ removal, only if “Namespace Management” is supported by controller.
	Workaround: Use **--quirks=0x8** option in **snap_rpc.py nvme_controller_create**.
	Keywords: NVMe, ESXi
	Reported in version: 4.1.0
-	Description: Ubuntu 22.04 does not support 500 VFs.
	Workaround: N/A
	Keywords: Virtio-blk; kernel driver; Ubuntu 22.04
	Reported in version: 4.1.0
-	Description: Virtio-blk Linux kernel driver does not handle PCIe FLR events. This issue was fixed in kernel 6.14 and above.
	Workaround: N/A
	Keywords: Virtio-blk; kernel driver
	Reported in version: 4.0.0
	Description: SPDK NVMe-oF/RDMA initiator fails to connect to kernel NVMe-oF/RDMA remote target.
	Workaround: Use setting **spdk_rpc.py bdev_nvme_set_options --io-queue-requests=128** on SPDK configuration
	Keywords: SPDK, NVMe-oF, RDMA, kernel
	Reported in version: 4.3.1
-	Description: Windows OS virtio-blk driver expects at least 64K data to be available for a single IO request
	Workaround: Use **seg_max** and **size_max** parameters configuration to match requirements (**seg_max** * **size_max** > 64K).
	Keywords: Windows, virtio-blk
	Reported in version: 4.3.1
-	Description: Some older Windows OS versions have malfunctioning inbox virtio-blk driver, expects a 3-party virtio-blk driver to be pre-installed to operate properly.

Ref #	Issue Details
	Workaround: Use a verified 3rd-party driver from Fedora.
	Keywords: Windows, virtio-blk
	Reported in version: 4.3.1
3679373	Description: Virtio-blk spdk driver (vfio-pci based) does not handle PCIe FLR events.
	Workaround: N/A
	Keywords: Virtio-blk; SPDK driver
	Reported in version: 4.3.0
-	Description: A new virtio-blk Linux kernel driver (starting kernel 4.18) does not support hot-unplug during traffic. Since the kernel may self-generate spontaneous IOs, on rare occasions, an issue may arise even when there is no traffic.
	Workaround: N/A
	Keywords: Virtio-blk; kernel driver
	Reported in version: 4.0.0
-	Description: When using SRIOV with VFs sharing the same driver as their PF (**sriov_driver_autoprobe=1**), unprobing the driver may take a long time, and some admin commands might timeout
	Workaround: N/A
	Keywords: SRIOV; driver.
	Discovered in version: 3.8.0-8

8.12.2.3 SNAP-4 Service Deployment

This section describes how to deploy SNAP as a container.

> ⚠ SNAP does not come pre-installed with the BFB.

8.12.2.3.1 1. Installing Full DOCA Image on DPU

To install NVIDIA® BlueField®-3 BFB:

```
[host] sudo bfb-install --rshim <rshimN> --bfb <image_path.bfb>
```

For more information, please refer to section "Installing Full DOCA Image on DPU" in the *DOCA Installation Guide for Linux*.

8.12.2.3.2 2. Firmware Installation

```
[dpu] sudo /opt/mellanox/mlnx-fw-updater/mlnx_fw_updater.pl --force-fw-update
```

For more information, please refer to section "Upgrading Firmware" in the *DOCA Installation Guide for Linux*.

8.12.2.3.3 3. Firmware Configuration

> ⚠ FW configuration may expose new emulated PCI functions, which can be later used by the host's OS. As such, user must make sure all exposed PCI functions (static/hotplug PFs, VFs) are backed by a supporting SNAP SW configuration, otherwise these functions will continue malfunctioning and host behavior will be undefined.

1. Clear the firmware config before implementing the required configuration:

```
[dpu] mst start
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 reset
```

2. Review the firmware configuration:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 query
```

Output example:

```
mlxconfig -d /dev/mst/mt41692_pciconf0 -e query | grep NVME
Configurations:                                 Default         Current         Next Boot
*        NVME_EMULATION_ENABLE                  False(0)        True(1)         True(1)
*        NVME_EMULATION_NUM_VF                  0               125             125
*        NVME_EMULATION_NUM_PF                  1               2               2
         NVME_EMULATION_VENDOR_ID               5555            5555            5555
         NVME_EMULATION_DEVICE_ID               24577           24577           24577
         NVME_EMULATION_CLASS_CODE              67586           67586           67586
         NVME_EMULATION_REVISION_ID             0               0               0
         NVME_EMULATION_SUBSYSTEM_VENDOR_ID     0               0               0
*       NVME_EMULATION_NUM_MSIX                 0               2               2
*       NVME_EMULATION_NUM_VF_MSIX              0               2               2
*       NVME_EMULATION_MAX_QUEUE_DEPTH          0               12              12
```

Where the output provides 5 columns:

- Non-default configuration marker (*)
- Firmware configuration name
- Default firmware value
- Current firmware value
- Firmware value after reboot – shows a configuration update which is pending system reboot

3. To enable storage emulation options, the first DPU must be configured to operate in internal CPU model:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s INTERNAL_CPU_MODEL=1 PF_BAR2_ENABLE=0
```

> ⚠ **PF_BAR2_ENABLE** is a deprecated option and must be explicitly disabled.

4. To enable the firmware config with virtio-blk emulation PF:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s VIRTIO_BLK_EMULATION_ENABLE=1 VIRTIO_BLK_EMULATION_NUM_PF=1
```

5. To enable the firmware config with NVMe emulation PF:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s NVME_EMULATION_ENABLE=1 NVME_EMULATION_NUM_PF=1
```

6. To enable the firmware configuration for any system (host or DPU) with a kernel page size of 64k (which can be verified using `getconf PAGESIZE`):

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s BAR_PAGE_ALIGNMENT=2
```

For a complete list of the SNAP firmware configuration options, refer to appendix "DPU Firmware Configuration".

Power cycle is required to apply firmware configuration changes.

8.12.2.3.3.1 3.1. RDMA/RoCE Firmware Configuration

RoCE communication is blocked for BlueField OS's default interfaces (named ECPFs, typically `mlx5_0` and `mlx5_1`). If RoCE traffic is required, additional network functions must be added, scalable functions (or SFs), which do support RoCE transport.

To enable RDMA/RoCE:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s PER_PF_NUM_SF=1
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s PF_SF_BAR_SIZE=8 PF_TOTAL_SF=2
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0.1 s PF_SF_BAR_SIZE=8 PF_TOTAL_SF=2
```

When using an OS with a 64KB page size, set `PF_SF_BAR_SIZE=10` instead of the default value of `8`.

This is not required when working over TCP or RDMA over InfiniBand.

8.12.2.3.3.2 3.2. SR-IOV Firmware Configuration

SNAP supports up to 512 total VFs on NVMe and up to 2000 total VFs on virtio-blk. The VFs may be spread between up to 4 virtio-blk PFs or 2 NVMe PFs.

The following examples are for reference. For complete details on parameter ranges, refer to appendix "DPU Firmware Configuration".

- Common example:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s SRIOV_EN=1 PER_PF_NUM_SF=1 LINK_TYPE_P1=2 LINK_TYPE_P2=2
PF_TOTAL_SF=1 PF_SF_BAR_SIZE=8
```

- Virtio-blk 250 VFs example (2 queue per VF):

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s VIRTIO_BLK_EMULATION_ENABLE=1
VIRTIO_BLK_EMULATION_NUM_VF=125 VIRTIO_BLK_EMULATION_NUM_PF=2 VIRTIO_BLK_EMULATION_NUM_MSIX=2
VIRTIO_BLK_EMULATION_NUM_VF_MSIX=2
```

- Virtio-blk 500 VFs example (2 queue per VF):

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s VIRTIO_BLK_EMULATION_ENABLE=1
VIRTIO_BLK_EMULATION_NUM_VF=250 VIRTIO_BLK_EMULATION_NUM_PF=2 VIRTIO_BLK_EMULATION_NUM_MSIX=2
VIRTIO_NET_EMULATION_ENABLE=0 NUM_OF_VFS=0 PCI_SWITCH_EMULATION_ENABLE=0 VIRTIO_BLK_EMULATION_NUM_VF_MSIX=2
```

- NVMe 512 VFs example (2 IO-queue per VF):

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s NVME_EMULATION_ENABLE=1 NVME_EMULATION_NUM_VF=256
NVME_EMULATION_NUM_PF=2 NVME_EMULATION_NUM_MSIX=2 NVME_EMULATION_NUM_VF_MSIX=2
NVME_EMULATION_MAX_QUEUE_DEPTH=12
```

8.12.2.3.3.3 3.3. Hot-plug Firmware Configuration

Once enabling PCIe switch emulation, BlueField can support up to 31 hotplug NVMe/Virtio-blk functions. "**PCI_SWITCH_EMULATION_NUM_PORT**-1" hot-plugged PCIe functions. These slots are shared among all DPU users and applications and may hold hot-plugged devices of type NVMe, virtio-blk, virtio-fs, or others (e.g., virtio-net).

To enable PCIe switch emulation and determine the number of hot-plugged ports to be used:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s PCI_SWITCH_EMULATION_ENABLE=1 PCI_SWITCH_EMULATION_NUM_PORT=32
```

PCI_SWITCH_EMULATION_NUM_PORT equals 1 + the number of hot-plugged PCIe functions.

For additional information regarding hot plugging a device, refer to section "Hot-pluggable PCIe Functions Management".

⚠ Hotplug is not guaranteed to work on AMD machines.

⚠ Enabling **PCI_SWITCH_EMULATION_ENABLE** could potentially impact SR-IOV capabilities on Intel and AMD machines.

⚠ Currently, hotplug PFs do not support SR-IOV.

8.12.2.3.3.4 3.4. UEFI Firmware Configuration

To use the storage emulation as a boot device, it is recommended to use the DPU's embedded UEFI expansion ROM drivers to be used by the UEFI instead of the original vendor's BIOS ones.

To enable UEFI drivers:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s EXP_ROM_VIRTIO_BLK_UEFI_x86_ENABLE=1 EXP_ROM_NVME_UEFI_x86_ENABLE=1
```

8.12.2.3.3.5 3.5. Socket Direct Firmware Configuration

BlueField-4 connects to the host over two PCI links. SNAP supports Socket Direct through a single emulation manager that serves both links.

To enable socket direct with NVMe PFs on both PCI Links:

```
[dpu] mlxconfig -d /dev/mst/mt41695_pciconf0 s PCI_BUS00_WIDTH=4 PCI_BUS04_WIDTH=4 PCI_BUS00_SPEED=5
PCI_BUS04_SPEED=5
<POWER CYCLE>
[dpu] mlxconfig -d /dev/mst/mt41695_pciconf0 s NVME_EMU_MNG_ENABLE=1 NVME_EMU_MNG_NUM_PF=1
[dpu] mlxconfig -d /dev/mst/mt41695_pciconf0.1 s NVME_EMU_MNG_ENABLE=1 NVME_EMU_MNG_NUM_PF=1
```

Once Socket Direct is enabled all the emulation related FW parameters are changed from XXX_EMULATION_XXX to XXX_EMU_MNG_XXX and can be configured for device /dev/mst/mt41695_pciconf0 (the first host link) and /dev/mst/mt41695_pciconf0.1 (the second host link)

To fully utilize BlueField-4 performance capabilities another DMA SF must be configured.

To configure a DMA SF first configure:

```
[dpu] mlxconfig -d /dev/mst/mt41695_pciconf1 s PF_SF_BAR_SIZE=14 PF_BAR2_ENABLE=0 PER_PF_NUM_SF=1 PF_TOTAL_SF=2
```

After a power cycle, BlueField-4 will automatically configure the DMA SF and SNAP will be use to maximize performance.

⚠ Socket Direct is currently only supported for BlueField-4

8.12.2.3.4 4. DPU Configurations

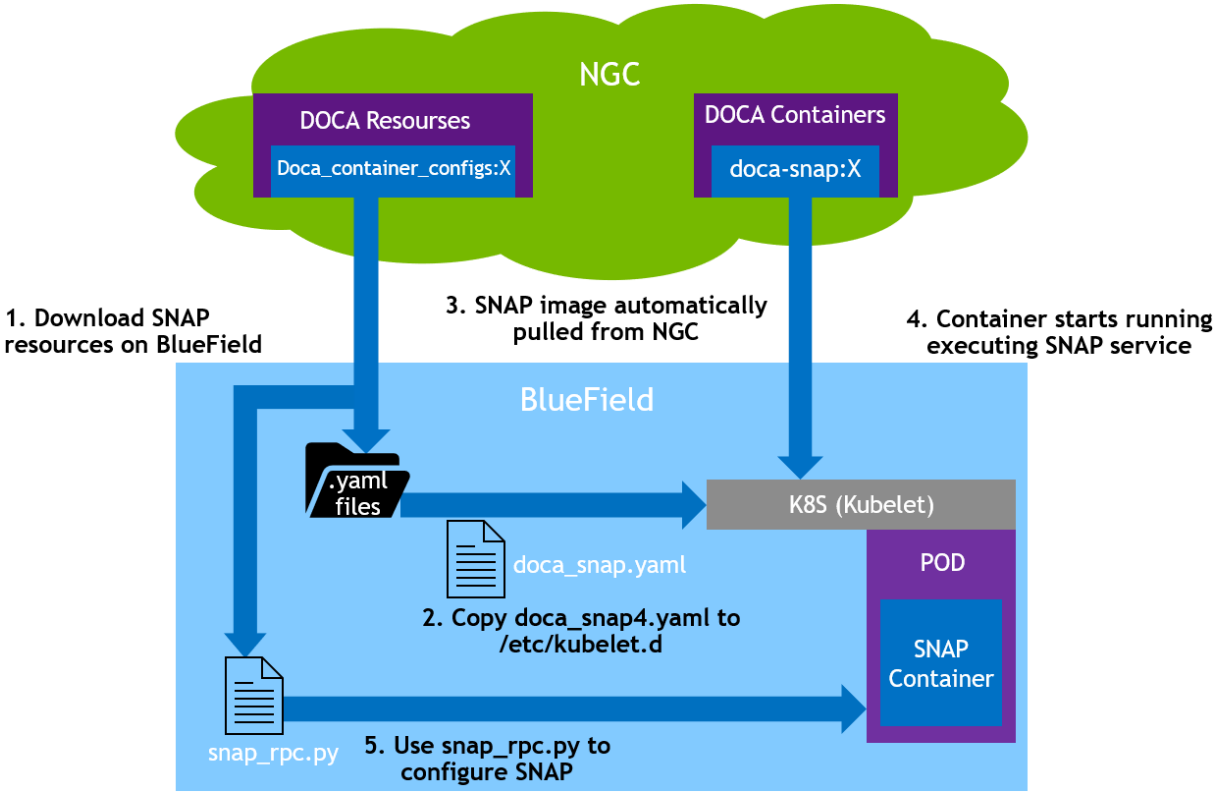
8.12.2.3.4.1 4.1. SNAP Container Deployment

SNAP container is available on the DOCA SNAP NVIDIA NGC catalog page.

SNAP container deployment on top of the BlueField DPU requires the following sequence:

1. Setup preparation and SNAP resource download for container deployment. See section "SNAP-4 Service Deployment#Preparation Steps" for details.
2. Adjust the **doca_snap.yaml** for advanced configuration if needed according to section "SNAP-4 Service Deployment#Adjusting YAML Configuration".
3. Deploy the container. The image is automatically pulled from NGC. See section "SNAP-4 Service Deployment#Spawning SNAP Container" for details.

The following is an example of the SNAP container setup.Edit



8.12.2.3.4.2 4.2. Preparation Steps

4.2.2. Step 1: Allocate Hugepages

Allocate **4GiB** hugepages for the SNAP container according to the DPU OS's **Hugepagesize** value:

1. Query the **Hugepagesize** value:

   ```
   [dpu] grep Hugepagesize /proc/meminfo
   ```

   For Ubuntu22 and Ubuntu24, the value should be 2048KB. For Ubuntu24 with 64k page size, the value should be 524288KB.

2. Use the **doca-hugepages** to configure the requested hugepages:

   - For OS with 2048KB hugepage:

     ```
     [dpu] doca-hugepages config --app snap --size 2048 --num 2048
     ```

   - For OS with 524288KB hugepage:

     ```
     [dpu] doca-hugepages config --app snap --size 524288 --num 8
     ```

     ⚠ For setups with a large size of hugepages (> 512MB), it is recommended to allocate slightly more than the requested amount (e.g., 5 GB instead of 4 GB), to ensure that SNAP has access to the required memory.

3. Reload the hugepages configuration for all applications based on the current database settings:

```
[dpu] doca-hugepages reload
```

If live upgrade is utilized in this deployment, it is necessary to allocate twice the amount of resources listed above for the upgraded container.

If other applications are running concurrently within the setup and are consuming hugepages, make sure to allocate additional hugepages beyond the amount described in this section for those applications.

When deploying SNAP with a high scale of connections (i.e., disks 500 or more), the default allocation of hugepages (4GiB) becomes insufficient. This shortage of hugepages can be identified through error messages in the SNAP and SPDK layers. These error messages typically indicate failures in creating or modifying QPs or other objects.

4.2.3. Step 2: Create nvda_snap Folder

The folder **/etc/nvda_snap** is used by the container for automatic configuration after deployment.

8.12.2.3.5 5. Downloading YAML Configuration

The **.yaml** file configuration for the SNAP container is **doca_snap.yaml**. The download command of the **.yaml** file can be found on the DOCA SNAP NGC page.

Internet connectivity is necessary for downloading SNAP resources. To deploy the container on DPUs without Internet connectivity, refer to appendix "Deploying Container on Setups Without Internet Connectivity".

8.12.2.3.5.1 6.1. Adjusting YAML Configuration

The **.yaml** file can easily be edited for advanced configuration.

- The SNAP **.yaml** file is configured by default to support Ubuntu setups (i.e., **Hugepagesize** = 2048 kB) by using hugepages-2Mi.
  To support other setups, edit the hugepages section according to the DPU OS's relevant **Hugepagesize** value. For example, to support CentOS 8.x configure **Hugepagesize** to 512MB:

```
limits:
   hugepages-512Mi: "<number-of-hugepages>Gi"
```

> ⚠ When deploying SNAP with a large number of controllers (500 or more), the default allocation of hugepages (4GB) may become insufficient. This shortage of hugepages can be identified through error messages, typically indicate failures in creating or modifying QPs or other objects. In these cases, more hugepages needed.

- The following example edits the **.yaml** file to request 16 CPU cores for the SNAP container and 4Gi memory and 4Gi hugepages:

```
    resources:
      requests:
        memory: "2Gi"
        hugepages-2Mi: "4Gi"
        cpu: "8"
      limits:
        memory: "4Gi"
        hugepages-2Mi: "4Gi"
        cpu: "16"
env:
  - name: APP_ARGS
    value: "-m 0xffff"
```

> ⚠ If all BlueField cores are requested, the user must verify no other containers are in conflict over the CPU resources.

> ⚠ When running the Virtio-fs service with a large number of cores, it is necessary to increase the number of IO buffers in SPDK. For example, to run with 16 cores, the size of the large IO buffer pool must be set to at least 4095. This can be configured by adding the RPC command **iobuf_set_options --large-pool-count 4095** to **spdk_rpc_init.conf** under **/etc/nvda_snap**. Depending on the scale and SPDK subsystems in use other SPDK configuration parameters may need to be adjusted. Refer to SPDK documentation for more details.

- To automatically configure SNAP container upon deployment, edit the files below according to the use case. During bring-up, SNAP will forward the content of these files into the appropriate RPC script, whether SPDK RPCs or SNAP RPCs. Ensure that the required RPCs for your use case are included.
    a. Add **spdk_rpc_init.conf** file under **/etc/nvda_snap/**. The file includes the required SPDK RPCs. File example:

    ```
    bdev_malloc_create 64 512
    ```

    b. Add **snap_rpc_init.conf** file under **/etc/nvda_snap/**. The file includes the required SPDK RPCs.
    Virtio-blk file example:

    ```
    virtio_blk_controller_create --pf_id 0 --bdev Malloc0
    ```

    NVMe file example:

    ```
    nvme_subsystem_create --nqn nqn.2022-10.io.nvda.nvme:0
    nvme_namespace_create -b Malloc0 -n 1 --nqn nqn.2022-10.io.nvda.nvme:0 --uuid 16dab065-
    ddc9-8a7a-108e-9a489254a839
    nvme_controller_create --nqn nqn.2022-10.io.nvda.nvme:0 --ctrl NVMeCtrl1 --pf_id 0 --suspended
    nvme_controller_attach_ns -c NVMeCtrl1 -n 1
    ```

```
nvme_controller_resume -c NVMeCtrl1
```

c. Edit the **.yaml** file accordingly (uncomment):

```
env:
  - name: SPDK_RPC_INIT_CONF
    value: "/etc/nvda_snap/spdk_rpc_init.conf"
  - name: SNAP_RPC_INIT_CONF
    value: "/etc/nvda_snap/snap_rpc_init.conf"
```

⚠ It is user responsibility to make sure SNAP configuration matches firmware configuration. That is, an emulated controller must be opened on all existing (static/hotplug) emulated PCIe functions (either through automatic or manual configuration). A PCIe function without a supporting controller is considered malfunctioned, and host behavior with it is anomalous.

⚠ Using **--wait-for-rpc** as one of the **APP_ARGS** and setting **SNAP_RPC_INIT_CONF** requires exactly one of **SPDK_RPC_INIT_CONF_JSON** or **SPDK_RPC_INIT_CONF** to be set which must contain a call toŽ **framework_start_init**

8.12.2.3.5.2 6.2. Spawning SNAP Container

Run the Kubernetes tool:

```
[dpu] systemctl restart containerd
[dpu] systemctl restart kubelet
[dpu] systemctl enable kubelet
[dpu] systemctl enable containerd
```

Copy the updated **doca_snap.yaml** file to the **/etc/kubelet.d** directory.

Kubelet automatically pulls the container image from NGC described in the YAML file and spawns a pod executing the container.

```
cp doca_snap.yaml /etc/kubelet.d/
```

The SNAP service starts initialization immediately, which may take a few seconds. To verify SNAP is running:

- Look for the message "SNAP Service running successfully" in the log
- Send **spdk_rpc.py spdk_get_version** to confirm whether SNAP is operational or still initializing

8.12.2.3.5.3 6.3. Debug and Log

View currently active pods, and their IDs (it might take up to 20 seconds for the pod to start):

```
crictl pods
```

Example output:

```
POD ID              CREATED               STATE         NAME
0379ac2c4f34c       About a minute ago    Ready         snap
```

View currently active containers, and their IDs:

```
crictl ps
```

View existing containers and their ID:

```
crictl ps -a
```

Examine the logs of a given container (SNAP logs):

```
crictl logs <container_id>
```

Examine the kubelet logs if something does not work as expected:

```
journalctl -u kubelet
```

The container log file is saved automatically by Kubelet under **/var/log/containers**.

The container log file is automatically saved by Kubelet to **/var/log/containers/**, using the filename format: **<pod_name>_default_snap-<container_id>.log**.

Refer to section "RPC Log History" for more logging information.

8.12.2.3.5.4 6.4. Logging Verbosity

To persist a custom log level across container restarts—or ensure it is applied during startup—add the relevant configuration command to the **snap_rpc_init.conf** file located at **/etc/nvda_snap/**.

Log level can also be modified at runtime using the **snap_log_level_set** RPC. For more details, refer to section "Log Management".

8.12.2.3.5.5 6.5. SNAP Logs (Source Package)

By default, the source package version of SNAP does not save logs automatically. To enable logging, follow the instructions in section "SNAP-4 Service Deployment#Run SNAP Service". For additional debugging information, refer to section "SNAP-4 Service Deployment#Build with Debug Prints Enabled".

To redirect SNAP output to a file, use the following command:

```
/opt/nvidia/nvda_snap/bin/snap_service > snap.log 2>&1
```

8.12.2.3.5.6 6.6. Exporting SNAP Logs (Container Only)

SNAP is integrated with SOS—a framework for consistent and structured log collection.
To generate a log package:

1. Clone the SOS report tool: https://github.com/NVIDIA/doca-sosreport.

2. Follow the installation instructions in the repository.
3. Run the following command:

```
sos report --only snap_service,container_log
```

This creates a comprehensive log package. You may include additional plugins depending on the nature of the issue. For more details, refer to the "Collecting DOCA Logs for NVIDIA Inspection" page.

8.12.2.3.5.7 6.7. Stop, Start, Restart SNAP Container

SNAP binaries are deployed within a Docker container as SNAP service, which is managed as a supervisorctl service. Supervisorctl provides a layer of control and configuration for various deployment options.

- In the event of a SNAP crash or restart, supervisorctl detects the action and waits for the exited process to release its resources. It then deploys a new SNAP process within the same container, which initiates a recovery flow to replace the terminated process.
- In the event of a container crash or restart, kubeletclt detects the action and waits for the exited container to release its resources. It then deploys a new container with a new SNAP process, which initiates a recovery flow to replace the terminated process.

> **General Kublet Comment**
>
> After containers crash or exit, the kubelet restarts them with an exponential back-off delay (10s, 20s, 40s, etc.) which is capped at five minutes. Once a container has run for 10 minutes without an issue, the kubelet resets the restart back-off timer for that container. Restarting the SNAP service without restarting the container helps avoid the occurrence of back-off delays.

6.7.1. Different SNAP Termination Options

6.7.1.1. Container Termination

- To kill the container, remove the **.yaml** file form **/etc/kubelet.d/**. To start the container, **cp** the **.yaml** file back to the same path:

```
cp doca_snap.yaml /etc/kubelet.d/
```

- To restart the container (with sig-term) using **crictl**, use the **-t** (timeout) option:

```
crictl stop -t 10 <container-id>
```

6.7.1.2. SNAP Process Termination

To restart the SNAP service without restarting the entire container, user can either use the **supervictl** tool to restart the SNAP service or terminate the SNAP service process on the DPU. Different signals correspond to different termination behaviors. For example:

- Restart sends SIGTERM

```
crictl exec -it $(crictl ps -s running -q --name snap) supervisorctl restart snap
```

- Pkill sends SIGKILL

```
pkill -9 -f snap
```

> ⚠ SNAP service termination may take time as it releases all allocated resources. The duration depends on the scale of the use case and any other applications sharing resources with SNAP.

6.7.1.3. SNAP Process Fast Restart

The duration can be improved by configuring **supervisorctl** to give the exited SNAP process a shorter or zero termination time when using **supervisorctl restart snap**.

This causes the new process to start up while the old process' resources are still being freed by the kernel.

The user must ensure that the hugepage allocation is sufficient to accommodate both processes running in parallel. To modify the time SNAP takes to exit, the user should use the relevant environment variable **SUPERVISOR_STOPWAITSECS**.

To restart the SNAP service without restarting the container, kill the SNAP service process on the DPU. Different signals can be used for different termination options. For example:

```
pkill -9 -f snap
```

8.12.2.3.6 7. SNAP Source Package Deployment

8.12.2.3.6.1 7.1. System Preparation

Allocate **4GiB** hugepages for the SNAP container according to the DPU OS's **Hugepagesize** value:

1. Query the **Hugepagesize** value:

   ```
   [dpu] grep Hugepagesize /proc/meminfo
   ```

   For Ubuntu22 and Ubuntu24, the value should be 2048KB. For Ubuntu24 with 64k page size, the value should be 524288KB.
2. Use the **doca-hugepages** to configure the requested hugepages:
   - For OS with 2048KB hugepage:

     ```
     [dpu] doca-hugepages config --app snap --size 2048 --num 2048
     ```
   - For OS with 524288KB hugepage:

     ```
     [dpu] doca-hugepages config --app snap --size 524288 --num 8
     ```

For setups with a large size of hugepages (> 512MB), it is recommended to allocate slightly more than the requested amount (e.g., 5 GB instead of 4 GB), to ensure that SNAP has access to the required memory.

3. Reload the hugepages configuration for all applications based on the current database settings:

```
[dpu] doca-hugepages reload
```

If live upgrade is utilized in this deployment, it is necessary to allocate twice the amount of resources listed above for the upgraded container.

If other applications are running concurrently within the setup and are consuming hugepages, make sure to allocate additional hugepages beyond the amount described in this section for those applications.

When deploying SNAP with a high scale of connections (i.e., disks 500 or more), the default allocation of hugepages (4GiB) becomes insufficient. This shortage of hugepages can be identified through error messages in the SNAP and SPDK layers. These error messages typically indicate failures in creating or modifying QPs or other objects.

8.12.2.3.6.2 7.2. Installing SNAP Source Package

Download the SNAP source package provided by NVIDIA on BlueField (BF).

Install the package:

- For Ubuntu, run:

```
dpkg -i snap-sources_<version>_arm64.*
```

- For CentOS, run:

```
rpm -i snap-sources_<version>_arm64.*
```

8.12.2.3.6.3 7.3. Build, Compile, and Install Sources

To build SNAP with a custom/legacy SPDK, see section "SNAP-4 Service Deployment#Replace the BFB SPDK".

1. Move to the sources folder. Run:

```
cd /opt/nvidia/nvda_snap/src/
```

2. Build the sources using **-Denable-spdk-compat=true** to ensure compatibility with SPDK, especially when using the out-of-box SPDK from a BFB. Then run:

```
meson setup /tmp/build -Denable-spdk-compat=true
```

3. Compile the sources. Run:

```
meson compile -C /tmp/build
```

4. Install the sources. Run:

```
meson install -C /tmp/build
```

8.12.2.3.6.4 7.4. Configure SNAP Environment Variables

To config the environment variables of SNAP, run:

```
source /opt/nvidia/nvda_snap/src/scripts/set_environment_variables.sh
```

8.12.2.3.6.5 7.5. Run SNAP Service

```
/opt/nvidia/nvda_snap/bin/snap_service
```

8.12.2.3.6.6 7.6. Replace the BFB SPDK (Optional)

Start with installing SPDK.

> ⚠ For legacy SPDK versions (e.g., SPDK 19.04) see appendix "Install Legacy SPDK".

To build SNAP with a custom SPDK, instead of following the basic build steps, perform the following:

1. Move to the sources folder. Run:

```
cd /opt/nvidia/nvda_snap/src/
```

2. Build the sources with spdk-compat enabled and provide the path to the custom SPDK. Run:

```
meson setup /tmp/build -Denable-spdk-compat=true -Dsnap_spdk_prefix=</path/to/custom/spdk>
```

3. Compile the sources. Run:

```
meson compile -C /tmp/build
```

4. Install the sources. Run:

```
meson install -C /tmp/build
```

5. Configure SNAP env variables and run SNAP service as explained in section "SNAP-4 Service Deployment#Configure SNAP Environment Variables" and "SNAP-4 Service Deployment#Run SNAP Service".

8.12.2.3.6.7 7.7. Build With Dynamically-linked Dependencies (Optional)

Instead of the basic build steps, perform the following:

1. Move to the sources folder. Run:

```
cd /opt/nvidia/nvda_snap/src/
```

2. Build the sources with **-Dlibsnapemu=shared**. Run:

```
meson setup /tmp/build -Denable-spdk-compat=true -Dlibsnapemu=shared /tmp/build
```

3. Compile the sources. Run:

```
meson compile -C /tmp/build
```

4. Install the sources. Run:

```
meson install -C /tmp/build
```

5. Configure SNAP environment variables and run SNAP service as explained in section "SNAP-4 Service Deployment#Configure SNAP Environment Variables" and "SNAP-4 Service Deployment#Run SNAP Service".

8.12.2.3.6.8 7.8. Build with Debug Prints Enabled (Optional)

Instead of the basic build steps, perform the following:

1. Move to the sources folder. Run:

```
cd /opt/nvidia/nvda_snap/src/
```

2. Build the sources with **buildtype=debug**. Run:

```
meson --buildtype=debug /tmp/build
```

3. Compile the sources. Run:

```
meson compile -C /tmp/build
```

4. Install the sources. Run:

```
meson install -C /tmp/build
```

5. Configure SNAP env variables and run SNAP service as explained in section "SNAP-4 Service Deployment#Configure SNAP Environment Variables" and "SNAP-4 Service Deployment#Run SNAP Service".

8.12.2.3.6.9 7.9. Automate SNAP Configuration (Optional)

The script **run_snap.sh** automates SNAP deployment. Users must modify the following files to align with their setup. The configuration files are in the same format explained in the "adjust the

YAML section". If different directories are utilized by the user, edits must be made to **run_snap.sh** accordingly:

1. Edit SNAP env variables in:

```
/opt/nvidia/nvda_snap/bin/set_environment_variables.sh
```

2. Edit SPDK initialization RPCs calls:

```
/opt/nvidia/nvda_snap/bin/spdk_rpc_init.conf
```

3. Edit SNAP initialization RPCs calls:

```
/opt/nvidia/nvda_snap/bin/snap_rpc_init.conf
```

4. Run the script:

```
/opt/nvidia/nvda_snap/bin/run_snap.sh
```

8.12.2.3.7 8. SNAP-4 Deployment for BlueField-4

This section details the deployment of SNAP on BlueField-4 hardware. The BlueField-4 architecture introduces structural changes that impact resource allocation and supported offloading pathways. The following sections outline the primary architectural shifts and the required parameters for deploying SNAP on BlueField-4.

8.12.2.3.7.1 8.1. Main Architectural Changes

8.1.1. Expanded Compute and Resource Allocation

The BlueField-4 architecture features an expanded hardware resource pool, including 64 available Arm cores and higher memory bandwidth. To ensure proper resource isolation and predictable data path execution, you must explicitly bind compute resources to the SNAP container.

8.1.1.1. Configuring Core Allocation

To provision 16 of the 64 available cores exclusively for the SNAP process, define the exact resource **requests** and **limits** in your YAML file, alongside the **APP_ARGS** environment variable:

```
resources:
      requests:
        memory: "2Gi"
        hugepages-2Mi: "4Gi"
        cpu: "8"
      limits:
        memory: "4Gi"
        hugepages-2Mi: "4Gi"
        cpu: "16"

env:
  - name: APP_ARGS
    value: "-m 0xffff"
```

8.1.2. GGA Architecture Deprecation

By design, the GGA architecture is no longer supported on BlueField-4. SNAP deployments now rely entirely on the BlueField-4's standard hardware-offloading engines and direct core processing.

This architectural transition is handled automatically by the system. No user intervention or manual reconfiguration is required to enable this standard operating mode on BlueField-4.

8.1.3. I/O Buffer Configuration for High Core Counts

When running SNAP with a high number of CPU cores, the default SPDK I/O buffer pool configuration may be insufficient. In these scenarios, you must increase the **small_pool_count** and **large_pool_count** parameters to ensure adequate buffer availability and prevent allocation failures.

You can apply this configuration using one of the following methods:

8.1.3.1. Option 1: Runtime Configuration via RPC

Start SNAP with the **--wait-for-rpc** flag, and then invoke the **iobuf_set_options** RPC to dynamically update the buffer pool parameters.

8.1.3.2. Option 2: Static Configuration via SPDK Configuration File

Add the following configuration block to your SPDK JSON configuration file:

```
{
  "subsystems": [
    {
      "subsystem": "iobuf",
      "config": [
        {
          "method": "iobuf_set_options",
          "params": {
            "small_pool_count": 32767,
            "large_pool_count": 4095,
            "small_bufsize": 9728,
            "large_bufsize": 131072
          }
        }
      ]
    }
  ]
}
```

Applying one of these configurations ensures stable operation when scaling SNAP to a large number of cores.

8.12.2.3.7.2 8.2. Supported Features (SNAP-4)

This NVMe SNAP-4 release on BlueField-4 hardware supports the following baseline capabilities:

- Physical and Virtual Emulation Functions
- Scale and Performance
- Live Update
- Crash Recovery

Additional features and enhancements are scheduled for upcoming SNAP and firmware releases.

8.12.2.4 SNAP-4 Service Environment Variables

8.12.2.4.1 Supported Environment Variables

Name	Description	Default
SNAP_RDMA_ZCOPY_ENABLE	Enable/disable RDMA zero-copy transport type. For more info refer to section "Zero Copy (SNAP-direct)".	**1** (enabled)
NVME_BDEV_RESET_ENABLE	It is recommended that namespaces discovered from the same remote target are not shared by different PCIe emulations. If it is desirable to do that, users should set the variable **NVME_BDEV_RESET_ENABLE** to **0**. By doing so, the user must ensure that SPDK bdev always completes IOs (either with success or failure) in a reasonable time. Otherwise, the system may stall until all IOs return.	**1** (enabled)
VBLK_RECOVERY_SHM	Enable/disable virtio-blk recovery using shared memory files. This allows recovering without using **--force_in_order**.	**1** (enabled)
SNAP_EMULATION_MANAGER	The name of the RDMA device configured to have emulation management capabilities. If the variable is not defined (default), SNAP searches through all available devices to find the emulation manager (which may slow down initialization process). Unless configured otherwise, SNAP selects the first ECPF (i.e., "mlx5_0") as the emulation manager.	Unset
VIRTIO_EMU_PROVIDER	Configure the Virtio-blk data path Provider. Valid options: **dpu**; **dpa**.	**dpu**
IO_CORE_DISTRIBUTION_POLICY	Support for IO/core Multiplexer (MP) capabilities in DPA provider for NVMe. For more info refer to section "NVMe DPA multiplexer (MP) data path Provider" - options "weighted_rr", "none"	**none** (disabled)
SUPERVISOR_STOPWAITSECS	SNAP process termination time when using **supervisorctl restart snap**	**10** (sec)
SNAP_LOG_FILE	File path to redirect SNAP logs to at startup, will append/create.	Unset

8.12.2.4.2 YAML Configuration

To change the SNAP environment variables add the following to the **doca_snap.yaml** and continue from section "Adjusting YAML Configuration".

```
env:
      - name: VARIABLE_NAME
        value: "VALUE"
```

For example:

```
env:
       - name: SNAP_RDMA_ZCOPY_ENABLE
         value: "1"
```

8.12.2.4.3 Source Package Configuration

To change the SNAP environment variables:

1. Add/modify the configuration under **scripts/set_environment_variables.sh**.
2. Rerun:

   ```
   source scripts/set_environment_variables.sh
   ```

3. Rerun SNAP.

8.12.2.5 SNAP-4 Service RPC Commands

Remote procedure call (RPC) protocol is used to control the SNAP service. NVMe/virtio-blk SNAP, like other standard SPDK applications, supports JSON-based RPC protocol commands to control any resources and create, delete, query, or modify commands easily from CLI.

SNAP supports all standard SPDK RPC commands in addition to an extended SNAP-specific command set. SPDK standard commands are executed by the **spdk_rpc.py** tool while the SNAP-specific command set extension is executed by the **snap_rpc.py** tool.

Full **spdk_rpc.py** command set documentation can be found in the SPDK official documentation site.

Full **snap_rpc.py** extended commands are detailed further down in this chapter.

8.12.2.5.1 Using JSON-based RPC Protocol

The JSON-based RPC protocol can be used via the **snap_rpc.py** script that is installed inside the SNAP container and can be accessed using **crictl** tool.

> The SNAP container is CRI-compatible.

- To query the active container ID:

  ```
  crictl ps -s running -q --name snap
  ```

- To post RPCs to the container using **crictl**:

  ```
  crictl exec <container-id> snap_rpc.py <RPC-method>
  ```

  For example:

  ```
  crictl exec 0379ac2c4f34c snap_rpc.py emulation_function_list
  ```

  In addition, an alias can be used:

```
alias snap_rpc.py="crictl ps -s running -q --name snap | xargs -I{} crictl exec -i {} snap_rpc.py "
alias spdk_rpc.py="crictl ps -s running -q --name snap | xargs -I{} crictl exec -i {} spdk_rpc.py "
```

- To open a bash shell to the container that can be used to post RPCs:

```
crictl exec -it <container-id> bash
```

8.12.2.5.2 Log Management

8.12.2.5.2.1 snap_log_level_set

SNAP allows dynamically changing the log level of the logger backend using the **snap_log_level_set**. Any log under the requested level is shown.

Parameter	Mandatory?	Type	Description
level	Yes	Number	Log level • 0 – Critical • 1 – Error • 2 – Warning • 3 – Info • 4 – Debug • 5 – Trace

8.12.2.5.2.2 snap_log_marker

Writes a custom marker message into the SNAP log. This is particularly useful for embedding **BEGIN** and **END** anchors during automated or manual testing, which makes subsequent log analysis significantly easier to parse.

To mark the beginning of a specific test sequence, run:

```
snap_rpc.py snap_log_marker "BEGIN: test_000"
```

- **message** (String): A custom text string, enclosed in double quotes (i.e., **"message"**), to write to the log.

8.12.2.5.3 PCIe Function Management

Emulated PCIe functions are managed through IB devices called emulation managers. Emulation managers are ordinary IB devices with special privileges to control PCIe communication and device emulations towards the host OS.

SNAP queries an emulation manager that supports the requested set of capabilities.

The emulation manager holds a list of the emulated PCIe functions it controls. PCIe functions may be approached later in 3 ways:

- **vuid** – recommended as it is guaranteed to remain constant (see appendix "PCIe BDF to VUID Translation" for details)
- **vhca_id**

- Function index (i.e., **pf_id** or **vf_id**)

8.12.2.5.3.1 emulation_function_list

emulation_function_list lists all existing functions.

The following is an example response for the **emulation_function_list** command:


```
[
  {
    "hotplugged": true,
    "hotplug state": "POWER_ON",
    "emulation_type": "VBLK",
    "pf_index": 0,
    "pci_bdf": "87:00.0",
    "vhca_id": 5,
    "vuid": "MT2306XZ009TVBLKS1D0F0",
    "PCI link": 0,
    "ctrl_id": "VblkCtrl1",
    "num_vfs": 0,
    "vfs": []
  }
]
```

Use **-a** or **--all**, to show all inactive VF functions.

SNAP supports 2 types of PCIe functions:

- Static functions – PCIe functions configured at the firmware configuration stage (physical and virtual). Refer to appendix "DPU Firmware Configuration" for additional information.
- Hot-pluggable functions – PCIe functions configured dynamically at runtime. Users can add detachable functions. Refer to section "Hot-pluggable PCIe Functions Management" for additional information.

BlueField-4 supports Socket Direct, which connects the BlueField-4 to the host over two PCI links. The following is an example response If Socket Direct is enabled:


```
[
  {
    "hotplugged": false,
    "emulation_type": "nvme",
    "pf_index": 0,
    "pci_bdf": "83:00.2",
    "vhca_id": 2,
    "vuid": "MT2333X00000ECPF0PF2",
    "PCI link": 0,
    "ctrl_id": "NVMeCtrl0",
    "num_vfs": 0,
    "vfs": []
  },
  {
    "hotplugged": false,
    "emulation_type": "nvme",
    "pf_index": 1,
    "pci_bdf": "85:00.2",
    "vhca_id": 5,
    "vuid": "MT2333X00000ECPF1PF2",
    "PCI link": 1,
    "ctrl_id": "NVMeCtrl1",
    "num_vfs": 0,
    "vfs": []
  }
]
```


8.12.2.5.3.2 emulation_resource_list

This RPC returns information about the global resource pools available for emulated functions.

It can be used to verify resource availability before creating or modifying hotplugged Virtio-blk or NVMe devices.

The RPC reports the number of available MSI-X resources and queue resources. The values returned by this RPC are updated according to the resource allocation rules described below.

The following is an example response for the **emulation_resource_list** command:

```
[
  {
    "num_available_msix_resources": 9636,
    "num_available_queue_resources": 32728
  }
]
```

⚠ NVMe queues (both Admin and I/O) consume two queue resources each, whereas virtio-blk virtqueues (vq) consume a single queue resource.

8.12.2.5.3.3 MSI-X Resource Pool Size

The **num_available_msix_resources** field indicates the number of available MSI-X resources in the global pool. The MSI-X resource pool size changes according to the following operations:

Operation	MSI-X Pool Size Change
virtio_blk_function_create	We don't have a way to export this macro.
virtio_blk_controller_modify (changes **num_msix** from A to B)	We don't have a way to export this macro.
nvme_function_create	We don't have a way to export this macro.
nvme_controller_modify (changes **num_msix** from A to B)	We don't have a way to export this macro.

⚠ In the formulas above, **S** represents the total pool size.

8.12.2.5.3.4 Queue Resource Pool Size

The **num_available_queue_resources** field indicates the number of available queue resources in the global pool. The queue resource pool size changes according to the following operations:

Operation	Queue resource pool size change
`virtio_blk_function_create`	We don't have a way to export this macro.
`virtio_blk_controller_modify` (changes `num_queues` from `A` to `B`)	We don't have a way to export this macro.
`nvme_function_create`	We don't have a way to export this macro.
`nvme_controller_create --num_queues X`	We don't have a way to export this macro.
`nvme_controller_modify` (changes `num_queues` from `A` to `B`)	We don't have a way to export this macro.

⚠ In the formulas above, `S` represents the total pool size, and
We don't have a way to export this macro.
denotes rounding the value `x` up to the nearest multiple of 8.

8.12.2.5.4 Hot-pluggable PCIe Functions Management

Hotplug PCIe functions are configured dynamically at runtime using RPCs. Once a new PCIe function is hot plugged, it appears in the host's PCIe device list and remains persistent until explicitly unplugged or the system undergoes a cold reboot. Importantly, this persistence continues even if the SNAP process terminates. Therefore, it is advised not to include hotplug/hotunplug actions in automatic initialization scripts (e.g., `snap_rpc_init.conf`).

⚠ Hotplug PFs do not support SR-IOV.

8.12.2.5.4.1 Two-step PCIe Hotplug

The following commands are designed to support modern two-step PCIe hotplug approach.

For more info about hotplug approaches, see section "Hot-pluggable PCIe Functions".

Virtio-blk Two-step PCIe Hotplug

virtio_blk_function_create

Create a new virtio-blk emulation function.

Command parameters:

Parameter	Mandatory?	Type	Description
`manager`	No	String	Emulation manager to manage hotplug function (unused)

virtio_blk_function_destroy

Delete an existing virtio-blk emulation function.

Command parameters:

Parameter	Mandatory?	Type	Description
`vuid`	Yes	String	Identifier of the hotplugged function to delete

virtio_blk_controller_hotplug

Exposes (hot plugs) the emulation function to the host OS.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller to expose to the host OS
`wait_for_done`	No	Bool	Block RPC until host discovers and acknowledges the new hotplug state
`mode`	No	Number	Specifies the host awareness strategy for the controller hotplug operation. Values: • `1` (host-aware): The operation takes the host system state into account to ensure safe handling. This is the recommended mode. • `2` (host-unaware): The operation ignores the host system state. **Host-unaware mode is unsafe. If this mode is used while the host is in a `not_ready` state, the RPC will return successfully (without error), but the actual hotplug behavior on the host is undefined.** • `3` (host-aware + attention button): The same as host aware mode, but operation also uses PCIe Attention Button feature. **This mode causes 5-second delay of device's hotplug state change to `POWER_OFF`. It is suggested to use `emulation_functions_list` check the hotplug state have changed to `POWER_OFF` before do next step if `--wait_for_done` flag is not used.** If no value is specified, the system attempts to use host-aware mode if supported. Otherwise, it reverts to the internal system default.

virtio_blk_controller_hotunplug

Removes (hot unplugs) the emulation function from the host OS.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller to expose to the host OS
`wait_for_done`	No	Bool	Specifies whether the RPC should block execution until the host system discovers and acknowledges the new hotplug state. • Enabled (True): The RPC waits synchronously for the host to acknowledge the configuration change before returning • Disabled (False): The RPC returns immediately, leaving the verification of the host state to the user When `wait_for_done` is disabled, it is the user's responsibility to verify that the host has identified the new function state. This can be done by querying the `pci_hotplug_state` parameter in the output of the `emulation_function_list` RPC.
`mode`	No	Number	Specifies the host awareness strategy for the controller hotunplug operation. Values: • `1` (host-aware): The operation takes the host system state into account to ensure safe handling. This is the recommended mode. • `2` (host-unaware): The operation ignores the host system state. ⛔ Host-unaware mode is unsafe. If this mode is used while the host is in a `not_ready` state, the RPC will return successfully (without error), but the actual hotunplug behavior on the host is undefined. • `3` (host-aware + attention button): The same as host aware mode, but operation also uses PCIe Attention Button feature. ⚠ This mode will cause 5 seconds delay of device's hotplug state change to `POWER_OFF`. It is suggested to use `emulation_functions_list` check the hotplug state have changed to `POWER_OFF` before do next step if `--wait_for_done` flag is not used. If no value is specified, the system attempts to use host-aware mode if supported. Otherwise, it reverts to the internal system default.

Virtio-blk Two-step PCIe Hotplug/Unplug Example

```
# Bringup
spdk_rpc.py bdev_nvme_attach_controller -b nvme0 -t rdma -a 1.1.1.1 -f ipv4 -s 4420 -n
nqn.2022-10.io.nvda.nvme:swx-storage
snap_rpc.py virtio_blk_function_create
snap_rpc.py virtio_blk_controller_create --vuid MT2114X12200VBLKS1D0F0 --bdev nvme0n1
snap_rpc.py virtio_blk_controller_hotplug -c VblkCtrl1 --wait_for_done

# Cleanup
snap_rpc.py virtio_blk_controller_hotunplug -c VblkCtrl1 --wait_for_done
snap_rpc.py virtio_blk_controller_destroy -c VblkCtrl1
snap_rpc.py virtio_blk_function_destroy --vuid MT2114X12200VBLKS1D0F0
spdk_rpc.py bdev_nvme_detach_controller nvme0
```

NVMe Two-step PCIe Hotplug

nvme_function_create

Creates a new NVMe emulation function.

Command parameters:

Parameter	Mandatory?	Type	Description
`manager`	No	String	Emulation manager to manage hotplug function (unused)

nvme_function_destroy

Deletes an existing NVMe emulation function.

Command parameters:

Parameter	Mandatory?	Type	Description
`vuid`	Yes	String	Identifier of the hotplugged function to delete

nvme_controller_hotplug

Exposes (hot plugs) the emulation function to the host OS.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller to expose to the host OS
`wait_for_done`	No	Bool	Block RPC until host discovers and acknowledges the new hotplug state
`mode`	No	Number	Specifies the host awareness strategy for the controller hotplug operation. Values: • `1` (host-aware): The operation takes the host system state into account to ensure safe handling. This is the recommended mode. • `2` (host-unaware): The operation ignores the host system state. ⊗ Host-unaware mode is unsafe. If this mode is used while the host is in a `not_ready` state, the RPC will return successfully (without error), but the actual hotplug behavior on the host is undefined. If no value is specified, the system attempts to use host-aware mode if supported. Otherwise, it reverts to the internal system default.

nvme_controller_hotunplug

Removes (hot unplugs) the emulation function from the host OS.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller to expose to the host OS
`wait_for_done`	No	Bool	Specifies whether the RPC should block execution until the host system discovers and acknowledges the new hotplug state. • Enabled (True): The RPC waits synchronously for the host to acknowledge the configuration change before returning • Disabled (False): The RPC returns immediately, leaving the verification of the host state to the user When `wait_for_done` is disabled, it is the user's responsibility to verify that the host has identified the new function state. This can be done by querying the `pci_hotplug_state` parameter in the output of the `emulation_function_list` RPC.
`mode`	No	Number	Specifies the host awareness strategy for the controller hotunplug operation. Values: • `1` (host-aware): The operation takes the host system state into account to ensure safe handling. This is the recommended mode. • `2` (host-unaware): The operation ignores the host system state. Host-unaware mode is unsafe. If this mode is used while the host is in a `not_ready` state, the RPC will return successfully (without error), but the actual hotunplug behavior on the host is undefined. If no value is specified, the system attempts to use host-aware mode if supported. Otherwise, it reverts to the internal system default.

NVMe Two-step PCIe Hotplug/Unplug Example

```
# Bringup
spdk_rpc.py bdev_nvme_attach_controller -b nvme0 -t rdma -a 1.1.1.1 -f ipv4 -s 4420 -n
nqn.2022-10.io.nvda.nvme:swx-storage
snap_rpc.py nvme_subsystem_create --nqn nqn.2014-08.com.nvda:nvme:0
snap_rpc.py nvme_function_create
snap_rpc.py nvme_controller_create --nqn nqn.2014-08.com.nvda:nvme:0 --vuid MT2114X12200VBLKS1D0F0 --suspended
snap_rpc.py nvme_namespace_create -b nvme0n1 -n 1 --nqn nqn.2014-08.com.nvda:nvme:0
snap_rpc.py nvme_controller_attach_ns -c NVMeCtrl1 -n 1
snap_rpc.py nvme_controller_resume -c NVMeCtrl1
snap_rpc.py nvme_controller_hotplug -c NVMeCtrl1 --wait_for_done

# Cleanup
snap_rpc.py nvme_controller_hotunplug -c NVMeCtrl1 --wait_for_done
snap_rpc.py nvme_controller_detach_ns -c NVMeCtrl1 -n 1
snap_rpc.py nvme_namespace_destroy -n 1 --nqn nqn.2014-08.com.nvda:nvme:0
snap_rpc.py nvme_controller_destroy -c NVMeCtrl1
snap_rpc.py nvme_subsystem_destroy --nqn nqn.2014-08.com.nvda:nvme:0
snap_rpc.py nvme_function_destroy --vuid MT2114X12200VBLKS1D0F0
spdk_rpc.py bdev_nvme_detach_controller nvme0
```

8.12.2.5.4.2 (Deprecated) PCIe Hotplug Legacy API

(Deprecated) Virtio-blk PCIe Hotplug Legacy API

Support for the legacy API ends in January 31, 2026.

Hotplug Commands

The following commands hot plug a new virtio-blk PCIe function to the system using legacy one-step interface.

virtio_blk_emulation_device_attach

Attach virtio-blk emulation function.

Command parameters:

Parameter	Mandatory?	Type	Description
`id`	No	Number	Device ID
`vid`	No	Number	Vendor ID
`ssid`	No	Number	Subsystem device ID
`ssvid`	No	Number	Subsystem vendor ID
`revid`	No	Number	Revision ID
`class_code`	No	Number	Class code
`num_msix`	No	Number	MSI-X table size
`total_vf`	No	Number	Maximal number of VFs allowed
`bdev`	No	String	Block device to use as backend
`num_queues`	No	Number	Number of IO queues (default 1, range 1-62). ⚠ The actual number of queues is limited by the number of queues supported by the hardware. ✓ It is recommended that the number of MSIX be greater than the number of IO queues (1 is used for the config interrupt).
`queue_depth`	No	Number	Queue depth (default 256, range 1-256) ⚠ It is only possible to modify the queue depth if the driver is not loaded.
`transitional_device`	No	Boolean	Transitional device support. See section "Virtio-blk Transitional Device Support" for more details.
`dbg_bdev_type`	No	Boolean	N/A – not supported

Hot Unplug Commands

The following commands hot-unplug a PCIe function from the system in 2 steps:

	Command	Description
1	emulation_device_detach_prepare	Prepare emulation function to be detached
2	emulation_device_detach	Detach emulation function

emulation_device_detach_prepare

This is the first step for detaching an emulation device. It prepares the system to detach a hot plugged emulation function. In case of success, the host's hotplug device state changes and you may safely proceed to the **emulation_device_detach** command.

The controller attached to the emulation function must be created and active when executing this command.

Command parameters:

Parameter	Mandatory?	Type	Description
vhca_id	No	Number	vHCA ID of PCIe function
vuid	No	String	PCIe device VUID
ctrl	No	String	Controller ID

⚠ At least one identifier must be provided to describe the PCIe function to be detached.

emulation_device_detach

This is the second step which completes detaching of the hotplugged emulation function. If the detach preparation times out, you may perform a surprise unplug using **--force** with the command.

⚠ The driver must be unprobed, otherwise errors may occur.

Command parameters:

Parameter	Mandatory?	Type	Description
vhca_id	No	Number	vHCA ID of PCIe function
vuid	no	String	PCIe device VUID
force	No	Boolean	Detach with failed preparation

⚠ At least one identifier must be provided to describe the PCIe function to be detached.

Virtio-blk Hot Plug/Unplug Example

```
// Bringup
spdk_rpc.py bdev_nvme_attach_controller -b nvme0 -t rdma -a 1.1.1.1 -f ipv4 -s 4420 -n
nqn.2022-10.io.nvda.nvme:swx-storage
snap_rpc.py virtio_blk_emulation_device_attach
snap_rpc.py virtio_blk_controller_create --vuid MT2114X12200VBLKS1D0F0 --bdev nvme0n1

// Cleanup
snap_rpc.py emulation_device_detach_prepare --vuid MT2114X12200VBLKS1D0F0
snap_rpc.py virtio_blk_controller_destroy -c VblkCtrl1
snap_rpc.py emulation_device_detach --vuid MT2114X12200VBLKS1D0F0
spdk_rpc.py bdev_nvme_detach_controller nvme0
```

(Deprecated) NVMe Hotplug Legacy Commands

Support for the legacy API ends in January 31, 2027.

The following commands perform hotplug operations for a new NVMe PCIe function.

Once a PCIe function is attached, it appears immediately in the host's PCIe device list and remains there until explicitly detached or until a cold reboot occurs. It is the user's responsibility to create and activate a controller instance to manage the new function immediately after attachment. Leaving a hotplugged function unmanaged may cause anomalous behavior in the host OS driver.

nvme_emulation_device_attach

Attaches an NVMe emulation function.

Command parameters:

Parameter	Mandatory?	Type	Description
id	No	Number	Device ID
vid	No	Number	Vendor ID
ssid	No	Number	Subsystem device ID
ssvid	No	Number	Subsystem vendor ID
revid	No	Number	Revision ID
class_code	No	Number	Class code
num_msix	No	Number	MSI-X table size
total_vf	No	Number	Maximal number of VFs allowed

Parameter	Mandatory?	Type	Description
num_queues	No	Number	Number of IO queues (default 31, range 1-31). ⚠ The actual number of queues is limited by the number of queues supported by the hardware. ✓ It is recommended that the number of MSIX be greater than the number of IO queues (1 is used for the admin queue).
version	No	String	Specification version (currently only 1.4 is supported)

Hot Unplug Legacy Commands

The following commands hot-unplug a PCIe function from the system in 2 steps:

	Command	Description
1	emulation_device_detach_prepare	Prepare emulation function to be detached
2	emulation_device_detach	Detach emulation function

emulation_device_detach_prepare

Prepares the system to detach a hotplugged emulation function.

This is the first step in the detachment sequence. Upon success, the device enters a safe state for removal, allowing you to proceed with the **emulation_device_detach** command.

⚠ A controller must be active and attached to the emulation function before executing this command.

Command parameters:

Parameter	Mandatory?	Type	Description
vhca_id	No	Number	vHCA ID of PCIe function
vuid	No	String	PCIe device VUID
ctrl	No	String	Controller ID

⚠ At least one identifier must be provided to describe the PCIe function to be detached.

emulation_device_detach

Completes the detachment of a hotplugged emulation function.

If the preparation phase (**emulation_device_detach_prepare**) times out, use the **--force** option to perform a surprise unplug.

> ⚠ Ensure the driver has been properly unbound from the device before running this command, or errors may occur.

Command parameters:

Parameter	Mandatory?	Type	Description
vhca_id	No	Number	vHCA ID of PCIe function
vuid	no	String	PCIe device VUID
force	No	Boolean	Detach with failed preparation

> ⚠ At least one identifier must be provided to describe the PCIe function to be detached.

NVMe Hot Plug/Unplug Example

```
// Bringup
spdk_rpc.py bdev_nvme_attach_controller -b nvme0 -t rdma -a 1.1.1.1 -f ipv4 -s 4420 -n nqn.2022-10.io.nvda.nvme:0
snap_rpc.py nvme_subsystem_create --nqn nqn.2022-10.io.nvda.nvme:0
snap_rpc.py nvme_namespace_create -b nvme0n1 -n 1 --nqn nqn.2022-10.io.nvda.nvme:0 --uuid 263826ad-19a3-4feb-
bc25-4bc81ee7749e
snap_rpc.py nvme_emulation_device_attach
snap_rpc.py nvme_controller_create --vuid MT240830045RNVMES1D0F0 --nqn nqn.2022-10.io.nvda.nvme:0 --suspended
snap_rpc.py nvme_controller_attach_ns -c NVMeCtrl1 -n 1
snap_rpc.py nvme_controller_resume -c NVMeCtrl1

// Cleanup
snap_rpc.py nvme_controller_detach_ns -c NVMeCtrl1 -n 1
snap_rpc.py emulation_device_detach_prepare --vuid MT240830045RNVMES1D0F0
snap_rpc.py nvme_controller_destroy -c NVMeCtrl1
snap_rpc.py emulation_device_detach --vuid MT240830045RNVMES1D0F0
snap_rpc.py nvme_namespace_destroy -n 1 --nqn nqn.2022-10.io.nvda.nvme:0
snap_rpc.py nvme_subsystem_destroy --nqn nqn.2022-10.io.nvda.nvme:0
spdk_rpc.py bdev_nvme_detach_controller nvme0
```

8.12.2.5.5 SPDK Bdev Management

8.12.2.5.5.1 bdev_list

Lists all active SNAP block devices (**bdevs**). This RPC is useful for identifying and managing custom block devices that operate outside standard SPDK **bdev** specifications.

8.12.2.5.6 Virtio-blk Emulation Management

Virtio-blk emulation is a storage protocol belonging to the virtio family of devices. These devices are found in virtual environments yet by design look like physical devices to the user within the virtual machine.

Each virtio-blk device (e.g., virtio-blk PCIe entry) exposed to the host, whether it is PF or VF, must be backed by a virtio-blk controller.

> ⚠ Virtio-blk limitations:

- Probing a virtio-blk driver on the host without an already functioning virtio-blk controller may cause the host to hang until such controller is opened successfully (no timeout mechanism exists).
- Upon creation of a virtio-blk controller, a backend device must already exist.

8.12.2.5.6.1 Virtio-blk Emulation Management Commands

virtio_blk_controller_create

Create a new SNAP-based virtio-blk controller over a specific PCIe function on the host. To specify the PCIe function to open a controller upon must be provided as described in section "PCIe Function Management":

1. `vuid` (recommended as it is guaranteed to remain constant).
2. `vhca_id.`
3. Function index – `pf_id`, `vf_id`.

The mapping for `pci_index` can be queried by running `emulation_function_list`.

Command parameters:

Parameter	Mandatory?	Type	Description
`vuid`	No	String	PCIe device VUID
`vhca_id`	No	Number	vHCA ID of PCIe function
`pf_id`	No	Number	PCIe PF index to start emulation on
`vf_id`	No	Number	PCIe VF index to start emulation on (if the controller is meant to be opened on a VF)
`pci_bdf`	No	String	PCIe device BDF
`ctrl`	No	String	Controller ID

Parameter	Mandatory?	Type	Description
num_queues	No	Number	Number of IO queues (default 1, range 1-256). It is recommended that the number of MSIX be greater than the number of IO queues (1 is used for the config interrupt). Based on effective num_msix value (can be queried from virtio_blk_controller_list RPC), it can be later aligned using virtio_blk_controller_modify RPC).
queue_size	No	Number	Queue size (default 256, range 1- not limited). Must be a power of 2.
size_max	No	Number	Maximal SGE data transfer size (default 65536 bytes, range 4K - 2M) SNAP's internal memory buffer pool supports a maximum allocation size of 2 MB. This should guide the user when setting the combined value of size_max * seg_max . This limit is not strictly enforced. If the backend supports zero-copy datapath, users may set size_max beyond 2 MB based on their performance or application needs.
seg_max	No	Number	Maximal SGE list length (default 32, range 1- queue_size) seg_max is implicitly upper-bounded by virtio queue size in virtio-blk specification. As such, any provided seg_max value larger than queue_size will be silently clamped down to equal queue_size value.
bdev	No	String	SNAP SPDK block device to use as backend
vblk_id	No	String	Serial number for the controller
admin_q	No	0/1	Enables live migration and NVIDIA vDPA This field is only applicable to controllers created through physical functions (PFs).
dynamic_msix	No	0/1	Dynamic MSIX for SR-IOV VFs on this PF. Only valid for PFs.

Parameter	Mandatory?	Type	Description
`vf_num_msix`	No	Number	Control the number of MSIX tables to associate with this controller. Valid only for VFs (whose parent PF controller is created using the `--dynamic_msix` option) and only when the dynamic MSIX management feature is enabled. Must be an even number ≥2. ⚠ This field is mandatory when the VF's MSIX is reclaimed using `virtio_blk_controller_vfs_msix_reclaim` or released using `--release_msix` on `virtio_blk_controller_destroy`.
`force_in_order`	No	0/1	Support virtio-blk crash recovery. Enabling this parameter to 1 may impact virtio-blk performance (default is 0). For more information, refer to section "Virtio-blk Crash Recovery".
`indirect_desc`	No	0/1	Enables indirect descriptors support for the controller's virt-queues. ⚠ When using the virtio-blk kernel driver, if indirect descriptors are enabled, it is always used by the driver. Using indirect descriptors for all IO traffic patterns may hurt performance in most cases.
`read_only`	No	0/1	Creates read only virtio-blk controller.
`suspended`	No	0/1	Creates controller in suspended state.
`live_update_listener`	No	0/1	Creates controller with the ability to listen for live update notifications via IPC.
`dbg_bdev_type`	No	0/1	N/A – not supported
`dbg_local_optimized`	No	0/1	N/A – not supported

Parameter	Mandatory?	Type	Description
`latency_optimized`	No	0/1	Create controller with (IO read) latency optimization. ⚠ This latency optimization applies only to non-ZCOPY flows. Using this flag may reduce bandwidth performance.
`quirks`	No	Integer (bitmask)	A bitmask used to support buggy or non-compliant drivers. Bit 0 (0x1): Complete protocol-corrupted virtio commands. Handles commands that arrive corrupted at the protocol level (`virtio-blk`) but are technically valid at the transport level (`virtio-pci`). When this bit is set, these commands are completed at the transport level (a "used ring element" is generated), even though they are not handled at the protocol level due to the corruption. Bit 1 (0x2): Wait for RESET after FLR before restarting the data path. Handles cases where, after a PCI Function Level Reset (FLR), the virtio BAR still reflects a stale DRIVER_OK (LIVE) state before the guest issues a real virtio RESET. When this bit is set, SNAP still performs normal FLR teardown, but suppresses LIVE events until a virtio RESET is observed. RESET clears the wait; the next LIVE is handled normally. FLR, RESET, and hotplug events are not blocked, so teardown can always proceed.

Example response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": "VblkCtrl1"
}
```

virtio_blk_controller_destroy

Destroy a previously created virtio-blk controller. The controller can be uniquely identified by the controller's name as acquired from `virtio_blk_controller_create()` .

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name
`force`	No	Boolean	Force destroying VF controller for SR-IOV
`release_queues`	No	Flag	Release queues resources back to free pool. Applies only for VFs.

virtio_blk_controller_suspend

While suspended, the controller stops receiving new requests from the host driver and only finishes handling of requests already in flight. All suspended requests (if any) are processed after resume.

Command parameters:

Parameter	Mandatory?	Type	Description
ctrl	Yes	String	Controller name

virtio_blk_controller_resume

After the controller stops receiving new requests from the host driver (i.e., is suspended) and only finishes handling of requests already in flight, the resume command will resume the handling of IOs by the controller.

Command parameters:

Parameter	Mandatory?	Type	Description
ctrl	Yes	String	Controller name

virtio_blk_controller_bdev_attach

Attach the specified bdev into virtIO-blk SNAP controller. It is possible to change the serial ID (using the **vblk_id** parameter) if a new bdev is attached.

Command parameters:

Parameter	Mandatory?	Type	Description
ctrl	Yes	String	Controller name
bdev	Yes	String	Block device name
vblk_id	No	String	Serial number for controller

virtio_blk_controller_bdev_detach

You may replace the bdev for virtio-blk controller. First, you should detach bdev from the controller. When bdev is detached, the controller stops receiving new requests from the host driver (i.e., is suspended) and finishes handling requests already in flight only.

At this point, you may attach a new bdev or destroy the controller.

When a new bdev is attached, the controller resumes handling all outstanding I/Os.

> ⚠ The block size cannot be changed if the driver is loaded.
>
> bdev may be replaced with a different block size if the driver is not loaded.

> ⚠ A controller with no bdev attached to it is considered a temporary state, in which the controller is not fully operational, and may not respond to some actions requested by the driver.

If there is no imminent intention to call **virtio_blk_controller_bdev_attach**, it is advised to attach a **none** bdev instead. For example:

```
snap_rpc.py virtio_blk_controller_bdev_attach -c VblkCtrl1 --bdev none --dbg_bdev_type null
```

Command parameters:

Parameter	Mandatory?	Type	Description
ctrl	Yes	String	Controller name

virtio_blk_controller_list

Retrieves a list of active virtio-blk SNAP controllers along with their characteristics.

Command parameters:

Parameter	Mandatory?	Type	Description
ctrl	No	String	The specific controller name to query. If omitted, the command returns a list of all controllers.

Example response:


```
{
    "vhca_id": 3,
    "queue_size": 256,
    "seg_max": 32,
    "size_max": 65536,
    "bdev": "Malloc0",
    "vblk_id": "00000000000000000000",
    "indirect_desc": false,
    "discard": true,
    "force_in_order": false,
    "admin_q": false,
    "dbg_local_optimized": false,
    "read_only": false,
    "plugged": true,
    "latency_optimized": false,
    "quirks": 0,
    "state": "STARTED",
    "admin_suspended": false,
    "driver_active": false,
    "device_status": 64,
    "num_msix": 63,
    "min_msix": 2,
    "max_msix": 256,
    "num_queues": 30,
    "min_queues": 1,
    "max_queues": 256,
    "free_msix": 0,
    "free_queues": 20,
    "vf_min_queues": 1,
    "vf_max_queues": 256,
    "vf_min_msix": 2,
    "vf_max_msix": 256
}
```


The PF-level **free_msix** and **free_queues** pools track the resources available for VF controller allocation. Their values are updated according to the rules in the following subsections.

MSI-X Resource Pool Size (free_msix)

The **free_msix** pool size changes according to the following operations:

Operation	MSI-X Pool Size Change
`virtio_blk_controller_create` (creates a VF controller with `--vf_num_msix X`)	We don't have a way to export this macro.
`virtio_blk_controller_modify` (changes `num_msix` from A to B)	We don't have a way to export this macro.

⚠ In the formulas above, S represents the total pool size.

Queue Resource Pool Size (`free_queues`)

The `free_queues` pool size changes according to the following operations:

Operation	Queue Pool Size Change
`virtio_blk_controller_create` (creates a VF controller with `--num_queues X`)	We don't have a way to export this macro.
`virtio_blk_controller_modify` (changes `num_queues` from A to B)	We don't have a way to export this macro.

⚠ In the formulas above, S represents the total pool size.

virtio_blk_controller_modify

This function allows user to modify some of the controller's parameters in real-time, after it was already created.

Modifications can only be done when the emulated function is in idle state - thus there is no driver communicating with it.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	No	String	Controller name
`num_queues`	No	int	Number of queues for the controller

Parameter	Mandatory?	Type	Description
num_msix	No	int	Number of MSIX to be used for a controller. Relevant only for VF controllers (when dynamic MSIX feature is enabled) and hotpluged PF.

⚠ Standard virtio-blk kernel driver currently does not support PCIe FLR. As such,

virtio_blk_controller_dbg_io_stats_get

Debug counters are per-controller I/O stats that can help to know the I/O distribution between different queues of the controller and the total I/O received on the controller.

Command parameters:

Parameter	Mandatory?	Type	Description
ctrl	Yes	String	Controller name

Example response:

```
"ctrl_id": "VblkCtrl2",
"queues": [
  {
    "queue_id": 0,
    "core_id": 0,
    "read_io_count": 19987068,
    "write_io_count": 6319931,
    "flush_io_count": 0
  },
  {
    "queue_id": 1,
    "core_id": 1,
    "read_io_count": 9769556,
    "write_io_count": 3180098,
    "flush_io_count": 0
  }
],
"read_io_count": 29756624,
"write_io_count": 9500029,
"flush_io_count": 0
    }
```

virtio_blk_controller_dbg_debug_stats_get

Debug counters are per-controller debug statistics that can help knowing the controller and queues health and status.

Command parameters:

Parameter	Mandatory?	Type	Description
ctrl	Yes	String	Controller name

Example response:

```
{
  "ctrl_id": "VblkCtrl1",
  "queues": [
    {
      "qid": 0,
      "state": "RUNNING",
      "hw_available_index": 6,
      "sw_available_index": 6,
```

```
      "hw_used_index": 6,
      "sw_used_index": 6,
      "hw_received_descs": 13,
      "hw_completed_descs": 13
    },
    {
      "qid": 1,
      "state": "RUNNING",
      "hw_available_index": 2,
      "sw_available_index": 2,
      "hw_used_index": 2,
      "sw_used_index": 2,
      "hw_received_descs": 6,
      "hw_completed_descs": 6
    },
    {
      "qid": 2,
      "state": "RUNNING",
      "hw_available_index": 0,
      "sw_available_index": 0,
      "hw_used_index": 0,
      "sw_used_index": 0,
      "hw_received_descs": 4,
      "hw_completed_descs": 4
    },
    {
      "qid": 3,
      "state": "RUNNING",
      "hw_available_index": 0,
      "sw_available_index": 0,
      "hw_used_index": 0,
      "sw_used_index": 0,
      "hw_received_descs": 3,
      "hw_completed_descs": 3
    }
  ]
}
```


virtio_blk_controller_vfs_dynamic_resources_reclaim

Reclaims all dynamic resources (MSI-X and queues) from the VFs back to the PF's free pools.

> ⚠ This function can only be applied to PFs and must only be executed when SR-IOV is not set on the host side (i.e., **sriov_numvfs = 0**).

Parameter	Mandatory?	Type	Description
ctrl	Yes	String	The name of the controller. This must be a PF.
skip_msix	No	Flag	If set, skips the reset of dynamic MSI-X resources.
skip_queues	No	Flag	If set, skips the reset of dynamic queue resources.

virtio_blk_controller_vfs_msix_reclaim

Reclaim virtio-blk SNAP controller VFs MSIX back to the free MSIX pool. Valid only for PFs.

Command parameters:

Parameter	Mandatory?	Type	Description
ctrl	Yes	String	Controller name

8.12.2.5.6.2 Virtio-blk Configuration Examples

Virtio-blk Configuration for Single Controller

```
spdk_rpc.py bdev_nvme_attach_controller -b nvme0 -t rdma -a 1.1.1.1 -f ipv4 -s 4420 -n
nqn.2022-10.io.nvda.nvme:swx-storage
```

```
snap_rpc.py virtio_blk_controller_create --vuid MT2114X12200VBLKS1D0F0 --bdev nvme0n1
```

Virtio-blk Cleanup for Single Controller

```
snap_rpc.py virtio_blk_controller_destroy -c VblkCtrl1
spdk_rpc.py bdev_nvme_detach_controller nvme0
```

Virtio-blk Dynamic Configuration For 125 VFs

1. Update the firmware configuration as described section "SR-IOV Firmware Configuration".
2. Reboot the host.
3. Run:

```
[dpu] spdk_rpc.py bdev_nvme_attach_controller -b nvme0 -t rdma -a 1.1.1.1 -f ipv4 -s 4420 -n
nqn.2022-10.io.nvda.nvme:swx-storage
[dpu] snap_rpc.py virtio_blk_controller_create --ctrl VblkCtrl0 --vuid MT2114X12200VBLKS1D0F0
[dpu] for i in `seq 0 124`; do snap_rpc.py virtio_blk_controller_create --pf_id 0 --vf_id $i --bdev nvme0n1
--ctrl VblkCtrl$((i+1)); done;

[host] modprobe -v virtio-pci && modprobe -v virtio-blk
[host] echo 125 > /sys/bus/pci/devices/0000:86:00.3/sriov_numvfs
```

4. For cleanup:

```
[host] echo 0 > /sys/bus/pci/devices/0000:86:00.3/sriov_numvfs
[host] modprobe -rv virtio-pci && modprobe -rv virtio-blk

[dpu] for i in `seq 0 124`; do snap_rpc.py virtio_blk_controller_destroy -c VblkCtrl$((i+1)) -f; done;
[dpu] spdk_rpc.py bdev_nvme_detach_controller nvme0
[dpu] snap_rpc.py snap_rpc.py virtio_blk_controller_destroy -c VblkCtrl$((i+1)) -f
```

Virtio-blk Suspend, Resume Example

```
[host] // Run fio
[dpu] snap_rpc.py virtio_blk_controller_suspend -c VBLKCtrl1
[host] // IOs will get suspended
[dpu] snap_rpc.py virtio_blk_controller_resume -c VBLKCtrl1
[host] // fio will resume sending IOs
```

Virtio-blk Bdev Attach, Detach Example

```
[host] // Run fio
[dpu] snap_rpc.py virtio_blk_controller_bdev_detach -c VBLKCtrl1
[host] // Bdev will be detached and IOs will get suspended
[dpu] snap_rpc.py virtio_blk_controller_bdev_attach -c VBLKCtrl1 --bdev null2
[host] // The null2 bdev will be attached into controller and fio will resume sending IOs
```

Notes

- Virtio-blk protocol controller supports one backend device only
- Virtio-blk protocol does not support administration commands to add backends. Thus, all backend attributes are communicated to the host virtio-blk driver over PCIe BAR and must be accessible during driver probing. Therefore, backends can only be changed once the PCIe function is not in use by any host storage driver.

8.12.2.5.7 NVMe Emulation Management

8.12.2.5.7.1 NVMe Subsystem

The NVMe subsystem as described in the NVMe specification is a logical entity which encapsulates sets of NVMe backends (or namespaces) and connections (or controllers). NVMe subsystems are extremely useful when working with multiple NVMe controllers especially when using NVMe VFs. Each NVMe subsystem is defined by its serial number (SN), model number (MN), and qualified name (NQN) after creation.

The RPCs listed in this section control the creation and destruction of NVMe subsystems.

8.12.2.5.7.2 NVMe Namespace

NVMe namespaces are the representors of a continuous range of LBAs in the local/remote storage. Each namespace must be linked to a subsystem and have a unique identifier (NSID) across the entire NVMe subsystem (e.g., 2 namespaces cannot share the same NSID even if they are linked to different controllers).

After creation, NVMe namespaces can be attached to a controller.

> ⚠ SNAP does not currently support shared namespaces between different controllers. So, each namespace should be attached to a single controller.

The SNAP application uses an SPDK block device framework as a backend for its NVMe namespaces. Therefore, they should be configured in advance. For more information about SPDK block devices, see SPDK bdev documentation and Appendix SPDK Configuration.

8.12.2.5.7.3 NVMe Controller

Each NVMe device (e.g., NVMe PCIe entry) exposed to the host, whether it is a PF or VF, must be backed by NVMe controller, which is responsible for all protocol communication with the host's driver.

Every new NVMe controller must also be linked to an NVMe subsystem. After creation, NVMe controllers can be addressed using either their name (e.g., `Nvmectrl1`) or both their subsystem NQN and controller ID.

8.12.2.5.7.4 Attaching NVMe Namespace to NVMe Controller

After creating an NVMe controller and an NVMe namespace under the same subsystem, the following method is used to attach the namespace to the controller.

8.12.2.5.7.5 NVMe Emulation Management Commands

nvme_subsystem_create

Create a new NVMe subsystem to be controlled by one or more NVMe SNAP controllers. An NVMe subsystem includes one or more controllers, zero or more namespaces, and one or more ports. An NVMe subsystem may include a non-volatile memory storage medium and an interface between the controller(s) in the NVMe subsystem and non-volatile memory storage medium.

Command parameters:

Parameter	Mandatory?	Type	Description
nqn	Yes	String	Subsystem qualified name
serial_number	No	String	Subsystem serial number (Max string length is 20 characters)
model_number	No	String	Subsystem model number
nn	No	Number	Maximal namespace ID allowed in the subsystem (default 0xFFFFFFFE; range 1-0xFFFFFFFE)
mnan	No	Number	Maximal number of namespaces allowed in the subsystem (default 1024; range 1-0xFFFFFFFE)
single_ctrl	No	Boolean	Subsystem is restricted to a single ctrl

Example request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "nvme_subsystem_create",
  "params": {
    "nqn": "nqn.2022-10.io.nvda.nvme:0"
  }
}
```

nvme_subsystem_destroy

Destroy (previously created) NVMe SNAP subsystem.

Command parameters:

Parameter	Mandatory?	Type	Description
nqn	Yes	String	Subsystem qualified name
force	No	Bool	Force the deletion of all the controllers and namespaces under the subsystem

nvme_subsystem_list

List NVMe subsystems.

nvme_namespace_create

Create new NVMe namespaces that represent a continuous range of LBAs in the previously configured bdev. Each namespace must be linked to a subsystem and have a unique identifier (NSID) across the entire NVMe subsystem.

Command parameters:

Parameter	Mandatory?	Type	Description
`nqn`	Yes	String	Subsystem qualified name
`bdev_name`	Yes	String	Block device to use as backend
`nsid`	Yes	Number	Namespace ID
`uuid`	No	Number	Namespace UUID ⚠ To safely detach/attach namespaces, the UUID should be provided to force the UUID to remain persistent.
`dbg_bdev_type`	No	string	Bdev plugin to be attached with the controller. Refer to section "Bdev" for more information.

nvme_namespace_destroy

Destroy a previously created NVMe namespaces.

Command parameters:

Parameter	Mandatory?	Type	Description
`nqn`	Yes	String	Subsystem qualified name
`nsid`	Yes	Number	Namespace ID

nvme_namespace_list

List NVMe SNAP namespaces.

Command parameters:

Parameter	Mandatory?	Type	Description
`nqn`	No	String	Subsystem qualified name

nvme_controller_create

Instantiates a new SNAP-based NVMe block controller over a designated host PCIe function.

Specify the target PCIe function using the `pci_index` parameter. Execute the `emulation_function_list` command to retrieve the correct `pci_index` mappings.

⚠ By default, the controller enables an optional NVMe command only if all namespaces attached during driver load support that specific command.

⚠ Explicitly set the corresponding support flag during creation to override the default behavior. For example, passing `--compare 0` disables the Compare command entirely, regardless of underlying namespace support.

Command parameters:

Parameter	Mandatory?	Type	Description
`nqn`	Yes	String	Subsystem qualified name.
`vuid`	No	Number	VUID of PCIe function.
`pf_id`	No	Number	PCIe PF index to start emulation on.
`vf_id`	No	Number	PCIe VF index to start emulation on (if the controller is destined to be opened on a VF).
`pci_bdf`	No	String	PCIe BDF to start emulation on.
`vhca_id`	No	Number	vHCA ID of PCIe function.
`ctrl`	No	Number	Controller ID.
`num_queues`	No	Number	Number of IO queues (default: 1, range: 1–31). ⚠ Limited by hardware support. ✓ The number of MSIX must be greater than the number of IO queues by at least one for the admin queue.
`mdts`	No	Number	MDTS (default: 7, range: 1–7).
`fw_slots`	No	Number	Maximum number of firmware slots (default: 4).
`write_zeroes`	No	0/1	Enable the `write_zeroes` optional NVMe command.
`compare`	No	0/1	Set the value of the compare support bit in the controller.

Parameter	Mandatory?	Type	Description
`compare_write`	No	0/1	Set the value of the `compare_write` support bit in the controller. ⚠ During crash recovery, all compare and write commands are expected to fail.
`deallocate_dsm`	No	0/1	Set the value of the DSM (dataset management) support bit in the controller. Only the `deallocate` DSM request is currently supported.
`suspended`	No	0/1	Open the controller in suspended state (requires `nvme_controller_resume` to become active). Recommended for recovery scenarios or when creating the controller while the driver is loaded.
`snapshot`	No	String	Create a controller from a snapshot file path (snapshot must be taken using `nvme_controller_snapshot_get`).
`dynamic_msix`	No	0/1	Enable dynamic MSIX management for the controller (default: 0). Applies only for PFs.
`vf_num_msix`	No	Number	Number of MSIX tables to associate with this controller. Valid only for VFs whose parent PF controller was created with `--dynamic_msix`, and when dynamic MSIX management is enabled. ⚠ Mandatory if VF's MSIX is reclaimed using `nvme_controller_vfs_msix_reclaim` or released using `--release_msix` on `nvme_controller_destroy`.
`admin_only`	No	0/1	Creates NVMe controller with admin queues only (no IO queues).
`quirks`	No	Number	Bitmask to support buggy/non-compliant drivers. • Bit 0 – Send "Namespace Attribute Changed" async event, even if disabled by driver. • Bit 1 – Keep sending these events even without a "Changed Namespace List" Get Log Page request. • Bit 2 – reserved. Should not be set. • Bit 3 – Force-enable "Namespace Management capability" NVMe OACS even if unsupported. • Bit 4 – Disable Scatter-Gather Lists support. See “OS Issues” section for details.
`queue_core_alloc_policy`	No	String	Core allocation policy: `"pre-allocation"` (predictable but may be imbalanced if active queues differ from configured), or `"on-demand"` (auto-balances at runtime, unpredictable upfront). Default: `"on-demand"`. ⚠ Core-to-queue mapping is not persistent across recovery/live update/migration.

Parameter	Mandatory?	Type	Description
`admin_identify_cmd_fwd`	No	String	Enable forwarding of Identify admin commands with selected CNS values (comma-separated hex with `0x` prefix) from SNAP to the upper application. Example: `--admin_identify_cmd_fwd 0x0,0x1,0x13`
`admin_set_features_cmd_fwd`	No	String	Enable forwarding of Set Features admin commands with selected FID values (comma-separated hex with `0x` prefix) from SNAP to the upper application. Example: `--admin_set_features_cmd_fwd 0xc0,0xe4`

`queue_depth`	No	Number	Specifies the maximum IO queue size for the controller. This parameter is valid only for VFs.
			• Range: 2–4096 (inclusive).
			• If set to `0` (default), the system uses the pre-configured value based on the following hierarchy:
			  • The value last set on the emulation function since the last cold reboot (persistent across application crashes or restarts).
			  • If never set, the `NVME_EMULATION_MAX_QUEUE_DEPTH` firmware parameter.
			SR-IOV configuration modifying this parameter on a VF while the parent PF has active VFs (i.e., `sriov_numvfs` or `sriov_totalvfs` is not `0`) leads to undefined behavior.
			Warning The actual queue depth utilized is restricted by the NVMe driver parameter `io_queue_depth` configured on the host machine.
			• Requirement: The host module's `io_queue_depth` should be configured to be at least equal to the `queue_depth` set in this RPC.
			• Persistence: To ensure this applies at boot, the parameter must be added to the host's kernel command line (`/proc/cmdline`).

Warning The output of the `nvme_controller_create` RPC explicitly excludes the admin queue from the count of all queue-related parameters. For users migrating from older SNAP versions, parameters such as `max_queues` and `vf_max_queues` will appear decreased by 1. This ensures that queue parameters consistently reflect only I/O queues.

Example request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "nvme_controller_create",
  "params": {
```

```
    "nqn": "nqn.2022-10.io.nvda.nvme:0",
    "pf_id": 0,
    "num_queues": 8,
  }
}
```

nvme_controller_destroy

Destroy a previously created NVMe controller. The controller can be uniquely identified by a controller name as acquired from `nvme_controller_create`.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name
`release_msix`	No	1/0	Release MSIX back to free pool. Applies only for VFs.
`force`	No	1/0	Forcefully destroy the controller (without this flag set, SNAP may block deletion in case driver is up). ⚠ After controller is destroyed with `--force` flag set, it is not guaranteed SNAP will perform a successful recovery after this. It is the user's responsibility to make sure either driver is properly cleaned up or controller is re-created with exact same configuration, before resuming SNAP controller back.
`release_queues`	No	Flag	Release queues resources back to free pool. Applies only for VFs.

nvme_controller_suspend

While suspended, the controller stops handling new requests from the host driver. All pending requests (if any) will be processed after resume.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name

Parameter	Mandatory?	Type	Description
`timeout_ms`	No	Number	Suspend timeout ⚠ The maximum time in milliseconds to let the controller finish all the inflight I/Os before discarding them. If `timeout_ms` is not provided, the operation waits until the IOs complete without a timeout on the SNAP layer.
`admin_only`	No	0/1	Suspend only the admin queue
`live_update_notifier`	No	0/1	Send a live update notification via IPC

nvme_controller_resume

The resume command continues the (previously suspended) controller's handling of new requests sent by the driver. If the controller is created in suspended mode, resume is also used to start initial communication with host driver.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name

nvme_controller_snapshot_get

Captures the current state of the controller and saves it to a file, which can then be used to instantiate a new controller.

⚠ To ensure data consistency, suspend the controller using the `nvme_controller_suspend` RPC before executing this function.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name
`filename`	Yes	String	File path

nvme_controller_vfs_dynamic_resources_reclaim

Reclaims all dynamic resources (MSI-X and queues) from the VFs back to the PF's free pools.

⚠ This function can only be applied to PFs and must only be executed when SR-IOV is not set on the host side (i.e., `sriov_numvfs = 0`).

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	The name of the controller. This must be a PF.
`skip_msix`	No	Flag	If set, skips the reset of dynamic MSI-X resources.
`skip_queues`	No	Flag	If set, skips the reset of dynamic queue resources.

nvme_controller_vfs_msix_reclaim

Reclaims all VFs MSIX back to the PF's free MSIX pool.

This function can only be applied on PFs and can only be run when SR-IOV is not set on host side (i.e., `sriov_numvfs = 0`).

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name

nvme_controller_list

Retrieves a list of all active (created) NVMe controllers along with their characteristics.

Command parameters:

Parameter	Mandatory?	Type	Description
`nqn`	No	String	The subsystem qualified name
`ctrl`	No	String	A specific controller name to search for

Example response:

```
{
    "ctrl_id": "NVMeCtrl1",
    "mdts": 7,
    "vhca_id": 2,
    "nqn": "nqn.2014-08.com.nvda:nvme:0",
    "plugged": true,
    "state": "STARTED",
    "admin_suspended": false,
    "admin_only": false,
    "driver_active": true,
    "cc": 4588033,
    "csts": 1,
    "subsys_cntlid": 1,
    "admin_identify_cmd_fwd": "",
    "admin_set_features_cmd_fwd": "",
    "num_msix": 32,
    "min_msix": 2,
    "max_msix": 64,
    "num_queues": 16,
    "min_queues": 0,
    "max_queues": 31,
    "num_used_queues": 0,
    "free_msix": 0,
    "free_queues": 0,
    "vf_min_queues": 0,
    "vf_max_queues": 31,
    "vf_min_msix": 2,
    "vf_max_msix": 64,
    "namespaces": []
}
```

The PF's **free_msix** and **free_queues** pools track the resources available for VF NVMe controller allocation. Their values are updated according to the rules in the following subsections.

MSI-X Resource Pool Size (free_msix)

The **free_msix** pool size changes according to the following operations:

Operation	MSI-X Pool Size Change
nvme_controller_create (creates a VF controller with **--vf_num_msix X**)	We don't have a way to export this macro.
nvme_controller_modify (changes **num_msix** from A to B)	We don't have a way to export this macro.

⚠ In the formulas above, S represents the total pool size.

Queue Resource Pool Size (free_queues)

The **free_queues** pool size changes according to the following operations:

Operation	Queue Pool Size Change
nvme_controller_create (creates a VF controller with **--num_queues X**)	We don't have a way to export this macro.
nvme_controller_modify (changes **num_queues** from A to B)	We don't have a way to export this macro.

⚠ In the formulas above, S represents the total pool size.

nvme_controller_modify

This function allows modifying some of the controller's parameters in real-time after it was already created.

Modifications can only be made when the emulated function is in idle state (i.e., no driver is communicating with it).

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name
`num_queues`	No	int	Number of queues for the controller
`num_msix`	No	int	Number of MSIX to be used for a controller. Relevant only for VF controllers (when dynamic MSIX feature is enabled) and hotpluged PF controllers.
`queue_depth`	No	Number	Specifies the maximum IO queue size. This parameter is valid only for VFs. • Range: 2–4096 (inclusive). • If set to `0` (default) the system uses the pre-configured value based on the following hierarchy: a. The value last set on the emulation function since the last cold reboot (persistent across application crashes or restarts). b. If never set, the `NVME_EMULATION_MAX_QUEUE_DEPTH` firmware parameter. SR-IOV configuration modifying this parameter on a VF while the parent PF has active VFs (i.e., `sriov_numvfs` or `sriov_totalvfs` is not `0`) leads to undefined behavior.

nvme_controller_attach_ns

Attach a previously created NVMe namespace to given NVMe controller under the same subsystem.

The result in the response object returns `true` for success and `false` for failure.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name
`nsid`	Yes	Number	Namespace ID

nvme_controller_detach_ns

Detach a previously attached namespace with a given NSID from the NVMe controller.

The result in the response object returns `true` for success and `false` for failure.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name

Parameter	Mandatory?	Type	Description
nsid	Yes	Number	Namespace ID

nvme_controller_dbg_debug_stats_get

Display debug counters for NVMe controller as following:

BAR Registers

Register	Description
CC	Controller Configuration register.
CSTS	Controller Status register.

Submission Queue (SQS) Fields

Field	Description
qid	Queue ID.
depth	Queue depth.
cqid	Associated completion queue ID.
state	Current queue state.
driver_sq_tail	Value of the SQ Tail Doorbell register as written by the driver.
device_sq_head	Value of the SQ Head pointer within the device.
Requests	Total number of requests received on this queue.

Completion Queue (CQS) Fields

Field	Description
qid	Queue ID.
depth	Queue depth.
state	Current queue state.
driver_cq_head	Value of the CQ Head Doorbell register as written by the driver.
device_cq_tail	Value of the CQ Tail pointer within the device.
Completions	Total number of completions sent on this queue.

Command Parameters

Parameter	Mandatory?	Type	Description
ctrl	Yes	String	Controller name

nvme_controller_dbg_io_stats_get

The result in the response object returns `true` for success and `false` for failure.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name

```
"ctrl_id": "NVMeCtrl2",
"queues": [
  {
    "queue_id": 0,
    "core_id": 0,
    "read_io_count": 19987068,
    "write_io_count": 6319931,
    "flush_io_count": 0
  },
  {
    "queue_id": 1,
    "core_id": 1,
    "read_io_count": 9769556,
    "write_io_count": 3180098,
    "flush_io_count": 0
  }
],
"read_io_count": 29756624,
"write_io_count": 9500029,
"flush_io_count": 0
    }
```

8.12.2.5.7.6 NVMe Configuration Examples

NVMe Configuration for Single Controller

On the DPU:

```
spdk_rpc.py bdev_nvme_attach_controller -b nvme0 -t rdma -a 1.1.1.1 -f ipv4 -s 4420 -n
nqn.2022-10.io.nvda.nvme:swx-storage
snap_rpc.py nvme_subsystem_create --nqn nqn.2022-10.io.nvda.nvme:0
snap_rpc.py nvme_namespace_create -b nvme0n1 -n 1 --nqn nqn.2022-10.io.nvda.nvme:0 --uuid 263826ad-19a3-4feb-
bc25-4bc81ee7749e
snap_rpc.py nvme_controller_create --nqn nqn.2022-10.io.nvda.nvme:0 --pf_id 0 --suspended
snap_rpc.py nvme_controller_attach_ns -c NVMeCtrl1 -n 1
snap_rpc.py nvme_controller_resume -c NVMeCtrl1
```

⚠ It is necessary to create a controller in a suspended state. Afterward, the namespaces can be attached, and only then should the controller be resumed using the `nvme_controller_resume` RPC.

⚠ To safely detach/attach namespaces, the UUID must be provided to force the UUID to remain persistent.

NVMe Cleanup for Single Controller

```
snap_rpc.py nvme_controller_detach_ns -c NVMeCtrl2 -n 1
snap_rpc.py nvme_controller_destroy -c NVMeCtrl2
snap_rpc.py nvme_namespace_destroy -n 1 --nqn nqn.2022-10.io.nvda.nvme:0
snap_rpc.py nvme_subsystem_destroy --nqn nqn.2022-10.io.nvda.nvme:0
spdk_rpc.py bdev_nvme_detach_controller nvme0
```

NVMe and Hotplug Cleanup for Single Controller

```
snap_rpc.py nvme_controller_detach_ns -c NVMeCtrl1 -n 1
snap_rpc.py emulation_device_detach_prepare --vuid MT2408300045RNVMES1D0F0
snap_rpc.py nvme_controller_destroy -c NVMeCtrl1
snap_rpc.py emulation_device_detach --vuid MT2408300045RNVMES1D0F0
snap_rpc.py nvme_namespace_destroy -n 1 --nqn nqn.2022-10.io.nvda.nvme:0
snap_rpc.py nvme_subsystem_destroy --nqn nqn.2022-10.io.nvda.nvme:0
spdk_rpc.py bdev_nvme_detach_controller nvme0
```

NVMe Configuration for 125 VFs SR-IOV

1. Update the firmware configuration as described section "SR-IOV Firmware Configuration".
2. Reboot the host.
3. Create a dummy controller on the parent PF:

```
[dpu] # snap_rpc.py nvme_subsystem_create --nqn nqn.2022-10.io.nvda.nvme:0
[dpu] # snap_rpc.py nvme_controller_create --nqn nqn.2022-10.io.nvda.nvme:0 --ctrl NVMeCtrl1 --pf_id 0 --
admin_only
```

4. Create 125 Bdevs (Remote or Local), 125 NSs and 125 controllers:

```
[dpu] for i in `seq 0 124`; do \
   # spdk_rpc.py bdev_null_create null$((i+1)) 64 512;
   # snap_rpc.py nvme_namespace_create -b null$((i+1)) -n $((i+1)) --nqn nqn.2022-10.io.nvda.nvme:0 --uuid
3d9c3b54-5c31-410a-b4f0-7cf2afd9e$((i+100));
   # snap_rpc.py nvme_controller_create --nqn nqn.2022-10.io.nvda.nvme:0 --ctrl NVMeCtrl$((i+2)) --pf_id 0
--vf_id $i --suspended;
   # snap_rpc.py nvme_controller_attach_ns -c NVMeCtrl$((i+2)) -n $((i+1));
   # snap_rpc.py nvme_controller_resume -c NVMeCtrl$((i+2));
done
```

5. Load the driver and configure VFs:

```
[host] # modprobe -v nvme
[host] # echo 125 > /sys/bus/pci/devices/0000\:25\:00.2/sriov_numvfs
```

8.12.2.5.8 Environment Variable Management

8.12.2.5.8.1 snap_global_param_list

snap_global_param_list lists all existing environment variables.

The following is an example response for the **snap_global_param_list** command:

```
[
  "SNAP_ENABLE_POLL_SKIP       : set     : 0  ",
  "SNAP_POLL_CYCLE_SIZE        : not set : 16 ",
  "SNAP_RPC_LOG_ENABLE         : set     : 1  ",
  "SNAP_MEMPOOL_SIZE_MB        : set     : 1024",
  "SNAP_MEMPOOL_4K_BUFFS_PER_CORE : not set : 1024",
  "SNAP_RDMA_ZCOPY_ENABLE      : set     : 1  ",
  "SNAP_TCP_XLIO_ENABLE        : not set : 1  ",
  "SNAP_TCP_XLIO_TX_ZCOPY      : not set : 1  ",
  "MLX5_SHUT_UP_BF             : not set : 0  ",
  "SNAP_SHARED_RX_CQ           : not set : 1  ",
  "SNAP_SHARED_TX_CQ           : not set : 1  ",
...
```

8.12.2.6 SNAP-4 Service Advanced Features

8.12.2.6.1 RPC Log History

RPC log history (enabled by default) records all the RPC requests (from **snap_rpc.py** and **spdk_rpc.py**) sent to the SNAP application and the RPC response for each RPC requests in a

dedicated log file, **/var/log/snap-log/rpc-log**. This file is visible outside the container (i.e., the log file's path on the DPU is **/var/log/snap-log/rpc-log** as well).

The **SNAP_RPC_LOG_ENABLE** env can be used to enable (**1**) or disable (**0**) this feature.

RPC log history is supported with SPDK version spdk23.01.2-12 and above.

When RPC log history is enabled, the SNAP application writes (in append mode) RPC request and response message to **/var/log/snap-log/rpc-log** constantly. Pay attention to the size of this file. If it gets too large, delete the file on the DPU before launching the SNAP pod.

8.12.2.6.2 SR-IOV

SR-IOV configuration depends on the kernel version and must be handled carefully to ensure device visibility and system stability across both hypervisor and DPU orchestrators.

To ensure a safe and stable SR-IOV setup, follow these steps:

1. Preconfigure VF controllers on the DPU – Before configuring SR-IOV on the host, ensure that the DPU is properly configured with all required VF controllers already created and opened.
    - VF functions are always visible and configurable on the DPU side. Use the following command to verify:

      ```
      snap_rpc.py emulation_function_list --all
      ```

    - Confirm that the configuration meets your requirements.
        - Check the number of resources allocated for the PF, specifically MSIX and queues (queried using **snap_rpc.py virtio_blk_controller_list** RPC output, field **free_queues** and **free_msix**) are enough to satisfy your needs for all underlying VFs.
        - Use dynamic MSIX if needed and supported.
    - Once host-side configuration begins, further modifications may not be possible.
2. Disable autoprobing with **sriov_drivers_autoprobe=0** – In deployments with many virtual devices, autoprobing must be disabled to ensure stable device discovery. Failing to disable autoprobing may result in:
    - Incomplete device visibility
    - Missing virtual disks
    - System hangs during initialization
    - Unreliable behavior in large-scale environments (more than 100 VFs)

      Recommended configuration for large-scale deployments:
        - Disable autoprobe:

          ```
          echo 0 > /sys/bus/pci/devices/<BDF>/sriov_drivers_autoprobe
          ```

- Manually bind the VFs to drivers using tools such as **driverctl**, or by writing to **bind** / **unbind** in **sysfs**.

3. Configure SR-IOV on the host – For small-scale deployments (fewer than 100 VFs), use the **sriov_totalvfs** entry:

```
echo <number_of_vfs> > /sys/bus/pci/devices/<BDF>/sriov_totalvfs
```

For newer drivers, use:

```
echo <number_of_vfs> > /sys/bus/pci/devices/<BDF>/sriov_numvfs
```

> ⚠ After SR-IOV configuration, no disks appear in the hypervisor by default. Disks are only visible inside VMs once the corresponding PCIe VF is assigned to the VM via a virtualization manager (e.g., libvirt, VMware). To use the device directly from the hypervisor, manually bind the VF to the appropriate driver.

Additional notes:

- Hot-plugged PFs do not support SR-IOV.
- For deployments requiring more than 127 VFs, add the following kernel parameter to the host's boot command line:

```
pci=assign-busses
```

Without this, the host may log errors such as:

```
pci 0000:84:00.0: [1af4:1041] type 7f class 0xffffff
pci 0000:84:00.0: unknown header type 7f, ignoring device
```

These errors prevent the virtio driver from probing the device.

8.12.2.6.3 Hot-pluggable PCIe Functions

PCI hotplug functions allow the addition or removal of PCI devices from a running system without requiring a machine shutdown or interrupting operations. This technology relies on hardware and software support to isolate individual slots, manage power and control signals, and ensure proper power sequencing during device insertion or removal. When a device is hotplugged, the operating system's hotplug service handles quiescing the device driver, controlling the physical slot, and managing power states while the bus remains active.

SNAP enables dynamic hot-plugging and hot-unplugging of PCIe functions at runtime via dedicated RPCs (see section "Hot-pluggable PCIe Functions Management"). Once a PCIe function is hotplugged, it persists in the host's PCIe device list until it is explicitly unplugged via RPC or a firmware reset occurs. This persistence is maintained even if the SNAP process terminates.

8.12.2.6.3.1 Two-step PCIe Hotplug

The two-step PCIe hotplug method divides the process into two distinct phases:

1. PCIe function creation (the function is accessible within the DPU application but remains hidden from the host OS/kernel).
2. PCIe function exposure to the host.

Symmetrically, hot-unplugging involves hiding the function from the host before destroying it.

A primary advantage of this approach is the ability to pre-configure a controller on the function prior to exposing it to the host driver. Users are required to create a controller to initiate the host exposure phase.

Sync vs. Async Hotplug Operations

Exposing a new PCIe function involves notifying the host and awaiting its response. SNAP provides two modes for this operation, controlled via the **--wait_for_done** option:

- Synchronized hotplug – The RPC blocks until the host acknowledges the action and updates its PCIe function list, or until the host is determined to be temporarily unavailable.
- Asynchronized hotplug – The RPC does not block. The user must actively validate the function's hotplug state using the **emulation_functions_list** RPC to check for host acceptance.

Synchronized operations (using **--wait_for_done**) are recommended for single hotplug/unplug actions. For multiple operations, it is more time-efficient to execute all actions asynchronously (without **--wait_for_done**) and then validate their status altogether with a single **emulation_functions_list** RPC call.

Host Awareness

During periods such as OS reboots, the host may temporarily be unable to accept new PCIe functions. SNAP offers two modes to handle these scenarios, configurable via the **--mode** parameter:

- Host-aware mode (recommended) – The device checks the host's state and takes it into consideration. If the host is unready, the firmware blocks the command and returns an RPC error. Users must manually retry the operation if a failure occurs.
- Host-unaware mode – The operation proceeds regardless of the host's state. The user is responsible for ensuring the PCI enumeration process completes successfully and avoiding undesired behaviors, as the firmware will not block the command.

Attention Button

Because the Linux Virtio PCI driver stack does not support the surprise removal of devices, ungraceful removal can cause the host PCI subsystem to repeatedly attempt device recovery (like resetting the device and reprogramming MSI-X vectors) even after it is gone.

The Attention Button feature, defined in the PCIe specification, enables a graceful hot-plug/hot-unplug sequence. When utilized via the **host_aware_ab** mode, all state transitions are intentionally delayed by 5 seconds to allow the host OS time to react to the corresponding PCIe events.

⚠ Issuing a hot-unplug RPC immediately after an async hotplug RPC without `--wait_for_done` (or vice versa) using the `host_aware_ab` mode will cause the second RPC to return a failure. However, this failure will not affect the results of the initial RPC

8.12.2.6.3.2 Legacy ("One-step") PCIe Hotplug

The deprecated legacy mechanism uses a single-step "create-and-expose" flow where the emulation function is instantiated on the DPU and immediately presented to the host in one RPC. This tight coupling means the function's lifecycle is directly bound to host enumeration without any intermediate hidden state.

This legacy approach has several structural limitations:

- The controller cannot be fully configured, tested, and bound to backing storage before being exposed to the host.
- Failures in host enumeration translate directly into RPC failures or undefined host behavior, as the operation cannot be retried on a hidden function.
- Functions cannot be hidden from the host for maintenance, migration, or gradual rollout while keeping the emulation object alive.
- It lacks options for selecting advanced host-awareness settings or the Attention Button flow.

8.12.2.6.4 Zero Copy (SNAP-direct)

⚠ Zero-copy is supported on SPDK 21.07 and higher.

SNAP-direct allows SNAP applications to transfer data directly from the host memory to remote storage without using any staging buffer inside the DPU.

SNAP enables the feature according to the SPDK BDEV configuration only when working against an SPDK NVMe-oF RDMA block device.

To enable zero copy, set the environment variable (as it is enabled by default):

```
SNAP_RDMA_ZCOPY_ENABLE=1
```

For more info refer to SNAP-4 Service Environment Variables.

8.12.2.6.5 NVMe/TCP XLIO Zero Copy

NVMe/TCP Zero Copy is implemented as a custom `NVDA_TCP` transport in SPDK NVMe initiator, and it is based on a new XLIO socket layer implementation.

The implementation is different for Tx and Rx:

- The NVMe/TCP Tx Zero Copy is similar between RDMA and TCP in that the data is sent from the host memory directly to the wire without an intermediate copy to Arm memory

- The NVMe/TCP Rx Zero Copy allows achieving partial zero copy on the Rx flow by eliminating copy from socket buffers (XLIO) to application buffers (SNAP). But data still must be DMA'ed from Arm to host memory.

To enable NVMe/TCP Zero Copy, use **SPDK v22.05.nvda --with-xlio** (**v22.05.nvda** or higher).

> For more information about XLIO including limitations and bug fixes, refer to the *NVIDIA Accelerated IO (XLIO) Documentation*.

To enable SNAP TCP XLIO Zero Copy:

1. SNAP container: Set the environment variables and resources in the YAML file to request 6G of hugepages:

```
resources:
  requests:
    memory: "4Gi"
    cpu: "8"
  limits:
    hugepages-2Mi: "6Gi"
    memory: "6Gi"
    cpu: "16"

## Set according to the local setup
env:
  - name: APP_ARGS
    value: "--wait-for-rpc"
  - name: SPDK_XLIO_PATH
    value: "/usr/lib/libxlio.so"
```

2. SNAP sources: Set the environment variables and resources in the relevant scripts
   a. In **run_snap.sh** , edit the **APP_ARGS** variable to use the SPDK command line argument **--wait-for-rpc** :

```
APP_ARGS="--wait-for-rpc"
```

   b. In **set_environment_variables.sh** , uncomment the **SPDK_XLIO_PATH** environment variable:

```
export SPDK_XLIO_PATH="/usr/lib/libxlio.so"
```

> NVMe/TCP XLIO requires a BlueField Arm OS hugepage size of 4Gi. For information on configuring the hugepages, refer to sections "Step 1: Allocate Hugepages" and "Adjusting YAML Configuration".
>
> At high scale, it is required to use the global variable **XLIO_RX_BUFS=4096** even though it leads to high memory consumption. Using **XLIO_RX_BUFS=1024** requires lower memory consumption but limits the ability to scale the workload.

> For more info refer to SNAP-4 Service Environment Variables.

> It is recommended to configure NVMe/TCP XLIO with the transport ack timeout option increased to 12.

```
[dpu] spdk_rpc.py bdev_nvme_set_options --transport-ack-timeout 12
```

Other **bdev_nvme** options may be adjusted according to requirements.

Expose an NVMe-oF subsystem with one namespace by using a TCP transport type on the remote SPDK target.

```
[dpu] spdk_rpc.py sock_set_default_impl -i xlio
[dpu] spdk_rpc.py framework_start_init
[dpu] spdk_rpc.py bdev_nvme_set_options --transport-ack-timeout 12
[dpu] spdk_rpc.py bdev_nvme_attach_controller -b nvme0 -t nvda_tcp -a 3.3.3.3 -f ipv4 -s 4420 -n
nqn.2023-01.io.nvmet
[dpu] snap_rpc.py nvme_subsystem_create --nqn nqn.2023-01.com.nvda:nvme:0
[dpu] snap_rpc.py nvme_namespace_create -b nvme0n1 -n 1 --nqn nqn. 2023-01.com.nvda:nvme:0 --uuid 16dab065-
ddc9-8a7a-108e-9a489254a839
[dpu] snap_rpc.py nvme_controller_create --nqn nqn.2023-01.com.nvda:nvme:0 --ctrl NVMeCtrl1 --pf_id 0 --suspended
--num_queues 16
[dpu] snap_rpc.py nvme_controller_attach_ns -c NVMeCtrl1 -n 1
[dpu] snap_rpc.py nvme_controller_resume -c NVMeCtrl1 -n 1

[host] modprobe -v nvme
[host] fio --filename /dev/nvme0n1 --rw randrw --name=test-randrw --ioengine=libaio --iodepth=64 --bs=4k --direct=1
--numjobs=1 --runtime=63 --time_based --group_reporting --verify=md5
```

For more information on XLIO, please refer to XLIO documentation.

8.12.2.6.6 Encryption

The SPDK version included with SNAP supports hardware encryption/decryption offload. To enable AES/XTS and allow the **mlx5_2** and **mlx5_3** SFs to support encryption, they must be designated as trusted.

1. Edit the configuration file **/etc/mellanox/mlnx-sf.conf**.
2. Append the following commands to configure the **VHCA_TRUST_LEVEL** and create the SFs:

   ```
   /usr/bin/mlxreg -d 03:00.0 --reg_name VHCA_TRUST_LEVEL --yes --indexes "vhca_id=0x0,all_vhca=0x1" --set
   "trust_level=0x1"
   /usr/bin/mlxreg -d 03:00.1 --reg_name VHCA_TRUST_LEVEL --yes --indexes "vhca_id=0x0,all_vhca=0x1" --set
   "trust_level=0x1"
   /sbin/mlnx-sf -action create --device 0000:03:00.0 --sfnum 0 --hwaddr 02:11:3c:13:ad:82
   /sbin/mlnx-sf -action create --device 0000:03:00.1 --sfnum 0 --hwaddr 02:76:78:b9:6f:52
   ```

3. Reboot the DPU to apply these changes.

8.12.2.6.6.1 Zero Copy (SNAP-direct) with Encryption

SNAP offers support for zero copy with encryption for **bdev_nvme** with an RDMA transport.

If another **bdev_nvme** transport or base bdev other than NVMe is used, then zero copy flow is not supported, and additional DMA operations from the host to the BlueField Arm are performed.

Refer to section "SPDK Crypto Example" to see how to configure zero copy flow with AES_XTS offload.

Command	Description
`mlx5_scan_accel_module`	Accepts a list of devices to be used for the crypto operation
`accel_crypto_key_create`	Creates a crypto key
`bdev_nvme_attach_controller`	Constructs NVMe block device
`bdev_crypto_create`	Creates a virtual block device which encrypts write IO commands and decrypts read IO commands

8.12.2.6.6.2 mlx5_scan_accel_module

Accepts a list of devices to use for the crypto operation provided in the `--allowed-devs` parameter. If no devices are specified, then the first device which supports encryption is used.

For best performance, it is recommended to use the devices with the largest InfiniBand MTU (4096). The MTU size can be verified using the `ibv_devinfo` command (look for the max and active MTU fields). Normally, the `mlx5_2` device is expected to have an MTU of 4096 and should be used as an allowed crypto device.

Command parameters:

Parameter	Mandatory?	Type	Description
`qp-size`	No	Number	QP size
`num-requests`	No	Number	Size of the shared requests pool
`allowed-devs`	No	String	Comma-separated list of allowed device names (e.g., "mlx5_2") ⚠ Make sure that the device used for RDMA traffic is selected to support zero copy.
`enable-driver`	No	Boolean	Enables accel_mlx5 platform driver. Allows AES_XTS RDMA zero copy.

8.12.2.6.6.3 accel_crypto_key_create

Creates crypto key. One key can be shared by multiple bdevs.

Command parameters:

Parameter	Mandatory?	Type	Description
`cipher`	Yes	Number	Crypto protocol (AES_XTS)
`key`	Yes	Number	Key
`key2`	Yes	Number	Key2

Parameter	Mandatory?	Type	Description
name	Yes	String	Key name

8.12.2.6.6.4 bdev_nvme_attach_controller

Creates NVMe block device.

Command parameters:

Parameter	Mandatory?	Type	Description
name	Yes	String	Name of the NVMe controller, prefix for each bdev name
trtype	Yes	String	NVMe-oF target trtype (e.g., rdma, pcie)
traddr	Yes	String	NVMe-oF target address (e.g., an IP address or BDF)
trsvcid	No	String	NVMe-oF target trsvcid (e.g., a port number)
addrfam	No	String	NVMe-oF target adrfam (e.g., ipv4, ipv6)
nqn	No	String	NVMe-oF target subnqn

8.12.2.6.6.5 bdev_crypto_create

This RPC creates a virtual crypto block device which adds encryption to the base block device.

Command parameters:

Parameter	Mandatory?	Type	Description
base_bdev_name	Yes	String	Name of the base bdev
name	Yes	String	Crypto bdev name
key-name	Yes	String	Name of the crypto key created with accel_crypto_key_create

8.12.2.6.6.6 SPDK Crypto Example

The following is an example of a configuration with a crypto virtual block device created on top of bdev_nvme with RDMA transport and zero copy support:

```
[dpu] # spdk_rpc.py mlx5_scan_accel_module --allowed-devs "mlx5_2" --enable-driver
[dpu] # spdk_rpc.py framework_start_init
[dpu] # spdk_rpc.py accel_crypto_key_create -c AES_XTS -k 00112233445566778899001122334455 -e
11223344556677889900112233445500 -n test_dek
[dpu] # spdk_rpc.py bdev_nvme_attach_controller -b nvme0 -t rdma -a 1.1.1.1 -f ipv4 -s 4420 -n
nqn.2016-06.io.spdk:cnode0
[dpu] # spdk_rpc.py bdev_crypto_create nvme0n1 crypto_0 -n test_dek
[dpu] # snap_rpc.py nvme_subsystem_create --nqn nqn.2023-05.io.nvda.nvme:0
[dpu] # snap_rpc.py nvme_controller_create --nqn nqn.2023-05.io.nvda.nvme:0 --pf_id 0 --ctrl NVMeCtrl0 --suspended
[dpu] # snap_rpc.py nvme_namespace_create -nqn nqn.2023-05.io.nvda.nvme:0 --bdev_name crypto_0 --nsid 1 --uuid
263826ad-19a3-4feb-bc25-4bc81ee7749e
[dpu] # snap_rpc.py nvme_controller_attach_ns --ctrl NVMeCtrl0 --nsid 1
[dpu] # snap_rpc.py nvme_controller_resume --ctrl NVMeCtrl0
```

8.12.2.6.7 Virtio-blk Live Migration

Live migration is a standard process supported by QEMU which allows system administrators to pass devices between virtual machines in a live running system. For more information, refer to QEMU VFIO Device Migration documentation.

Live migration is supported for SNAP virtio-blk devices in legacy and standard VFIO modes. Legacy mode uses drivers like NVIDIA's proprietary vDPA-based Live Migration Solution, while standard mode leverages the latest kernel capabilities using the **virtio-vfio-pci** kernel driver. Legacy mode can be enabled/disabled using the environment variable `VIRTIO_CTRL_VDPA_ADMIN_Q` (enabled by default).

> ⚠ In the standard virtio live migration process, the device is expected to complete all inflight I/Os, with no configurable timeout. If the remote storage is unavailable (disconnected or non-responsive), the device migration will wait indefinitely. This means migration time cannot be guaranteed, representing a degradation compared to the functionality of legacy mode.

8.12.2.6.7.1 Software Requirements for Standard VFIO

- Kernel – 6.16-rc3+ (using **virtio-vfio-pci** driver)
- QEMU – 9.2+
- libvirt – 10.6+

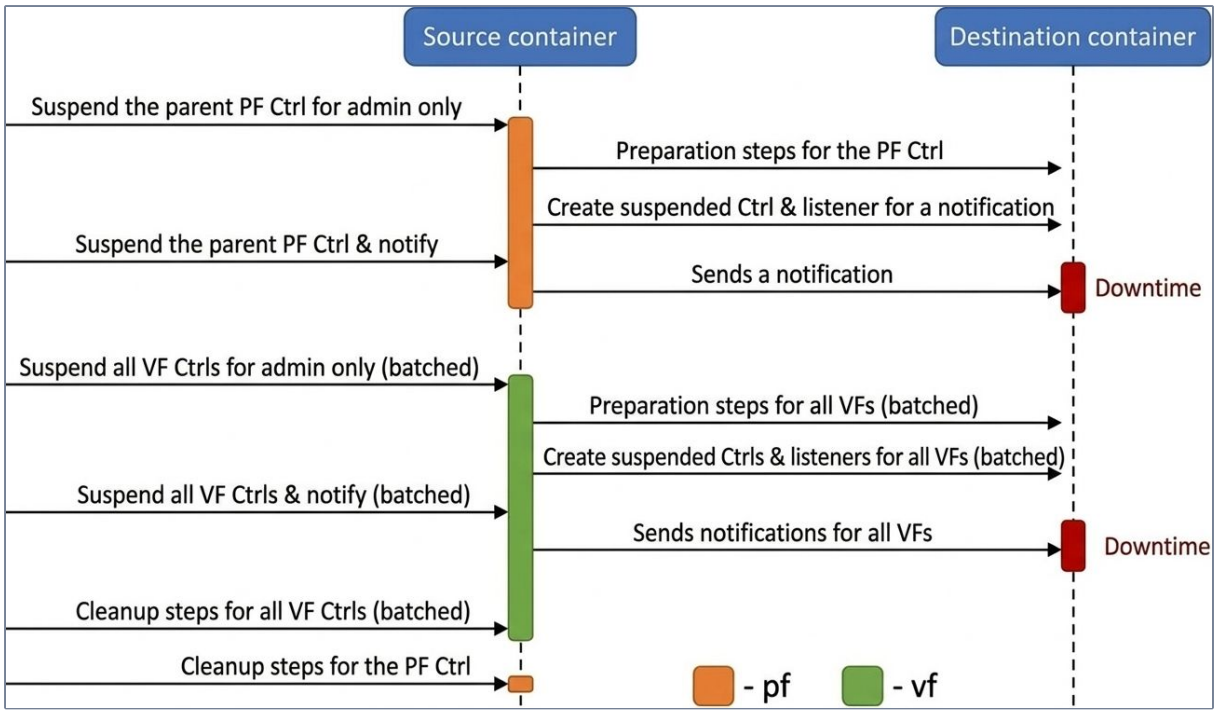
8.12.2.6.7.2 SNAP Configuration

1. Set the environment variable **VIRTIO_CTRL_VDPA_ADMIN_Q** to 1 (default) for legacy or 0 for standard VFIO mode.
2. Create a PF Controller with Admin Queue (common to both modes):

```
snap_rpc.py virtio_blk_controller_create --admin_q …
```

8.12.2.6.8 SNAP Container Live Upgrade

The Live Upgrade feature enables administrators to update a container's SNAP image without incurring system downtime.



Although newer releases may introduce features that alter system behavior, the upgrade architecture is engineered to maintain full backward and forward compatibility.

> **Version Compatibility**
>
> Before initiating an upgrade, consult the "Changes and New Features" section of the *SNAP-4 Service Release Notes* to verify the explicitly supported upgrade paths and compatibility matrices for your current version.

8.12.2.6.8.1 Live Upgrade Prerequisites

To enable live upgrade, perform the following modifications:

1. Allocate double hugepages for the destination and source containers.
2. Make sure the requested amount of CPU cores is available.
   The default YAML configuration sets the container to request a CPU core range of 8-16. This means that the container is not deployed if there are fewer than 8 available cores, and if there are 16 free cores, the container utilizes all 16.
   For instance, if a container is currently using all 16 cores and, during a live upgrade, an additional SNAP container is deployed. In this case, each container uses 8 cores during the upgrade process. Once the source container is terminated, the destination container starts utilizing all 16 cores.

   > For 8-core DPUs, the **.yaml** must be edited to the range of 4-8 CPU cores.

3. Change the name of the **doca_snap.yaml** file that describes the destination container (e.g., **doca_snap_new.yaml**) so as to not overwrite the running container **.yaml**.

4. Change the name of the new **.yaml** pod and container on lines 16 and 20, respectively (e.g., **snap-new**).
5. Deploy the the destination container by copying the new yaml (e.g., **doca_snap_new.yaml**) to kubelet.

> ⚠ After deploying the destination container, until the live update process is complete, avoid making any configuration changes via RPC. Specifically, do not create or destroy hotplug functions.

> ⚠ Live updates are exclusively supported between consecutive software releases. Skipping versions during a live update process is not supported and may result in system failure.

> ⚠ When restoring a controller in the destination container during a live update, it is recommended to use the same arguments originally used for controller creation in the source container.

> ⚠ User may need to update the RPC alias since the new container name has been edited.

> ⚠ Performing a live update disables the ML Optimizer Online service in the source container. The service in the destination container remains unaffected and operates normally.

> ⚠ Supported Block Devices for live_update.py
>
> Official support for the **live_update.py** script is strictly limited to the following SPDK block devices: NVMe-oF/RDMA, Null, Malloc, AIO and Delay **bdev**. Although users can technically extend the script logic to accommodate other block devices, NVIDIA does not officially validate or support such configurations.

8.12.2.6.8.2 SNAP Container Live Upgrade Procedure

1. Follow the steps in section "Live Upgrade Prerequisites" and deploy the destination SNAP container using the modified **yaml** file.
2. Query the source and destination containers:

```
crictl ps -r
```

3. Check for **SNAP started successfully** in the logs of the destination container, then copy the live update from the container to your environment.

```
[dpu] crictl logs -f <dest-container-id>
[dpu] crictl exec <dest-container-id> cp /opt/nvidia/nvda_snap/bin/live_update.py /etc/nvda_snap/
```

4. Run the **live_update.py** script to move all active objects from the source container to the destination container:

```
[dpu] cd /etc/nvda_snap
[dpu] ./live_update.py -s <source-container-id> -d <dest-container-id>
```

Transition freely between a SNAP source package service and a SNAP container service in either direction.

- Use the **-s 0** flag to specify that the original process (source) is running from the SNAP source package.
- The tool does not support transitioning directly from one SNAP source package service to another

Customize the suspend timeout by passing the **-t** or **--timeout** parameter (e.g., **-t 1** sets the timeout to 1 ms). The system defaults to 20 ms.

5. After the script completes, the live update process also completes, delete the source container by removing the YAML from kubelet tool:

To post RPCs, use the **crictl** tool:

```
crictl exec -it <container-id X> snap_rpc.py <RPC-method>
crictl exec -it <container-id Y> spdk_rpc.py <RPC-method>
```

To automate the SNAP configuration (e.g., following failure or reboot) as explained in section "Automate SNAP Configuration (Optional)", **spdk_rpc_init.conf** and **snap_rpc_init.conf** must not include any configs as part of the live upgrade. Then, once the transition to the new container is done, **spdk_rpc_init.conf** and **snap_rpc_init.conf** can be modified with the desired configuration.

Live Upgrade --skip-spdk (optional)

With --skip-spdk, the live update tool migrates only SNAP-layer objects. Users must carry over SPDK-layer state separately on the destination process before live upgrade.

Run the live upgrade **--skip-spdk** tool:
[dpu] ./live_update.py -s <source-container-id> -d <dest-container-id> --skip-spdk

SPDK attribute migration suggestion:

Complete these steps before running the live upgrade tool.

1. Run save_config SPDK RPC on the source container
   [dpu] crictl exec -i <source-container-id> bash -c 'spdk_rpc.py save_config > /etc/nvda_snap/config.json'
2. Start the destination container configured with the **SPDK_RPC_INIT_CONF_JSON** variable

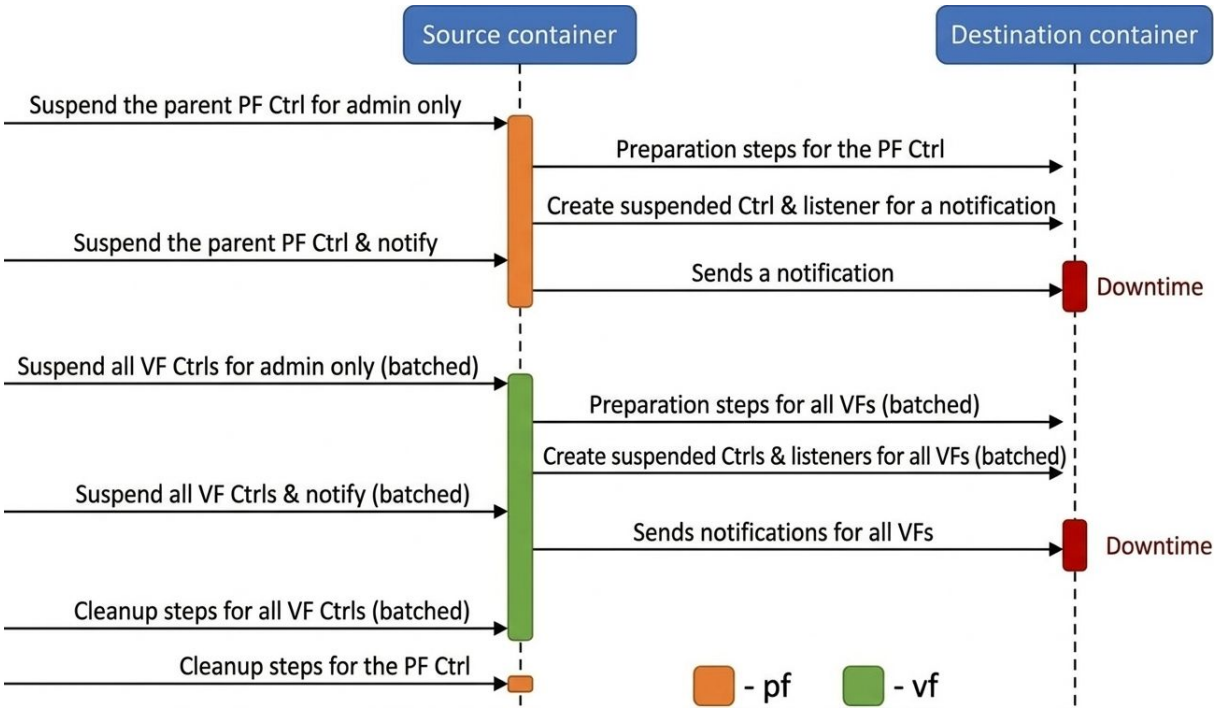
```
env:
  - name: SPDK_RPC_INIT_CONF_JSON
```

```
    value: "/etc/nvda_snap/config.json"
```

The **config.json** file must exist before the destination container starts. SPDK block devices are created during destination SNAP initialization.

8.12.2.6.8.3 SNAP Container Live Upgrade Commands

The live upgrade process allows you to move SNAP controllers and SPDK block devices between containers while minimizing host VM disruption. PFs are live-updated individually, while all VFs are batched together to reduce total downtime.



The upgrade is executed using a dedicated live update tool, which handles PF controllers sequentially and then processes all VF controllers in batched phases.

PF Live Update (Sequential, per PF)

1. Suspend controller (admin only): On the source container, suspend the controller in admin-only mode. This ensures the controller stops processing admin commands from the host driver, avoiding state changes during the handover. I/O traffic continues, meaning downtime has not yet started.
    - NVMe example:

      ```
      snap_rpc.py nvme_controller_suspend --ctrl NVMeCtrl0VF0 --admin_only
      ```

    - Virtio-blk example:

      ```
      snap_rpc.py virtio_blk_controller_suspend --ctrl VBLKCtrl0 --events_only
      ```

2. Prepare destination container: On the destination container, create all required objects for the new controller, including attaching the backend device.
    - NVMe example:

```
spdk_rpc.py bdev_nvme_attach_controller ...
snap_rpc.py nvme_subsystem_create ...
snap_rpc.py nvme_namespace_create -n 1 ...
```

    - Virtio-blk example:

```
spdk_rpc.py bdev_nvme_attach_controller ...
```

3. Create suspended controller (as listener): On the destination container, create the controller in a suspended state and designate it as a listener for a live update notification from the source container. The controller in the source container is still handling I/O, meaning downtime has not yet started.
    - NVMe example:

```
snap_rpc.py nvme_controller_create --pf_id 0 --vf_id 0 --ctrl NVMeCtrl0VF0 --live_update_listener
--suspended ...
snap_rpc.py nvme_controller_attach_ns -c NVMeCtrl0VF0 -n 1
```

    - Virtio-blk example:

```
snap_rpc.py virtio_blk_controller_create  --pf_id 0 --vf_id 0 --ctrl VBLKCtrl0VF0 --suspended --
live_update_listener ...
```

4. Suspend and notify: On the source container, suspend the controller using the `--live_update_notifier` flag. This triggers the start of downtime and sends a notification to the destination container. Once suspended, the controller on the destination container resumes and starts handling I/O, marking the end of downtime.
    - NVMe example:

```
snap_rpc.py nvme_controller_suspend --ctrl NVMeCtrl0VF0 --live_update_notifier --timeout_ms 20
```

    - Virtio-blk example:

```
snap_rpc.py virtio_blk_controller_suspend --ctrl VBLKCtrl0 -t 20 -lu
```

5. Cleanup source PF: The PF controller on the source is destroyed only after all of its related VFs have been migrated and cleaned up.

    ⚠ The PF controller must remain present in the source container until all related VFs have been successfully removed.

VF Live Update (Batched, All VFs Together)

After each PF is live-updated, all active VFs are collected and processed together in batched phases.

1. NVMe only – Prepare destination for all NVMe VFs (batched): On the destination container, create all required bdevs, subsystems, and namespaces.
    - NVMe example:

```
spdk_rpc.py bdev_nvme_attach_controller ...    # for VF0
spdk_rpc.py bdev_nvme_attach_controller ...    # for VF1
snap_rpc.py nvme_subsystem_create ...           # for VF0
snap_rpc.py nvme_namespace_create ...           # for VF0
snap_rpc.py nvme_subsystem_create ...           # for VF1
snap_rpc.py nvme_namespace_create ...           # for VF1
...
```

2. Suspend all VF controllers on source (admin only): On the source container, suspend all VF controllers using a single batched command. I/O traffic continues for all VFs at this point, meaning downtime has not yet started.
    - NVMe example:

```
snap_rpc.py nvme_controller_suspend --ctrl NVMeCtrl0VF0 --admin_only
snap_rpc.py nvme_controller_suspend --ctrl NVMeCtrl0VF1 --admin_only
...
```

    - Virtio-blk example:

```
snap_rpc.py virtio_blk_controller_suspend --ctrl VBLKCtrl0VF0 --events_only
snap_rpc.py virtio_blk_controller_suspend --ctrl VBLKCtrl0VF1 --events_only
...
```

3. Prepare destination for all VFs (batched): On the destination container, create all required backend devices (for virtio-blk) and controllers (suspended, with live update listener) for every VF using a single batched command.
    - NVMe example:

```
snap_rpc.py nvme_controller_create --pf_id 0 --vf_id 0 --ctrl NVMeCtrl0VF0 --live_update_listener
--suspended ...
snap_rpc.py nvme_controller_create --pf_id 0 --vf_id 1 --ctrl NVMeCtrl0VF1 --live_update_listener
--suspended ...
...
```

    - Virtio-blk example:

```
spdk_rpc.py bdev_nvme_attach_controller ...    # for VF0
spdk_rpc.py bdev_nvme_attach_controller ...    # for VF1
snap_rpc.py virtio_blk_controller_create --pf_id 0 --vf_id 0 --ctrl VBLKCtrl0VF0 --suspended --
live_update_listener ...
snap_rpc.py virtio_blk_controller_create --pf_id 0 --vf_id 1 --ctrl VBLKCtrl0VF1 --suspended --
live_update_listener ...
...
```

4. Suspend and notify all VFs (batched): On the source container, suspend all VF controllers with the live update notifier flag using a single batched command. This initiates downtime for all VFs. The destination controllers receive the notifications and resume handling I/O, marking the end of downtime.
    - NVMe example:

```
snap_rpc.py nvme_controller_suspend --ctrl NVMeCtrl0VF0 --live_update_notifier --timeout_ms 20
snap_rpc.py nvme_controller_suspend --ctrl NVMeCtrl0VF1 --live_update_notifier --timeout_ms 20
...
```

    - Virtio-blk example:

```
snap_rpc.py virtio_blk_controller_suspend --ctrl VBLKCtrl0VF0 --live_update_notifier --timeout_ms
20
snap_rpc.py virtio_blk_controller_suspend --ctrl VBLKCtrl0VF1 --live_update_notifier --timeout_ms
20
...
```

5. Cleanup all VFs on source (batched): After the migration is complete, clean up all VF controller objects and backend devices on the source container using a single batched command.

- NVMe example:

```
snap_rpc.py nvme_controller_detach_ns ...   # for each VF
snap_rpc.py nvme_controller_destroy ...     # for each VF
spdk_rpc.py bdev_nvme_detach_controller ... # for each VF's bdev
snap_rpc.py nvme_namespace_destroy ...      # for each VF
```

- Virtio-blk example:

```
snap_rpc.py virtio_blk_controller_destroy --ctrl VBLKCtrl0VF0
snap_rpc.py virtio_blk_controller_destroy --ctrl VBLKCtrl0VF1
spdk_rpc.py bdev_nvme_detach_controller ...  # for each VF's bdev
```

6. Cleanup PF on source: Finally, destroy the PF controller on the source now that all of its VFs have been removed.
   - NVMe example:

```
snap_rpc.py nvme_controller_detach_ns ...
snap_rpc.py nvme_namespace_destroy ...
spdk_rpc.py bdev_nvme_detach_controller ...
snap_rpc.py nvme_controller_destroy ...
```

   - Virtio-blk example:

```
snap_rpc.py virtio_blk_controller_destroy --ctrl VBLKCtrl0
spdk_rpc.py bdev_nvme_detach_controller ...
```

8.12.2.6.8.4 Shared Memory Pool Live Update

The Shared Memory Pool Live Update feature resolves resource constraints when a host lacks sufficient hugepages to run two full SNAP processes concurrently during a live update. By allowing the two processes to share a portion of the memory pool, this optimization significantly reduces the peak hugepage requirement.

Configuration	Total Hugepages Required	Memory Breakdown
Standard	4GB	2GB per process (1GB SNAP + 1GB SPDK)
Optimized	3GB	1GB Source SPDK + 1GB Destination SPDK + 1GB Shared SNAP

Although physical hugepage consumption decreases to 3GB, container orchestration layers (such as Kubernetes) may still require the memory limit set to the full 4GB to authorize the transition. The host does not need this additional 1GB physically free, and the update process will not actually consume it.

Memory Pool Architecture

To facilitate shared memory, SNAP divides its memory management into two distinct pools:

- Base mempool – The essential memory required for the process to operate.
- Extended mempool – Additional memory used for standard operation, managed dynamically during the update.

Live Update Configuration

Enable this feature by configuring specific environment variables in the **doca_vfs.yaml** file. The **SNAP_MEMPOOL_SIZE_MB** value must strictly exceed the **SNAP_MEMPOOL_BASE_SIZE_MB** value.

Configuration example:

```
env:
  - name: SNAP_MEMPOOL_SIZE_MB
    value: "1024"
  - name: SNAP_MEMPOOL_BASE_SIZE_MB
    value: "512"
```

Live Update Procedure

This optimized workflow introduces two specific RPCs: **memory_manager_deallocate_basepool** and **memory_manager_allocate_extpool**.

Execute the live update using the following strict sequence:

1. Prepare source – Issue the **memory_manager_deallocate_basepool** RPC on the source (old) container to release shared resources.
2. Deploy destination – Start the destination (new) container. It initializes using only the Base Mempool.
3. Execute handover – Run the standard live update script and wait for completion.
4. Cleanup – Destroy the source container.
5. Finalize destination – Issue the **memory_manager_allocate_extpool** RPC on the destination container to reclaim full memory capacity.

State Invalidation

If the source container's SNAP instance terminates, restarts, or recovers at any point during this process, the state becomes invalid. Completely restart the service before reattempting the live update procedure.

8.12.2.6.9 SR-IOV Dynamic MSI-X and Queues Management

Message Signaled Interrupts eXtended (MSI-X) enables devices to utilize multiple interrupt vectors, reducing CPU utilization, improving device performance, and enhancing scalability compared to traditional shared interrupts.

In SR-IOV environments, the default system behavior distributes MSI-X vectors and queues evenly across all Virtual Functions (VFs). However, this uniform distribution is often suboptimal for heterogeneous environments where VFs attach to different VMs with varying resource requirements. Dynamic MSI-X and queues Management allows administrators to manually override this default and explicitly control the number of MSI-X vectors and queues allocated to each individual VF.

The configuration steps and behaviors described in this section apply to all emulation types (specifically NVMe and virtio-blk).

8.12.2.6.9.1 Lifecycle and Persistence

Dynamic MSI-X and queues management follows a strict lifecycle for resource allocation and reclamation.

Allocation Workflow

1. Reclaim: When no VF controllers are open (**sriov_numvfs=0**), PF-related MSI-X vectors and queues return from the VFs into the PF's global free pool.
2. Allocate: Administrators allocate MSI-X vectors and queues from the free pool to a specific VF during controller creation.
3. Release: Resources return back to the free pool when a VF controller is destroyed.

Persistence Rules

Once configured, the MSI-X allocation and queues for a VF remains persistent across various state changes:

State Change	Effect on MSI-X Configuration
Application restart/crash	No change
Closing/reopening PF	No change (unless dynamic support is used)
Explicit VF release	Released (returns to pool)
PF explicit reclaim	Reclaimed (returns to pool)
Arm cold boot	Reset (returns to pool)

8.12.2.6.9.2 Configuration Procedure (NVMe Example)

The following procedure demonstrates Dynamic MSI-X and queues configuration for an NVMe controller. Apply the same logic when configuring virtio-blk.

Step 1: Reclaim Resources

Ensure no VFs are active, then reclaim all MSI-X vectors and queues back to the PF's free pool:

```
snap_rpc.py nvme_controller_vfs_dynamic_resources_reclaim -c <CtrlName>
```

Step 2: Query Resource Constraints

Query the controller to view the available resources currently residing in the PF's free pool:

```
snap_rpc.py nvme_controller_list -c <CtrlName>
```

Output definitions:

- **free_msix** – Total MSI-X vectors available in the PF pool (assigned via **vf_num_msix**).
- **free_queues** – Total queues (doorbells) available in the PF pool (assigned via **num_queues**).
- **vf_min_msix** / **vf_max_msix** – The valid configuration range for the **vf_num_msix** parameter.

- **vf_min_queues** / **vf_max_queues** – The valid configuration range for the **num_queues** parameter.

Step 3: Create VF and Distribute Resources

Create the VF controller and specify the exact resource allocation:

```
snap_rpc.py nvme_controller_create --vf_num_msix <n> --num_queues <m> ...
```

You must explicitly specify both **vf_num_msix** and **num_queues**. Omitting one can cause a mismatch between the MSI-X allocation and queue configuration, potentially leading to driver malfunctions.

Resource Calculation Guidelines

Resource allocations differ based on the protocol. As a baseline, use the formula **num_queues = vf_num_msix - 1** and apply the following rules to determine your exact values:

Protocol	Allocation Formula
NVMe	Allocate 1 MSI-X vector per I/O queue + 1 MSI-X vector for the Admin queue
Virtio	Allocate 1 MSI-X vector per virtqueue + 1 MSI-X vector for BAR configuration notifications

Step 4: Resource Modification (Optional)

Resources allocated to a VF can be dynamically modified as needed:

```
snap_rpc.py nvme_controller_modify -c <CtrlName> --num_msix X --num_queues Y
```

Step 5: VF Teardown

When destroying the VF, ensure resources are explicitly released back to the free pool:

```
snap_rpc.py nvme_controller_destroy -c <CtrlName> --release_msix --release_queues ...
```

Step 6: Enable SR-IOV

Enable the VFs on the host driver:

```
echo <N> > /sys/bus/pci/devices/<BDF>/sriov_numvfs
```

8.12.2.6.9.3 Safe Configuration Methods

Strict ordering of operations is required to prevent system instability during SR-IOV configuration.

Host Deadlock Risk

Open all VF controllers in SNAP before binding VFs to the host driver. If VFs bind to the driver before configuration completes, the driver may attempt to use unallocated resources. Insufficient resources in this state can trigger a host deadlock that is recoverable only via a cold boot.

To configure Dynamic MSI-X safely and avoid deadlock, utilize one of the following two methods:

Method A: Disable Autoprobe (Recommended)

Disable automatic driver binding, complete the VF configuration, and manually bind them afterward:

1. Disable autoprobe to prevent immediate binding:

```
echo 0 > /sys/bus/pci/devices/<BDF>/sriov_driver_autoprobe
```

2. Perform SNAP Configuration (Execute Steps 1-5 from above).
3. Manually bind VFs to the driver:

```
echo "0000:01:00.0" > /sys/bus/pci/drivers/nvme/bind
```

Method B: Use VFIO Driver

Utilize the **vfio-pci** driver for SR-IOV configuration instead of the standard kernel driver.

1. Bind PF to the VFIO driver:

```
echo 0000:af:00.2 > /sys/bus/pci/drivers/vfio-pci/bind
```

2. Enable SR-IOV support:

```
echo 1 > /sys/module/vfio_pci/parameters/enable_sriov
```

3. Create VFs:

```
echo <N> > /sys/bus/pci/drivers/vfio-pci/0000:af:00.2/sriov_numvfs
```

8.12.2.6.10 Recovery

The recovery feature enables the restoration of controller state after the SNAP application terminates—either gracefully or unexpectedly (e.g., due to **kill -9**).

Recovery is only possible if the SNAP application is restarted with the exact same configuration that was active prior to the shutdown or crash.

Supported Block Devices for live_update.py

Official support for the **live_update.py** script is strictly limited to the following SPDK block devices: NVMe-oF/RDMA, Null, Malloc, and Delay **vbdev**. Although users can technically extend the script logic to accommodate other block devices, NVIDIA does not officially validate or support such configurations.

8.12.2.6.10.1 NVMe Recovery

NVMe recovery enables the restoration of an NVMe controller after a SNAP application terminates, whether gracefully or due to a crash (e.g., **kill -9**).

To perform NVMe recovery:

1. Re-create the controller in a suspended state using the exact same configuration as before the crash (including the same bdevs, number of queues, namespaces, and namespace UUIDs).
2. Resume the controller only after all namespaces have been attached.

The recovery process uses shared memory files located under **/dev/shm** on the BlueField to restore the controller's internal state. These files are deleted when the BlueField is reset, meaning recovery is not supported after a BF reset.

8.12.2.6.10.2 Virtio-blk Crash Recovery

To use virtio-blk recovery, the controller must be re-created with the same configuration as before the crash (i.e. the same bdevs, num queues, etc).

The following options are available to enable virtio-blk crash recovery.

Virtio-blk Crash Recovery with --force_in_order

For virtio-blk crash recovery with **--force_in_order**, disable the **VBLK_RECOVERY_SHM** environment variable and create a controller with the **--force_in_order** argument.

In virtio-blk SNAP, the application is not guaranteed to recover correctly after a sudden crash (e.g., **kill -9**).

To enable the virtio-blk crash recovery, set the following:

```
snap_rpc.py virtio_blk_controller_create --force_in_order …
```

Setting **force_in_order** to 1 may impact virtio-blk performance as it will serve the command in-order.

If **--force_in_order** is not used, any failure or unexpected teardown in SNAP or the driver may result in anomalous behavior because of limited support in the Linux kernel virtio-blk driver.

Virtio-blk Crash Recovery without --force_in_order

For virtio-blk crash recovery without `--force_in_order`, enable the `VBLK_RECOVERY_SHM` environment variable and create a controller without the `--force_in_order` argument.

Virtio-blk recovery allows the virtio-blk controller to be recovered after a SNAP application is closed whether gracefully or after a crash (e.g., `kill -9`).

To use virtio-blk recovery without `--force_in_order` flag. `VBLK_RECOVERY_SHM` must be enabled, the controller must be recreated with the same configuration as before the crash (i.e., same bdevs, num queues, etc).

When `VBLK_RECOVERY_SHM` is enabled, virtio-blk recovery uses files on the BlueField under `/dev/shm` to recover the internal state of the controller. Shared memory files are deleted when the BlueField is reset. For this reason, recovery is not supported after BlueField reset.

8.12.2.6.10.3 SNAP Configuration Recovery

SNAP can store its configuration as defined by user RPCs and, upon restart, reload it from a configuration JSON file. This mechanism is intended for recovering a previously configured SNAP state - it cannot be used for the initial configuration.

Usage:

1. Set the environment variable `SNAP_RPC_INIT_CONF_JSON` to the directory path where the configuration file will be stored.
2. The configuration file, `snap_config.json`, is created in this directory after all changes in your script have been successfully applied.
3. If a new configuration (different from the pre-shutdown configuration) is required after restarting SNAP, delete the existing `snap_config.json` file before applying the new settings.

When this method is used, there is no need to re-run `snap` RPCs or set RPCs in init files after the initial configuration — SNAP will automatically load the saved configuration from the `SNAP_RPC_INIT_CONF_JSON` path. This approach is recommended for fast recovery.

- When modifying controller or function configurations, ensure the controller/function is not bounded to any driver until the configuration process is complete. If the change is interrupted, recovery may fail.
- Hotplugged emulation functions persist between SNAP runs (but not across BlueField resets) and should be set only once during initial configuration. Only controllers created on these functions are stored in the saved configuration state.
- If crash recovery after a reboot is supported, store the file inside the container at `/etc/nvda_snap`. For unsupported use cases, store it in a temporary location such as `/tmp/` or `/dev/shm`.

8.12.2.6.10.4 Improving SNAP Recovery Time

The following table outlines features designed to accelerate SNAP initialization and recovery processes following termination.

Feature	Description	How to?
SPDK JSON-RPC configuration file	An initial configuration can be specified for the SPDK configuration in SNAP. The configuration file is a JSON file containing all the SPDK JSON-RPC method invocations necessary for the desired configuration. Moving from posting RPCs to JSON file improves bring-up time. For more information check SPDK JSON-RPC documentation.	To generate a JSON-RPC file based on the current configuration, run: `spdk_rpc.py save_config > config.json` The **config.json** file can then be passed to a new SNAP deployment using the environment variable in the YAML **SPDK_RPC_INIT_CONF_JSON**. If SPDK encounters an error while processing the JSON configuration file, the initialization phase fails, causing SNAP to exit with an error code.

Disable SPDK accel functionality	The SPDK accel functionality is necessary when using NVMe TCP features. If NVMe TCP is not used, accel should be manually disabled to reduce the SPDK startup time, which can otherwise take few seconds. To disable all accel functionality edit the flags **disable_signature**, **disable_crypto**, **enable_driver** and **enable_module**. And remove accel_set_driver.	Remove this from config file:
		`"method": "accel_set_driver", "params": { "name": "mlx5" }`
		And edit the config file as follows:
		`{ "method": "mlx5_scan_accel_module", "params": { "qp_size": 64, "cq_size": 1024, "num_requests": 2048, "enable_driver": false, "split_mb_blocks": 0, "siglast": false, "qp_per_domain": false, "disable_signature": true, "disable_crypto": true, "enable_module": false } }`

Provide the emulation manager name	If the **SNAP_EMULATION_MANAGER** environment variable is not defined (default), SNAP searches through all available devices to find the emulation manager which may slow down initialization process. Explicitly defining the device reduces the chance of initialization delays.	Use **SNAP_EMULATION_MANAGER** to modify the the variable on the YAML. Refer to SNAP-4 Service Environment Variables for more information.
SNAP configuration recovery	SNAP configuration recovery enables restoring the SNAP state without the need to re-post SNAP RPCs. By moving from posting individual RPCs to using a pre-saved JSON configuration file, the bring-up time is significantly improved.	Set **SNAP_RPC_INIT_CONF_JSON** to a path where config file should be saved. Also use an SPDK JSON-RPC configuration file.
Hugepages allocation	SNAP allocates a mempool from hugepages. Reducing its size can impact the duration of SNAP's crash recovery.	SNAP_MEMPOOL_SIZE_MB is set to1024MB by default.

8.12.2.6.11 Watchdog and Heartbeat Monitoring

The Watchdog and Heartbeat Monitoring feature is an experimental reliability mechanism designed to enhance the robustness of the system by automatically detecting and recovering from application hangs, crashes, or unresponsive components. This mechanism minimizes service disruption by triggering recovery procedures without requiring manual intervention.

The heartbeat system functions as a periodic signal emitted by the SNAP service to indicate its operational status. These signals serve as an indicator that the service is active and functioning as expected.

A dedicated watchdog component monitors the presence and frequency of heartbeat signals. If the heartbeat is not received within a predefined timeout interval, the watchdog determines that the monitored component is unresponsive and initiates a predefined recovery action.

The typical sequence of operations is as follows:

1. A SNAP component becomes unresponsive due to a crash, hang, or other failure condition.
2. The watchdog detects the absence of the expected heartbeat signal.
3. A recovery action is automatically triggered.
4. The SNAP service is restarted, and previously configured virtual disk states are restored.
5. Normal operation resumes.

The entire recovery process is designed to complete within a few seconds, thereby minimizing downtime.

8.12.2.6.11.1 Configuring the Watchdog

The behavior of the Watchdog and Heartbeat Monitoring system is configurable through environment variables. These variables allow the user to specify parameters such as heartbeat intervals, timeouts, and recovery policies without requiring changes to the application code.

Environment Variable	Impact on Recovery	Default Value
`SNAP_HEARTBEAT_INTERVAL_MS`	Interval (in milliseconds) between heartbeat signals from SNAP	`0` (disabled)
`SNAP_HEARTBEAT_THREAD_ID`	ID of the thread responsible for processing heartbeat signals	`1`

For more configuration options, check the `snap_watchdog.py` script.

8.12.2.6.11.2 Running the Watchdog

To initiate the watchdog service while the SNAP application is running, execute the following command:

```
./snap_watchdog.py --daemon
```

This command launches the watchdog in the background, where it continuously monitors the health of the SNAP service and initiates recovery procedures as necessary.

> ⚠ snap_watchdog.py requires using python 3.7 or above.
> If SNAP is running in a BFB that only contains an older version of python (i.e. Anolis 8), user must also run the command:
>
> ```
> pip install dataclasses
> ```

8.12.2.6.12 IO/Core Multiplexer

The I/O Core Multiplexer (MP) is a configurable mechanism that determines how I/O requests from a single source are distributed across the available DPU cores. This feature is critical for optimizing performance based on application-specific needs, particularly in scenarios involving high I/O workloads.

The multiplexer offers two policy modes:

- None (Default) – All I/O operations originating from a single source are processed by a single DPU core. I/O sources are distributed across DPU cores in a balanced manner.
  - Recommended for: Low-latency environments
  - Optimization focus: I/O latency
- (Weighted) Round Robin – I/O requests from a single source are distributed across multiple DPU cores in a round-robin sequence. If the backend supports per-core weight configuration (e.g., SPDK NVMe-oF bdev), the distribution follows those weights. Otherwise, the I/Os are spread evenly.
  - Recommended for: Bandwidth-intensive environments or systems with low per-core backend throughput (e.g., TCP-based backends)
  - Optimization focus: I/O bandwidth

To configure the IO/Core Multiplexer policy, users need to set the **IO_CORE_DISTRIBUTION_POLICY** environment variable. The available options are:

- **none** – Refers to the default policy where all I/Os from a single source are handled by a single DPU core
- **weighted_rr** – Refers to the (Weighted) Round Robin policy, distributing I/Os across multiple cores

  > ⚠ **weighted_rr** policy is not supported for virtio-blk.

8.12.2.6.13 DPA Execution Unit Management

The DPA is an auxiliary processor designed to accelerate data-path operations. Its architecture comprises a cluster of 16 cores, each containing 16 execution units (EUs).

- Total SNAP capacity – 171 EUs (index range: 0–170).
- Supported protocols – SNAP utilizes DPA applications to accelerate NVMe and virtio-blk protocols.

- Hardware constraint – The hardware enforces a strict limit of 128 queues (threads) per DPA EU.

By default, the system shares all EUs (0–170) across NVMe, virtio-blk, and other system DPA applications (such as virtio-net).

> **Resource Conflict**
>
> If other DPA applications run concurrently with SNAP, configure a DPA resource YAML file to explicitly allocate EUs and prevent resource overlaps. For further details, consult the *Single Point of Resource Distribution* documentation

8.12.2.6.13.1 Method 1: YAML-based Resource Management (Recommended)

The YAML-based tool serves as the primary method for controlling DPA EUs, providing a centralized and consistent approach to allocating resources across applications.

Configuration Requirements

Requirement	Details
Partitioning	Run SNAP DPA applications exclusively on the `ROOT` partition. EUs configured in other partitions are inaccessible.
Valid range	Allocate only EUs 0–170 for SNAP operations
Allocation rules	• At least 1 EU must be allocated per application instance. • EU allocations must not overlap across different applications. • EU groups are not supported.
Application names	Match the YAML configuration to SNAP's internal application names exactly: • `dpa_helper` : Used for virtio-blk and NVMe (DPU mode) ⚠ Match instances to the number of allocated Arm cores/EUs. • `dpa_virtq_split` : Used for virtio-blk (DPA mode) • `dpa_nvme` : Used for NVMe (DPA mode)

Multi-Container Configuration

When deploying multiple SNAP containers, enforce unique application naming:

1. Set the environment variable `SNAP_DPA_INSTANCE_ID_ENV` to a unique ID inside each container.
2. Update the YAML file to reflect the instance names using the `<APP_NAME>_<ID>` format.

Example of two virtio-blk DPU containers:

```
---
version: 25.04
---
DPA_APPS:
  dpa_helper_1:
    - partition: ROOT
      affinity_EUs: [1-16]
  dpa_helper_2:
    - partition: ROOT
```

```
    affinity_EUs: [17-32]
```

Deployment Workflow

1. Define your configuration in a source file (e.g., **~/DPA_RESOURCE_INPUT.yaml**).
2. Execute the management tool to generate the final system configuration:

```
dpa-resource-mgmt config -d mlx5_0 -f ~/DPA_RESOURCE_INPUT.yaml
```

> Never manually edit the output YAML file generated by the **dpa-resource-mgmt** tool.

3. Point SNAP to the generated configuration file:

```
export SNAP_DPA_YAML_PATH=~/ROOT.YAML
```

> **Container Implementations**
>
> Ensure the generated YAML file path is properly mounted into the container, such as mapped to **/etc/nvda_snap**).

Default Configuration Reference

For reference, the standard default configuration is as follows:

```
---
version: 25.04
---
DPA_APPS:
  dpa_helper:
    - partition: ROOT
      affinity_EUs: [1-16]
  dpa_virtq_split:
    - partition: ROOT
      affinity_EUs: [0-169]
  dpa_nvme:
    - partition: ROOT
      affinity_EUs: [0-169]
```

8.12.2.6.13.2 Method 2: Core Mask (Specific Use Case)

The Core Mask method provides an alternative approach for EU allocation.

> Use Case Restriction
>
> This method is the default, preconfigured approach strictly for running SNAP-4 concurrently with the SNAP virtio-fs service. For all other scenarios requiring DPA EU modifications, utilize the YAML-based method described above.

Configuration

To assign a specific set of EUs via a mask, set the corresponding environment variable using a hexadecimal string.

Application	Environment Variable
Virtio-blk / NVMe (DPU Mode)	`dpa_helper_core_mask=0x<EU_mask>`
NVMe (DPA Mode)	`dpa_nvme_core_mask=0x<EU_mask>`
Virtio-blk (DPA Mode)	`dpa_virtq_split_core_mask=0x<EU_mask>`

Mask Logic

The system parses the core mask from right to left, utilizing valid hexadecimal digits.

Example: `dpa_virtq_split_core_mask=0xff00`

- The hexadecimal `ff00` sets 8 bits high (specifically bits 8–15).
- Result: The system allocates 8 EUs (EUs 8 through 15) to the virtio-blk application.

8.12.2.6.14 SNAP ML Optimizer Online

The SNAP ML Online Optimizer is a dynamic performance-tuning utility that continuously adjusts polling parameters within the SNAP I/O subsystem. It maximizes controller throughput by identifying the optimal configuration based on real-time hardware conditions, workload patterns, and system constraints.

8.12.2.6.14.1 Operational Workflow

The optimizer utilizes predictive modeling to converge on an optimal setup efficiently, eliminating the need to exhaustively test all possible parameter combinations. It operates using the following iterative loop:

1. Execute actions: Iteratively modifies internal configuration parameters (actions) during runtime.
2. Measure rewards: Measures the resulting system performance (reward) after each configuration change. Currently, the optimizer explicitly targets and maximizes IOPS as the primary reward metric.
3. Predict next state: Uses predictive models to determine the most promising configuration to evaluate next.

8.12.2.6.14.2 Background Optimization Behavior

The SNAP ML Optimizer continuously analyzes active workloads and applies optimization actions in the background. It dynamically learns workload profiles and adapts rapidly when it recognizes returning traffic patterns.

Once the optimizer characterizes a workload and applies the optimal parameters, it transitions into an idle state. It resumes active tuning only when it detects a significant change in the underlying traffic pattern.

Configuring SNAP ML Online Optimizer

The ML Optimizer can be enabled using one of two methods:

1. Environment variable – Set `SNAP_ML_OPTIMIZER_ENABLED=1` when launching SNAP.

2. Runtime RPC – Execute the `snap_ml_optimizer_create` RPC.

ML Optimizer RPCs

The following Remote Procedure Calls (RPCs) are used to manage the optimizer at runtime.

snap_ml_optimizer_create

The `snap_ml_optimizer_create` initializes or recreates the ML Optimizer, which will immediately begin analyzing and optimizing the system.

> ⚠ Enabling the ML Optimizer will override any system parameters previously applied via `snap_actions_set`, environment variables, or default settings.

snap_ml_optimizer_destroy

The `snap_ml_optimizer_destroy` command tops the ML Optimizer service.

- Once stopped, the system restores the default SNAP parameters.
- All workload data collected by the optimizer is discarded. Any future instances of the optimizer will begin the learning process from scratch.

snap_ml_optimizer_is_running

The `snap_ml_optimizer_is_running` command queries the current operational status of the ML Optimizer.

snap_ml_optimizer_current_parameters

The `snap_ml_optimizer_current_parameters` command retrieves the currently active internal parameters. This command returns the active configuration regardless of whether it was applied by the ML Optimizer or manually configured by the user.

snap_actions_set

The `snap_actions_set` command is used to dynamically adjust SNAP parameters (known as "actions") that control polling behavior. This command is a core feature of SNAP-AI tools, enabling both automated optimization for specific environments and workloads, as well as manual adjustment of polling parameters.

Command parameters:

Parameter	Mandatory?	Type	Description
`poll_size`	No	Number	Maximum number of IOs SNAP passes in a single polling cycle (integer; 1-256)
`poll_ratio`	No	Number	The rate in which SNAP poll cycles occur (float; 0< `poll_ratio` ≤1)
`max_inflights`	No	Number	Maximum number of in-flight IOs per core (integer; 1-65535)
`max_iog_batch`	No	Number	Maximum fairness batch size (integer; 1-4096)

Parameter	Mandatory?	Type	Description
max_new_ios	No	Number	Maximum number of new IOs to handle in a single poll cycle (integer; 1-4096)

> **Warning:** snap_actions_set cannot be used when ML Optimizer Online is enabled

snap_reward_get

The snap_reward_get command retrieves performance counters, specifically completion counters (or "reward"), which are used by the optimizer to monitor and enhance SNAP performance.

No parameters are required for this command.

snap_observations_get

The snap_observations_get returns the current state of SNAP as a set of predefined parameters.

No parameters are required for this command.

8.12.2.6.15 SNAP ML Optimizer Offline (DEPRECATED)

8.12.2.6.15.1 SNAP ML Optimizer Preparation Steps

Machine Requirements

The device should be able to SSH to the BlueField:

- Python 3.10 or above
- At least 6 GB of free storage

Setting Up SNAP ML Optimizer

To set up the SNAP ML optimizer:

1. Copy the snap_ml folder from the container to the shared nvda_snap folder and then to the requested machine:

   ```
   crictl exec -it $(crictl ps -s running -q --name snap) cp -r /opt/nvidia/nvda_snap/bin/snap_ml /etc/
   nvda_snap/
   ```

2. Change directory to the snap_ml folder:

   ```
   cd tools/snap_ml
   ```

3. Create a virtual environment for the SNAP ML optimizer.

   ```
   python3 -m venv snap_ml
   ```

   This ensures that the required dependencies are installed in an isolated environment.

4. Activate the virtual environment to start working within this isolated environment:

```
source snap_ml/bin/activate
```

5. Install the Python package requirements:

```
pip3 install --no-cache-dir -r requirements.txt
```

This may take some time depending on your system's performance.

6. Run the SNAP ML Optimizer.

```
python3 snap_ml.py --help
```

Use the **--help** flag to see the available options and usage information:

```
--version Show the version and exit.
-f, --framework <TEXT> Name of framework (Recommended: ax , supported: ax, pybo).
-t, --total-trials <INTEGER> Number of optimization iterations. The recommended range is 25-60.
--filename <TEXT> where to save the results (default: last_opt.json).
--remote <TEXT> connect remotely to the BlueField card, format: <bf_name>:<username>:<password>
--snap-rpc-path <TEXT> Snap RPC prefix (default: container path).
--log-level <TEXT> CRITICAL | ERROR | WARN | WARNING | INFO | DEBUG
--log-dir <TEXT> where to save the logs.
```

8.12.2.6.15.2 Run the Offline ML Optimizer

To optimize SNAP's parameters for your environment, use the following command:

```
python3 snap_ml.py --framework ax --total-trials 40 --filename example.json --remote
<bf_hostname>:<username>:<password> --log-dir <log_directory>
```

Results and Post-optimization Actions

Once the optimization process is complete, the tool automatically applies the optimized parameters. These parameters are also saved in a **example.json** file in the following format:

```
{
    "poll_size": 30,
    "poll_ratio": 0.6847347955107689,
    "max_inflights": 32768,
    "max_iog_batch": 512,
    "max_new_ios": 32
}
```

Additionally, the tool documents all iterations, including the actions taken and the rewards received, in a timestamped file named **example_<timestamp>.json**.

Applying Optimized Parameters Manually

Users can apply the optimized parameters on fresh instances of SNAP service by explicitly calling the **snap_actions_set** RPC with the optimized parameters as follows:

```
snap_rpc.py snap_actions_set -poll_size 30 -poll_ratio 0.6847 --max_inflights 32768 -max_iog_batch 512 -max_new_ios
32
```

It is only recommended to use the optimized parameters if the system is expected to behave similarly to the system on which the SNAP ML optimizer is used.

Deactivating Python Environment

Once users are done using the SNAP ML Optimizer, they can deactivate the Python virtual environment by running:

```
deactivate
```

8.12.2.6.16 Plugins

Plugins are modular components or add-ons that enhance the functionality of the SNAP application. They integrate seamlessly with the main software, allowing additional features without requiring changes to the core codebase. Plugins are designed for use only with the source package, as it allows customization during the build process, such as enabling or disabling plugins as needed.

In containerized environments, the SNAP application is shipped as a pre-built binary with a fixed configuration. Since the binary in the container is precompiled, adding or removing plugins is not possible. The containerized software only supports the plugins included during its build. For environments requiring plugin flexibility, such as adding custom plugins, the source package must be used.

To build a SNAP source package with a plugin, perform the following instead of following the basic build steps:

1. Move to the sources folder. Run:

   ```
   cd /opt/nvidia/nvda_snap/src/
   ```

2. Build the sources with plugin to be enabled. Run:

   ```
   meson setup /tmp/build -Denable-bdev-null=true -Denable-bdev-malloc=true
   ```

3. Compile the sources. Run:

   ```
   meson compile -C /tmp/build
   ```

4. Install the sources. Run:

   ```
   meson install -C /tmp/build
   ```

5. Configure the SNAP environment variables and run SNAP service as explained in sections "Configure SNAP Environment Variables" and "Run SNAP Service".

8.12.2.6.16.1 Bdev

SNAP supports various block device (**bdev**) plugins to emulate storage backends without modifying core software. SPDK serves as the default **bdev** plugin; SNAP routes block device operations through SPDK unless an alternative is explicitly specified. For detailed SPDK documentation, refer to spdk_bdev.

The following plugins provide specific storage emulation capabilities:

Asynchronous I/O Block Device

The AIO plugin uses the Linux asynchronous I/O (**libaio**) interface to support block devices backed by regular files or raw Linux block devices. It routes SPDK storage operations through the kernel instead of user-space drivers.

- Use cases: Testing, development, and environments lacking direct hardware access. It also enables UEFI boot (including read-only boot images) by exposing a bootable disk before initializing hardware drivers.
- Constraints: Introduces higher latency than native SPDK drivers due to kernel overhead.
- Build option: Enable via **--enable-bdev-aio** .

AIO configuration example:

1. Create a file-backed AIO **bdev** :

```
fallocate -l 10G test
spdk_rpc.py bdev_aio_create test aio_test 512
```

2. Create an AIO **bdev** backed by a disk image and attach it to an NVMe controller:

```
spdk_rpc.py bdev_aio_create /path/to/disk.img aio_test 512
snap_rpc.py nvme_subsystem_create --nqn nqn.2022-10.io.nvda.nvme:0
snap_rpc.py nvme_namespace_create -b aio_test -n 1 --nqn nqn.2022-10.io.nvda.nvme:0 --uuid
263826ad-19a3-4febbc25-4bc81ee7749e
snap_rpc.py nvme_controller_create --nqn nqn.2022-10.io.nvda.nvme:0 --pf_id 0 --suspended
snap_rpc.py nvme_controller_attach_ns -c NVMeCtrl1 -n 1
snap_rpc.py nvme_controller_resume -c NVMeCtrl1
```

3. Delete the AIO **bdev** :

```
spdk_rpc.py bdev_aio_delete aio_test
```

Malloc

The Malloc plugin creates a memory-backed block device by allocating a buffer in RAM.

- Use cases: Performance analysis and debugging.
- Constraints: Not for production. Data is volatile and permanently lost upon system shutdown or device resizing.
- Build option: Enable via **--enable-bdev-malloc** .

Malloc configuration example:

1. Create a Malloc **bdev** and attach it to an NVMe controller:

```
snap_rpc.py snap_bdev_malloc_create --bdev test 64 512
snap_rpc.py nvme_subsystem_create -s nqn.2020-12.mlnx.snap
snap_rpc.py nvme_namespace_create -s nqn.2020-12.mlnx.snap -t malloc -b test -n 1
snap_rpc.py nvme_controller_create --pf_id=0 -s nqn.2020-12.mlnx.snap --mdts=7
snap_rpc.py nvme_controller_attach_ns -c NVMeCtrl1 -n 1
```

2. Resize a Malloc **bdev** :

```
snap_rpc.py snap_bdev_malloc_resize test 32
```

This destroys the existing device and creates a new one; all data is lost.

3. Delete the Malloc **bdev** :

```
snap_rpc.py snap_bdev_malloc_destroy test
```

NULL

The NULL plugin acts as a dummy block device that accepts I/O requests without performing actual physical storage operations.

- Use cases: Benchmarking and testing scenarios that require minimal system resources and no real storage hardware.
- Constraints: Not for production. Strictly for performance analysis and debugging.
- Build option: Enable via **`--enable-bdev-null`**.

NULL configuration example:

1. Create a NULL **`bdev`** and attach it to an NVMe controller:

```
snap_rpc.py snap_bdev_null_create_dbg test 1 512
snap_rpc.py nvme_subsystem_create -s nqn.2020-12.mlnx.snap
snap_rpc.py nvme_namespace_create -s nqn.2020-12.mlnx.snap -t malloc -b test -n 1
snap_rpc.py nvme_controller_create --pf_id=0 -s nqn.2020-12.mlnx.snap --mdts=7
snap_rpc.py nvme_controller_attach_ns -c NVMeCtrl1 -n 1
```

2. Delete the NULL **`bdev`**:

```
snap_rpc.py snap_bdev_null_destroy_dbg test
```

8.12.2.6.17 Single Protocol Mode

Single Protocol Mode restricts a SNAP instance to a single designated storage protocol (NVMe or virtio-blk). This configuration facilitates strict resource isolation and enables parallel container deployment.

8.12.2.6.17.1 System Behaviors

When operating in this mode, the system enforces the following behaviors:

- Initialization checks – During startup, SNAP verifies that the underlying hardware supports the configured protocol. If a capability mismatch occurs, the service fails to start and outputs a critical log detailing the missing hardware requirements.
- Function filtering – The **`emulation_function_list`** RPC response strictly filters its output, displaying only the functions associated with the configured protocol.
- Strict rejection – The system immediately rejects any hotplug or controller creation requests targeting an unsupported protocol, returning an explicit error message to the user.

8.12.2.6.17.2 Configuration

Enable and define the protocol restriction using the **`SNAP_SUPPORTED_PROTOCOL`** environment variable.

Parameter Value	Supported Protocol	Description
0 (or unset)	All protocols	Default behavior: The SNAP instance supports all available storage protocols.
1	NVMe only	Restricts the instance strictly to NVMe protocol operations.
2	Virtio-blk only	Restricts the instance strictly to virtio-blk protocol operations.

8.12.2.6.17.3 Deployment Guidelines and Constraints

When deploying multiple parallel SNAP containers, adhere to the following best practices to prevent resource contention and system conflicts:

- Protocol separation: Deploy concurrent containers using different protocols (e.g., one NVMe container alongside one virtio-blk container). Running multiple containers configured for the exact same protocol can cause severe conflicts when sharing firmware protocol objects.
- Resource partitioning: Running two parallel containers requires doubling the total hugepage allocation. Ensure other system resources, such as CPU cores and RAM, are explicitly partitioned between the instances. It is highly recommended to run each service on a dedicated set of CPU cores.
- Object isolation: Do not share identical operational objects across different containers. For example, avoid connecting two separate containers to the exact same remote block device (**bdev**).
- Live update preparation: Both containers of the same protocol (the source and the destination) should run on the same cores, entirely separate from the cores used by any other protocol. Set the CPU limit higher than the requested CPU to allow the temporary destination container to spawn during the update.

The following example demonstrates how to configure two distinct containers (Container A and Container B). Each container requests 2 CPU cores but specifies a limit of 4 cores to accommodate the temporary spawn of an additional container during a live update.

- Container A:

```
resources:
     requests:
       memory: "2Gi"
       hugepages-2Mi: "4Gi"
       cpu: "2"
     limits:
       memory: "4Gi"
       hugepages-2Mi: "4Gi"
       cpu: "4"
...
     - name: APP_ARGS
       value: "-m 0x3"
```

- Container B:

```
resources:
     requests:
       memory: "2Gi"
       hugepages-2Mi: "4Gi"
       cpu: "2"
     limits:
       memory: "4Gi"
       hugepages-2Mi: "4Gi"
       cpu: "4"
...
     - name: APP_ARGS
```

```
        value: "-m 0xc0"
```

8.12.2.6.18 UEFI Boot Configuration Walkthrough

This section provides a step-by-step guide for configuring SNAP NVMe and virtio-blk devices as UEFI boot targets. The workflow encompasses firmware configuration, SNAP backend preparation, and host BIOS/UEFI integration.

8.12.2.6.18.1 Prerequisites

Before initiating the configuration, ensure the environment meets the following baseline requirements:

- Software: BlueField must run a full DOCA image with SNAP-4 deployed (either as a container or a source-package service).
- Firmware: Storage emulation must be enabled in the firmware for NVMe and/or virtio-blk Physical Functions (PFs). Static PFs are strongly recommended for boot configurations.
- Storage backend: A bootable backend must exist and be reachable (e.g., a local NVMe SSD, a remote NVMe-oF RDMA target, or another supported SPDK **bdev**). This backend must contain a pre-installed OS or an accessible installer ISO.

8.12.2.6.18.2 Step 1: Configure UEFI Expansion ROM in Firmware

UEFI boot requires the activation of NVIDIA's UEFI expansion ROM drivers on the DPU. This allows the platform firmware to utilize NVIDIA's driver during the UEFI boot phase instead of relying on the vendor's default BIOS implementation.

1. Ensure the device operates in the internal CPU model and that storage emulation is active.
   - **INTERNAL_CPU_MODEL=1**
   - **PF_BAR2_ENABLE=0**

     > ⚠ This parameter is deprecated and must strictly be set to **0**.

   - **NVME_EMULATION_ENABLE=1** and/or **VIRTIO_BLK_EMULATION_ENABLE=1** (with at least one PF configured).
2. Execute the following **mlxconfig** command to enable the UEFI expansion ROMs for x86 hosts:

   ```
   mlxconfig -d /dev/mst/mt41692_pciconf0 s \
       EXP_ROM_VIRTIO_BLK_UEFI_x86_ENABLE=1 \
       EXP_ROM_NVME_UEFI_x86_ENABLE=1
   ```

3. Perform a complete power cycle of both the host and the DPU to apply the firmware configuration.

8.12.2.6.18.3 Step 2: Prepare SNAP Backend and Controller on the DPU

The UEFI stack strictly requires boot devices to be fully configured before the boot phase initiates. The SNAP PCIe function must be visible, and the backend disk must be attached to the emulated controller prior to the host reaching the UEFI boot sequence.

Hotplug Constraints

If utilizing hotplug PFs instead of static PFs, the hotplug operation must finalize before the BIOS begins the UEFI boot sequence. This limits hotplug boot support exclusively to warm reboot or external DPU power scenarios.

Option A: NVMe Configuration (Recommended Pattern)

To utilize NVMe for the system disk, create the controller in a suspended state, attach the namespace, and then resume operations. This guarantees the namespace is available upon initialization.

1. Attach the remote backend:

```
spdk_rpc.py bdev_nvme_attach_controller -b nvme0 -t rdma \
    -a <target_ip> -f ipv4 -s 4420 -n <subnqn>
```

2. Create the subsystem and namespace:

```
snap_rpc.py nvme_subsystem_create \
    --nqn nqn.2025-01.com.nvda:nvme:boot0

snap_rpc.py nvme_namespace_create \
    -b nvme0n1 -n 1 \
    --nqn nqn.2025-01.com.nvda:nvme:boot0 \
    --uuid <boot_ns_uuid>
```

3. Create the controller in a suspended state:

```
snap_rpc.py nvme_controller_create \
    --nqn nqn.2025-01.com.nvda:nvme:boot0 \
    --pf_id 0 \
    --ctrl NVMeBoot0 \
    --suspended \
    --num_queues 2
```

4. Attach the namespace and resume the controller:

```
snap_rpc.py nvme_controller_attach_ns -c NVMeBoot0 -n 1
snap_rpc.py nvme_controller_resume -c NVMeBoot0
```

Queue Requirements

The UEFI NVMe expansion ROM strictly requires a minimum of 2 I/O queues. Always configure **`--num_queues=2`** or higher on the boot controller.

Option B: virtio-blk Configuration

Configure a single system disk using the virtio-blk protocol:

1. Attach the remote backend:

```
spdk_rpc.py bdev_nvme_attach_controller -b nvme0 -t rdma \
    -a <target_ip> -f ipv4 -s 4420 -n <subnqn>
```

2. Create the virtio-blk controller:

```
snap_rpc.py virtio_blk_controller_create \
    --pf_id 0 \
    --ctrl VBLKBoot0 \
```

```
    --bdev nvme0n1
```

Validation

After executing either option, run `snap_rpc.py emulation_function_list` or `{nvme,virtio_blk}_controller_list` to verify the emulation function is present and successfully bound to the correct controller.

8.12.2.6.18.4 Step 3: Host BIOS/UEFI Configuration

With the firmware and SNAP environment fully configured, map the host BIOS/UEFI to boot from the SNAP-provided disk.

1. Access BIOS: Reboot the host into the BIOS/UEFI setup utility. Ensure the DPU remains powered and the SNAP service is fully active during this process.
2. Identify the device: Navigate to the UEFI boot menu and verify the SNAP emulated device appears as a bootable NVMe or virtio-blk disk.

   ⚠ Exact name and vendor strings depend on the specific firmware configuration.

3. Set boot priority: Select the emulated disk as the primary boot option. Alternatively, boot from an installer ISO, install the OS to the SNAP-provided disk, and subsequently set it as the primary system disk.

ℹ Troubleshooting: Boot Stalls

UEFI boot times over SNAP should parallel non-SNAP, bare-metal configurations. If the boot process stalls significantly (e.g., exceeding 1 minute), investigate the deployment for backend connectivity failures or configuration mismatches.

8.12.2.7 SNAP-4 Service Appendixes

8.12.2.7.1 Appendix – DPU Firmware Configuration

Before configuring SNAP, the user must ensure that all firmware configuration requirements are met. By default, SNAP is disabled and must be enabled by running both common SNAP configurations and additional protocol-specific configurations depending on the expected usage of the application (e.g., hot-plug, SR-IOV, UEFI boot, etc).

After configuration is finished, the host must be power cycled for the changes to take effect.

⚠ To verify that all configuration requirements are satisfied, users may query the current/next configuration by running the following:

```
mlxconfig -d /dev/mst/mt41692_pciconf0 -e query
```

8.12.2.7.1.1 System Configuration Parameters

Parameter	Description	Possible Values
INTERNAL_CPU_MODEL	Enable BlueField to work in internal CPU model ⚠ Must be set to 1 for storage emulations.	0/1
SRIOV_EN	Enable SR-IOV	0/1
PCI_SWITCH_EMULATION_ENABLE	Enable PCIe switch for emulated PFs	0/1
PCI_SWITCH_EMULATION_NUM_PORT	The maximum number of hotplug emulated PFs which equals PCI_SWITCH_EMULATION_NUM_PORT –1. For example, if PCI_SWITCH_EMULATION_NUM_PORT=32, then the maximum number of hotplug emulated PFs would be 31. ⚠ One switch port is reserved for all static PFs.	[0,2-32]

⚠ SRIOV_EN is valid only for static PFs.

8.12.2.7.1.2 RDMA/RoCE Configuration

By default, BlueField's RDMA/RoCE communication is blocked for its primary OS interfaces (known as ECPFs, typically mlx5_0 and mlx5_1).

If RoCE traffic is required, you must create additional network functions (scalable functions) that support RDMA/RoCE.

⚠ This configuration is not required when working over TCP or RDMA/IB.

To enable RoCE interfaces, run the following from within the DPU:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s PER_PF_NUM_SF=1
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s PF_SF_BAR_SIZE=8 PF_TOTAL_SF=2
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0.1 s PF_SF_BAR_SIZE=8 PF_TOTAL_SF=2
```

8.12.2.7.1.3 NVMe Configuration

Parameter	Description	Possible Values
NVME_EMULATION_ENABLE	Enable NVMe device emulation	0/1

Parameter	Description	Possible Values
NVME_EMULATION_NUM_PF	Number of static emulated NVMe PFs	[0-2]
NVME_EMULATION_NUM_MSIX	Number of MSIX assigned to emulated NVMe PFs ⚠ Firmware treats this value as a best effort value. The effective number of MSI-X given to the function should be queried as part of the nvme_controller_list RPC.	[0-63]
NVME_EMULATION_NUM_VF_MSIX	Number of MSIX per emulated NVMe VF ⚠ Firmware treats this value as a best effort value. The effective number of MSI-X given to the function should be queried as part of the nvme_controller_list RPC. ⚠ This value should match the maximum number of queues assigned to a VF's NVMe SNAP controller through the nvme_controller_create num_queues parameter +1 for the admin queue, as each queue requires one MSIX interrupt.	[0-4095]
NVME_EMULATION_NUM_VF	Number of VFs per emulated NVMe PF ⚠ If not 0, overrides NUM_OF_VFS ; valid only when SRIOV_EN=1 .	[0-256]
EXP_ROM_NVME_UEFI_x86_ENABLE	Enable NVMe UEFI exprom driver ⚠ Used for UEFI boot process.	0/1
NVME_EMULATION_MAX_QUEUE_DEPTH	Defines the default maximum queue depth for NVMe I/O queues for Physical and Virtual Functions. The value should be set to the binary logarithm of the desired maximum queue size. • A value of 0 (default) limits the queue size to 64. • The recommended value is 12 , which allows for a queue size of 4096.	[0-12]

8.12.2.7.1.4 Virtio-blk Configuration

Due to virtio-blk protocol limitations, using bad configuration while working with static virtio-blk PFs may cause the host server OS to fail on boot.

Before continuing, make sure you have configured:

- A working channel to access Arm even when the host is shut down. Setting such channel is out of the scope of this document. Please refer to *NVIDIA BlueField DPU BSP* documentation for more details.
- Add the following line to **/etc/nvda_snap/snap_rpc_init.conf** :

```
virtio_blk_controller_create -pf_id 0
```

For more information, please refer to section "Virtio-blk Emulation Management".

Parameter	Description	Possible Values
VIRTIO_BLK_EMULATION_ENABLE	Enable virtio-blk device emulation	0/1
VIRTIO_BLK_EMULATION_NUM_PF	Number of static emulated virtio-blk PFs ⚠ See warning above.	[0-30]
VIRTIO_BLK_EMULATION_NUM_MSIX	Number of MSIX assigned to emulated virtio-blk PFs ⚠ Firmware treats this value as a best effort value. The effective number of MSI-X given to the function should be queried as part of the virtio_blk_controller_list RPC command.	[0-63]
VIRTIO_BLK_EMULATION_NUM_VF_MSIX	Number of MSIX per emulated virtio-blk VF ⚠ Firmware treats this value as a best effort value. The effective number of MSI-X given to the function should be queried as part of the virtio_blk_controller_list RPC command. ⚠ This value should match the maximum number of queues assigned to a VF's Virtio BLK SNAP controller through the virtio_blk_controller_create num_queues parameter + 1 if the admin queue is requested , as each queue requires one MSIX interrupt.	[0-4095]
VIRTIO_BLK_EMULATION_NUM_VF	Number of VFs per emulated virtio-blk PF ⚠ If not 0, overrides NUM_OF_VFS ; valid only when SRIOV_EN=1	[0-2047]
EXP_ROM_VIRTIO_BLK_UEFI_x86_ENABLE	Enable virtio-blk UEFI exprom driver ⚠ Used for UEFI boot process.	0/1

8.12.2.7.2 Appendix – Configure Persistent Network Interfaces

To configure persistent network interfaces so they are not lost after reboot. Under **/etc/sysconfig/network-scripts** modify the following four files, or create them if do not exist, then perform a reboot:

```
# cd /etc/sysconfig/network-scripts/
# cat ifcfg-p0
```

```
  NAME="p0"
  DEVICE="p0"
  NM_CONTROLLED="no"
  DEVTIMEOUT=30
  PEERDNS="no"
  ONBOOT="yes"
  BOOTPROTO="none"
  TYPE=Ethernet
  MTU=9000

# cat ifcfg-p1
  NAME="p1"
  DEVICE="p1"
  NM_CONTROLLED="no"
  DEVTIMEOUT=30
  PEERDNS="no"
  ONBOOT="yes"
  BOOTPROTO="none"
  TYPE=Ethernet
  MTU=9000

# cat ifcfg-enp3s0f0s0
  NAME="enp3s0f0s0"
  DEVICE="enp3s0f0s0"
  NM_CONTROLLED="no"
  DEVTIMEOUT=30
  PEERDNS="no"
  ONBOOT="yes"
  BOOTPROTO="static"
  TYPE=Ethernet
  IPADDR=1.1.1.1
  PREFIX=24
  MTU=9000

# cat ifcfg-enp3s0f1s0
  NAME="enp3s0f1s0"
  DEVICE="enp3s0f1s0"
  NM_CONTROLLED="no"
  DEVTIMEOUT=30
  PEERDNS="no"
  ONBOOT="yes"
  BOOTPROTO="static"
  TYPE=Ethernet
  IPADDR=1.1.1.2
  PREFIX=24
  MTU=9000
```

8.12.2.7.3 Appendix – Building SNAP Container with Custom SPDK

The SNAP source package contains the files necessary for building a container with a custom SPDK.

To build the container:

1. Download and install the SNAP sources package:

   ```
   [dpu] # dpkg -i /path/snap-sources_<version>_arm64.deb
   ```

2. Navigate to the **src** folder and use it as the development environment:

   ```
   [dpu] # cd /opt/nvidia/nvda_snap/src
   ```

3. Copy the following to the container folder:
   - SNAP source package – required for installing SNAP inside the container
   - Custom SPDK – to **container/spdk**. For example:

     ```
     [dpu] # cp /path/snap-sources_<version>_arm64.deb  container/
     [dpu] # git clone -b v23.01.1 --single-branch --depth 1 --recursive --shallow-submodules https://
     github.com/spdk/spdk.git container/spdk
     ```

4. Modify the **spdk.sh** file if necessary as it is used to compile SDPK.
5. To build the container:
   - For Ubuntu, run:

     ```
     [dpu] # ./container/build_public.sh --snap-pkg-file=snap-sources_<version>_arm64.deb
     ```

   - For CentOS, run:

```
[dpu] # rpm -i snap-sources-<version>.el8.aarch64.rpm
[dpu] # cd /opt/nvidia/nvda_snap/src/
[dpu] # cp /path/snap-sources_<version>_arm64.deb container/
[dpu] # git clone -b v23.01.1 --single-branch --depth 1 --recursive --shallow-submodules https://
github.com/spdk/spdk.git container/spdk
[dpu] # yum install docker-ce docker-ce-cli
[dpu] # ./container/build_public.sh --snap-pkg-file=snap-sources_<version>_arm64.deb
```

6. Transfer the created image from the Docker tool to the crictl tool. Run:

```
[dpu] # docker save doca_snap:<version> doca_snap.tar
[dpu] # ctr -n=k8s.io images import doca_snap.tar
```

> ⚠ To transfer the container image to other setups, refer to appendix "Deploying Container on Setups Without Internet Connectivity".

7. To verify the image, run:

```
[DPU] #  crictl images
IMAGE                                TAG                 IMAGE ID            SIZE
docker.io/library/doca_snap          <version>           79c503f0a2bd7       284MB
```

8. Edit the image filed in the **container/doca_snap.yaml** file. Run:

```
image: doca_snap:<version>
```

9. Use the YAML file to deploy the container. Run:

```
[dpu] # cp doca_snap.yaml /etc/kubelet.d/
```

> ⚠ The container deployment preparation steps are required.

8.12.2.7.4 Appendix – Deploying Container on Setups Without Internet Connectivity

When Internet connectivity is not available on a DPU, Kubelet scans for the container image locally upon detecting the SNAP YAML. Users can load the container image manually before the deployment.

To accomplish this, users must download the necessary resources using a DPU with Internet connectivity and subsequently transfer and load them onto DPUs that lack Internet connectivity.

1. To download the **.yaml** file:

```
[bf] # wget --content-disposition https://api.ngc.nvidia.com/v2/resources/nvidia/doca/
doca_container_configs/versions/<path-to-yaml>/doca_snap.yaml
```

> ⚠ Access the latest download command on NGC. The **doca_snap:4.1.0-doca2.0.2** tag is used in this section as an example, and the latest tag is also available on NGC.

2. To download SNAP container image:

```
[bf] # crictl pull nvcr.io/nvidia/doca/doca_snap:4.1.0-doca2.0.2
```

3. To verify that the SNAP container image exists:

```
[bf] # crictl images

IMAGE                                     TAG                 IMAGE ID            SIZE
nvcr.io/nvidia/doca/doca_snap             4.1.0-doca2.0.2     9d941b5994057       267MB
k8s.gcr.io/pause                          3.2                 2a060e2e7101d       251kB
```

> **k8s.gcr.io/pause** image is required for the SNAP container.

4. To save the images as a **.tar** file:

```
[bf] # mkdir images
[bf] # ctr -n=k8s.io image export images/snap_container_image.tar nvcr.io/nvidia/doca/doca_snap:4.1.0-
doca2.0.2
[bf] # ctr -n=k8s.io image export images/pause_image.tar k8s.gcr.io/pause:3.2
```

5. Transfer the **.tar** files and run the following to load them into Kubelet:

```
[bf] # sudo ctr --namespace k8s.io image import images/snap_container_image.tar
[bf] # sudo ctr --namespace k8s.io image import images/pause_image.tar
```

6. Now, the image exists in the tool and is ready for deployment.

```
[bf] # crictl images
IMAGE                                     TAG                 IMAGE ID            SIZE
nvcr.io/nvidia/doca/doca_snap             4.1.0-doca2.0.2     9d941b5994057       267MB
k8s.gcr.io/pause                          3.2                 2a060e2e7101d       251kB
```

8.12.2.7.5 Appendix – Install Legacy SPDK

To build SPDK-19.04 for SNAP integration:

1. Cherry-pick a critical fix for SPDK shared libraries installation (originally applied on upstream only since v19.07).

```
[spdk.git] git cherry-pick cb0c0509
```

2. Configure SPDK:

```
[spdk.git] git submodule update --init
[spdk.git] ./configure --prefix=/opt/mellanox/spdk --disable-tests --without-crypto --without-fio --with-
vhost --without-pmdk --without-rbd --with-rdma --with-shared --with-iscsi-initiator --without-vtune
[spdk.git] sed -i -e 's/CONFIG_RTE_BUILD_SHARED_LIB=n/CONFIG_RTE_BUILD_SHARED_LIB=y/g' dpdk/build/.config
```

> The flags **--prefix** , **--with-rdma** , and **--with-shared** are mandatory.

3. Make SPDK (and DPDK libraries):

```
[spdk.git] make && make install
[spdk.git] cp dpdk/build/lib/* /opt/mellanox/spdk/lib/
[spdk.git] cp dpdk/build/include/* /opt/mellanox/spdk/include/
```

8.12.2.7.6 Appendix – PCIe BDF to VUID Translation

PCIe BDF (Bus, Device, Function) is a unique identifier assigned to every PCIe device connected to a computer. By identifying each device with a unique BDF number, the computer's OS can manage the system's resources efficiently and effectively.

PCIe BDF values are determined by host OS and are hence subject to change between different runs, or even in a single run. Therefore, the BDF identifier is not the best fit for permanent configuration.

To overcome this problem, NVIDIA devices add an extension to PCIe attributes, called VUIDs. As opposed to BDF, VUID is persistent across runs which makes it useful as a PCIe function identifier.

PCI BDF and VUID can be extracted one out of the other, using **lspci** command:

1. To extract VUID out of BDF:

```
[host] lspci -s <BDF> -vvv | grep -i VU | awk '{print $4}'
```

2. To extract BDF out of VUID:

```
[host] ./get_bdf.py <VUID>
[host] cat ./get_bdf.py
#!/usr/bin/python3

import subprocess
import sys

vuid = sys.argv[1]

# Split the output into individual PCI function entries
lspci_output = subprocess.check_output(['lspci']).decode().strip().split('\n')

# Create an empty dictionary to store the results
pci_functions = {}

# Loop through each PCI function and extract the BDF and full info
for line in lspci_output:
    bdf = line.split()[0]
    if vuid in subprocess.check_output(['lspci', '-s', bdf, '-vvv']).decode():
        print(bdf)
        exit(0)

print("Not Found")
```

8.12.2.7.7 Appendix – SNAP Memory Consumption

This appendix explains how SNAP consumes memory and how to manage memory allocation.

The user must allocate the DPA hugepages memory according to the section "Step 1: Allocate Hugepages". It is possible to use use a portion of the DPU memory allocation in the SNAP container as described in section "Adjusting YAML Configuration". This configuration includes the following minimum and maximum values:

- The minimum allocation which the SNAP container consumes:

```
resources:
  requests:
    memory: "4Gi"
```

- The maximum allocation that the SNAP container is allowed to consume:

```
resources:
  limits:
    hugepages-2Mi: "4Gi"
```

Hugepage memory is used by the following:

- SPDK **mem-size** global variable which controls the SPDK hugepages consumption (configurable in SPDK, 1GB by default)
- SNAP **SNAP_MEMPOOL_SIZE_MB** – used with non-ZC mode as IO buffers staging buffers on the Arm. By default, the SNAP mempool consumes 1G from the SPDK **mem-size** hugepages allocation. SNAP mempool may be configured using the **SNAP_MEMPOOL_SIZE_MB** global variable (minimum is 64 MB).

  > ⚠ If the value assigned is too low, with non-ZC, a performance degradation could be seen.

- SNAP and SPDK internal usage – 1G should be used by default. This may be reduced depending on the overall scale (i.e., VFs/num queues/QD).
- XLIO buffers – allocated only when NVMeTCP XLIO is enabled.

The following is the limit of the container memory allowed to be used by the SNAP container:

```
resources:
  limits:
    memory: "6Gi"
```

> ℹ This includes the hugepages limit (in this example, additional 2G of non-hugepages memory).

The SNAP container also consumes DPU SHMEM memory when NVMe recovery is used (described in section "NVMe Recovery"). In addition, the following resources are used:

```
limits:
  memory:
```

8.12.2.7.8 Appendix – Host OS Configuration

With Linux environment on host OS, additional kernel boot parameters may be required to support SNAP related features:

- To use SR-IOV:
  - For Intel, **intel_iommu=on iommu=pt** must be added
  - For AMD, **amd_iommu=on iommu=pt** must be added
- To use PCIe hotplug, **pci=realloc** must be added
- **modprobe.blacklist=virtio_blk,virtio_pci** for non-built-in **virtio-blk** driver or **virtio-pci** driver

To view boot parameter values, run:

```
cat /proc/cmdline
```

It is recommended to use the following with virtio-blk:

```
[dpu] cat /proc/cmdline BOOT_IMAGE … pci=realloc modprobe.blacklist=virtio_blk,virtio_pci
```

To enable VFs (virtio_blk/NVMe):

```
echo 125 > /sys/bus/pci/devices/0000\:27\:00.4/sriov_numvfs
```

8.12.2.7.8.1 Intel Server Performance Optimizations

```
cat /proc/cmdline
BOOT_IMAGE=(hd0,msdos1)/vmlinuz-5.15.0_mlnx root=UUID=91528e6a-b7d3-4e78-9d2e-9d5ad60e8273 ro crashkernel=auto
resume=UUID=06ff0f35-0282-4812-894e-111ae8d76768 rhgb quiet iommu=pt intel_iommu=on pci=realloc
modprobe.blacklist=virtio_blk,virtio_pci
```

8.12.2.7.8.2 AMD Server Performance Optimizations

```
cat /proc/cmdline
cat /proc/cmdline BOOT_IMAGE=(hd0,msdos1)/vmlinuz-5.15.0_mlnx root=UUID=91528e6a-b7d3-4e78-9d2e-9d5ad60e8273 ro
crashkernel=auto resume=UUID=06ff0f35-0282-4812-894e-111ae8d76768 rhgb quiet iommu=pt amd_iommu=on pci=realloc
modprobe.blacklist=virtio_blk,virtio_pci
```

8.12.2.7.9 Appendix – SPDK Configuration

The SPDK **bdev_nvme** module utilizing the RDMA transport does not support Completion Queue (CQ) resizing.

This constraint limits the number of SPDK NVMe RDMA I/O queues that can be created per core, which consequently restricts the maximum number of supported PCIe functions (including VFs, static PFs, and hot-plug PFs).

The exact limit depends on factors such as the maximum queue depth configured on the NVMe-oF RDMA target and the number of paths to that target. For example, with a maximum queue depth of 128 (the default for SPDK NVMe-oF targets) and a single path per subsystem, the default CQ size supports only up to 16 I/O queues per core.

To support larger-scale deployments, you must either enable a Shared Receive Queue (SRQ) or configure a larger CQ size.

8.12.2.7.9.1 Enable SRQ

The **bdev_nvme** module can be configured to use a SRQ. This allows multiple connections to share receive buffers, significantly increasing scalability. This is the recommended approach, though it may result in a minor IOPS reduction (around 5%) at exceptionally large scales (more than 500 queue pairs per core).

To enable SRQ, set the **--rdma-srq-size** option via the **bdev_nvme_set_options** RPC:

```
spdk_rpc.py bdev_nvme_set_options --rdma-srq-size <RDMA_SRQ_SIZE>
```

The SRQ size must be large enough to minimize Receiver Not Ready (RNR) events, which degrade performance. The optimal value depends on your specific traffic profile.

Size Parameters	Value/ Limit	Notes
Minimum recommended	**4096**	Setting the size smaller than 4096 is not recommended.

Size Parameters	Value/ Limit	Notes
Maximum supported	`32767`	The hard limit for the SRQ size.
Disable SRQ	`0`	Setting the size to 0 disables SRQ usage.

8.12.2.7.9.2 Configure CQ Size

The `bdev_nvme_set_options` RPC provides two parameters to set the initial and maximum CQ size: `--rdma-initial-cq-size` and `--rdma-max-cq-size`.

> ⚠ To avoid CQ resizing (which is unsupported), both parameters must be set to the exact same value.

The CQ size must be large enough to accommodate all Completion Queue Entries (CQEs) across all queue pairs. You can estimate the maximum number of outstanding CQEs using the formula

We don't have a way to export this macro.

> ⛔ If the CQ is too small or if the application does not poll it frequently enough, CQEs may be lost. This will cause I/O operations to hang indefinitely unless an I/O timeout is configured. For this reason, enabling SRQ is generally the safer and recommended approach.

Usage example:

```
spdk_rpc.py bdev_nvme_set_options --rdma-initial-cq-size <RDMA_CQ_SIZE> --rdma-max-cq-size <RDMA_CQ_SIZE>
```

Parameter	Value / Limit	Notes
Minimum recommended	`4096`	Setting the size smaller than 4096 is not recommended.
Maximum supported	`4194303`	The hard limit for the CQ size.

8.12.2.7.9.3 Disable `bdev` Read/Write Bypass for Large VF Scale

In large-scale deployments (e.g., 250+ Virtual Functions), SNAP and SPDK configurations utilizing virtio-blk or NVMe backends may experience significant read throughput degradation if the SPDK `bdev` read/write bypass path is enabled. Disabling this bypass restores read performance in these scenarios. Apply this configuration when tuning large VF deployments for read-heavy workloads.

> ⚠ You must issue this RPC during SPDK startup alongside your other initialization RPCs.

To disable the bypass path, run:

```
spdk_rpc.py bdev_set_options --disable-rw-bypass
```

To re-enable the bypass path (which is recommended for all standard, non-scaled configurations), run:

```
spdk_rpc.py bdev_set_options --enable-rw-bypass
```

.

8.12.2.7.10 Appendix – Custom Bdev Modules

This section explains how to extend the SNAP service with custom SPDK Bdev modules. The guide provides two distinct workflows:

1. Production Build: Creating a new container image that permanently includes the custom module.
2. Development Build: Manually building and running inside a container for rapid iteration and testing.

> ⚠ The examples provided below utilize the external **passthru** bdev module included with SPDK (**spdk/test/external_code/passthru**). You should adapt these paths and commands to suit your specific custom module and deployment environment.

8.12.2.7.10.1 Method 1: Building a Production Container (Recommended)

This method generates a standalone container image with the SNAP service and your custom functionality built-in, making it suitable for deployment via Kubernetes.

Environment Preparation

Create a working directory and clone the SPDK source code corresponding to your SNAP version.

```
# Create working directory (example: /opt/build)
mkdir -p /opt/build
cd /opt/build

# Clone SPDK sources
git clone https://github.com/Mellanox/spdk --branch v25.05.2.nvda

# Verify directory structure
ls -F
# Output should show: Dockerfile  spdk/
```

Create the Dockerfile

Create a **Dockerfile** in your working directory with the following content. This multi-stage build compiles the module and then copies it into the final SNAP image.

```
# Stage 1: Builder
FROM nvcr.io/nvstaging/doca/doca_snap:4.9.0-6-doca3.2.0 as builder

# Install build dependencies
RUN apt-get update && apt-get install -y \
    autoconf libtool python3-pyelftools libaio-dev \
    libncurses-dev libfuse3-dev patchelf libcmocka-dev make

# Copy external code source
COPY spdk/test/external_code /external_code

# Build the custom module
WORKDIR /external_code
```

```
ENV SPDK_HEADER_DIR=/opt/mellanox/spdk/include
ENV SPDK_LIB_DIR=/opt/mellanox/spdk/lib
ENV DPDK_LIB_DIR=/opt/mellanox/spdk/include

RUN make passthru_shared

# Stage 2: Final Image
FROM nvcr.io/nvstaging/doca/doca_snap:4.9.0-6-doca3.2.0

# Copy compiled shared object
COPY --from=builder /external_code/passthru/libpassthru_external.so /opt/nvidia/spdk/lib/
ENV SNAP_LD_PRELOAD=/opt/nvidia/spdk/lib/libpassthru_external.so

# Copy RPC python script
COPY --from=builder /external_code/passthru/bdev_passthru.py /usr/lib/python3/dist-packages/spdk/rpc/
ENV SPDK_RPC_PLUGIN="spdk.rpc.bdev_passthru"
```

Build and Deploy

1. Build the container image:

```
docker build -t doca_snap_custom_bdev:latest -f Dockerfile .
```

2. Import to Kubernetes (Kubelet):

```
# Export image to tarball
docker save doca_snap_custom_bdev:latest > doca_snap_custom_bdev.tar

# Import into containerd
ctr -n=k8s.io images import doca_snap_custom_bdev.tar
```

3. Edit **/etc/kubelet.d/doca_snap.yaml** to use the new image:

```
image: doca_snap_custom_bdev:latest
```

Configuration Example

Once the container is running, you can configure the custom bdev module via RPC.

```
# Identify the running SNAP container ID
SNAP_ID=$(crictl ps -s running -q --name snap)

# Create a base Malloc bdev
crictl exec $SNAP_ID spdk_rpc.py bdev_malloc_create 32 4096 -b Malloc0

# Construct the custom passthru bdev on top of Malloc0
crictl exec $SNAP_ID spdk_rpc.py construct_ext_passthru_bdev -b Malloc0 -p TestPT

# Create NVMe Subsystem and Controller
crictl exec $SNAP_ID snap_rpc.py nvme_subsystem_create --nqn nqn.2023-05.io.nvda.nvme:0
crictl exec $SNAP_ID snap_rpc.py nvme_controller_create --nqn nqn.2023-05.io.nvda.nvme:0 --pf_id 0 --ctrl NVMeCtrl0
--suspended

# Create Namespace using the custom bdev (TestPT)
crictl exec $SNAP_ID snap_rpc.py nvme_namespace_create --nqn nqn.2023-05.io.nvda.nvme:0 --bdev_name TestPT --nsid 1

# Attach Namespace and Resume Controller
crictl exec $SNAP_ID snap_rpc.py nvme_controller_attach_ns --ctrl NVMeCtrl0 --nsid 1
crictl exec $SNAP_ID snap_rpc.py nvme_controller_resume --ctrl NVMeCtrl0
```

8.12.2.7.10.2 Manual Development Workflow

This approach sets up an interactive development container, ideal for workflows requiring frequent code changes, recompilation, and testing.

Environment Preparation

Prepare the working directory and clone sources (same as the Production method).

```
mkdir -p /opt/build
cd /opt/build
git clone https://github.com/Mellanox/spdk --branch v25.05.2.nvda
```

Create Development Dockerfile

Create a simplified **Dockerfile** that installs dependencies and provides a shell.

```
FROM nvcr.io/nvstaging/doca/doca_snap:4.9.0-6-doca3.2.0 as builder

RUN apt-get update && apt-get install -y \
    autoconf libtool python3-pyelftools libaio-dev \
    libncurses-dev libfuse3-dev patchelf libcmocka-dev make

ENTRYPOINT /bin/bash
```

Build and Run Interactive Container

1. Build the dev image:

```
docker build -t doca_snap_custom_bdev_dev:latest -f Dockerfile .
```

2. Run interactively. Mount the necessary system volumes and your source code directory:

```
docker run -ti --privileged --net=host \
    --volume /dev/hugepages:/dev/hugepages \
    --volume /dev/shm:/dev/shm \
    --volume /dev/infiniband:/dev/infiniband \
    --volume /etc/nvda_snap:/etc/nvda_snap \
    --volume ${PWD}/spdk/test/external_code:/external_code \
    doca_snap_custom_bdev_dev:latest
```

Build and Test Inside Container

Once inside the container shell, perform the following steps:

1. Compile the module:

```
cd /external_code/
export SPDK_HEADER_DIR=/opt/mellanox/spdk/include
export SPDK_LIB_DIR=/opt/mellanox/spdk/lib
export DPDK_LIB_DIR=/opt/mellanox/spdk/include

make passthru_shared
```

2. Launch the service manually, preloading your compiled shared object:

```
# Copy RPC definition
cp /external_code/passthru/bdev_passthru.py /usr/bin/

# Load environment variables
source /opt/nvidia/nvda_snap/bin/set_environment_variables.sh

# Start Service with LD_PRELOAD
LD_PRELOAD=/external_code/passthru/libpassthru_external.so \
/opt/nvidia/nvda_snap/bin/snap_service -m 0xff &
```

3. Configure and test:

```
# Create base Malloc bdev
spdk_rpc.py bdev_malloc_create 32 4096 -b Malloc0

# Create custom Passthru bdev (explicitly specifying plugin)
spdk_rpc.py --plugin bdev_passthru construct_ext_passthru_bdev -b Malloc0 -p TestPT

# Setup NVMe Controller logic
snap_rpc.py nvme_subsystem_create --nqn nqn.2023-05.io.nvda.nvme:0
snap_rpc.py nvme_controller_create --nqn nqn.2023-05.io.nvda.nvme:0 --pf_id 0 --ctrl NVMeCtrl0 --suspended
snap_rpc.py nvme_namespace_create --nqn nqn.2023-05.io.nvda.nvme:0 --bdev_name TestPT --nsid 1
snap_rpc.py nvme_controller_attach_ns --ctrl NVMeCtrl0 --nsid 1
snap_rpc.py nvme_controller_resume --ctrl NVMeCtrl0
```

8.12.3 DOCA SNAP-3 User Guide

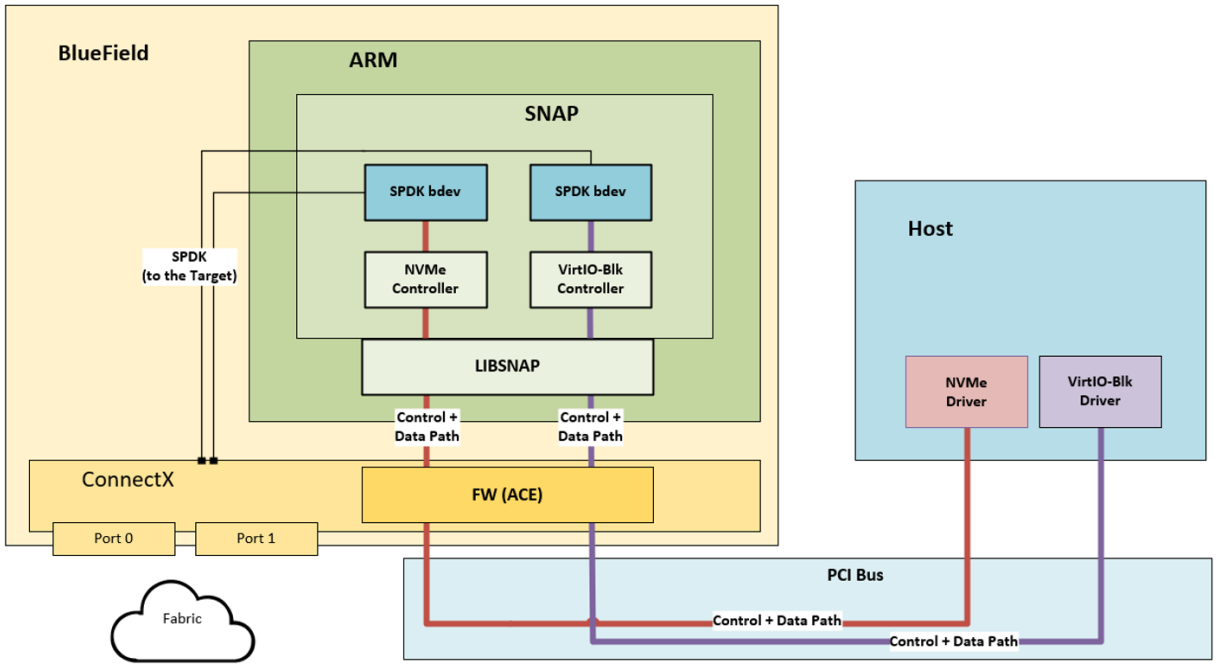
This guide details the configuration and usage of the SNAP to emulate NVMe and virtio-blk storage devices on NVIDIA® BlueField®-2 DPUs.

8.12.3.1 Introduction

8.12.3.1.1 SNAP

NVIDIA® BlueField® SNAP and virtio-blk SNAP (Storage-defined Network Accelerated Processing) technology enables hardware-accelerated virtualization of local storage. NVMe/virtio-blk SNAP presents networked storage as a local block-storage-device such as an SSD, emulating a local drive on the PCIe bus. The host OS/hypervisor uses its standard storage driver, unaware that communication is done, not with a physical drive, but with NVMe/virtio-blk SNAP framework. Any logic may be applied to the I/O requests or to the data via the NVMe/virtio-blk SNAP framework prior to redirecting the request and/or data over a fabric-based network to remote or local storage targets.

NVMe/virtio-blk SNAP is based on BlueField-2 DPU family technology and combines unique hardware-accelerated storage virtualization with the advanced networking and programmability capabilities of the DPU. NVMe/virtio-blk SNAP together with the BlueField DPU enable a world of applications addressing storage and networking efficiency and performance.



The traffic from a host-emulated PCIe device is redirected to its matching storage controller opened on the **mlnx_snap** service. The controller, from its side, holds at least one open backend device (usually SPDK block device). When a command is received, the controller executes it. Admin commands are answered immediately, while I/O commands are redirected to the backend device

for processing. The request handling pipeline is completely asynchronous and the workload is distributed across all Arm cores (allocated to SPDK application) to achieve the best performance.

The following are key concepts for SNAP:

- Full flexibility in fabric/transport/protocol (e.g. NVMe-oF/iSCSI/other, RDMA/TCP, ETH/IB)
- NVMe and virtio-blk emulation support
- Easy data manipulation
- Using Arm cores for data path

> ⚠ BlueField SNAP/virtio-blk SNAP are licensed software. Users must purchase a license per BlueField device to use them.

8.12.3.1.2 Libsnap

Libsnap is a common library designed to assist in common tasks for applications wishing to interact with emulated hardware over BlueField DPUs and take the most advantage from the hardware capabilities. As such, libsnap exposes a simple API for the upper layer application to create, modify, query, and destroy different emulation objects, such PCIe BAR management, emulated queues etc.

In addition, the library provides a set of helper functions to perform efficient DMA transactions between host and DPU memory.

SNAP application makes extensive usage of the libsnap library for resource management and efficient DMA operations required by the storage controllers.

8.12.3.2 SNAP Installation

8.12.3.2.1 SNAP Installation Process

8.12.3.2.1.1 DPU Image Installation

The BlueField OS image (BFB) includes all needed packages for mlnx_snap to operate: MLNX_OFED, RDMA-CORE libraries, and the supported SPDK version, libsnap and mlnx-snap headers, libraries and binaries.

To see which operating systems are supported, refer to the BlueField Software Documentation under Release Notes → Supported Platforms and Interoperability → Supported Linux Distributions.

RShim must be installed on the host to connect to the NVIDIA® BlueField® DPU. To install RShim, please follow the instructions described in the BlueField Software Documentation → BlueField DPU SW Manual → DPU Operation → DPU Bring-up and Driver Installation → Installing Linux on DPU → Step 1: Set up the RShim Interface.

Use RShim interface from the x86 host machine to install the desired image:

```
BFB=/<path>/latest-bluefield-image.bfb
cat $BFB > /dev/rshim0/boot
```

Optionally, it is possible to connect to the remote console of the DPU and watch the progress of the installation process. Using the **screen** tool, for example:

```
screen /dev/rshim0/console
```

8.12.3.2.1.2 Post-installation Configuration

Firmware Configuration

Refer to Firmware Configuration to confirm the firmware configuration matches the SNAP application's requirements (SR-IOV support, MSI-X resources, etc).

Network Configuration

Before enabling **mlnx_snap** or configuring it, users must first verify that uplink ports are configured correctly, and that network connectivity toward the remote target works properly.

By default, two SF interfaces are opened—one over each PF as configured in **/etc/mellanox/mlnx-sf.conf**—which match RDMA devices **mlx5_2** and **mlx5_3** respectively. As mentioned, only these interfaces may support RoCE/RDMA transport for the remote storage.

If working with an InfiniBand link, an active InfiniBand port must be made available to allow for InfiniBand support. Once an active IB port is available, users must configure the port RDMA device in the JSON configuration file (see **rdma_device** under "Configuration File Examples" for **mlnx_snap** to work on that port).

If working with bonding, it is transparent to MLNX SNAP configuration, and no specific configuration is necessary on NVMe/virtio-blk SNAP level.

8.12.3.2.1.3 Out-of-box Configuration

NVMe/virito-blk SNAP is disabled by default. Once enabled (see section "Firmware Configuration"), the out-of-box configuration of NVMe/virtio-blk SNAP includes a single NVMe controller, backed by a 64MB RAM-based SPDK block device (e.g. RAM drive) in non-offload mode. Out-of-box configuration does not include virtio-blk devices.

A sample configuration file for the out-of-box NVMe controller is located in **/etc/mlnx_snap/mlnx_snap.json**. For additional information about its values, please see section "Non-offload Mode".

The default initialization command set is described in **/etc/mlnx_snap/spdk_rpc_init.conf** and **/etc/mlnx_snap/snap_rpc_init.conf**, as follows:

- spdk_rpc_init.conf

```
bdev_malloc_create 64 512
```

- snap_rpc_init.conf

```
subsystem_nvme_create  Mellanox_NVMe_SNAP "Mellanox NVMe SNAP Controller"
controller_nvme_create mlx5_0 --subsys_id 0 --pf_id 0
controller_nvme_namespace_attach -c NvmeEmu0pf0 spdk Malloc0 1
```

⚠ BlueField out-of-box configuration is slightly different than BlueField-2. For a clean out-of-box experience, **/etc/mlnx_snap/snap_rpc_init.conf** is a symbolic link pointing to the relevant hardware-oriented configuration.

⚠ To make any other command set persistent, users may update and modify **/etc/mlnx_snap/spdk_rpc_init.conf** and **/etc/mlnx_snap/snap_rpc_init.conf** for according to their needs. Refer to section "SNAP Commands" for more information.

8.12.3.2.2 SNAP Service Control (systemd)

To enable, start, stop, or check the status of SNAP service, run:

```
systemctl {start | stop | status} mlnx_snap
```

8.12.3.2.3 Logging

The mlnx_snap application output is captured by the SystemD and stored in the internal database. Users are able to get the output from the service console by using the following SystemD commands:

- **systemctl status mlnx_snap**
- **journalctl -u mlnx_snap**

SystemD keeps logs in a binary format under the **/var/run/log/journal/** directory, which is stored on the tmpfs (i.e. it is not persistent).

SystemD redirects log messages into the **rsyslog** service. Configuration of **rsyslog** is default to CentOS/RHEL, so users may find all those messages in the **/var/log/messages** directory.

The **rsyslog** daemon may be configured to send messages to a remote (centralized) syslog server if desired.

8.12.3.3 SNAP Commands

8.12.3.3.1 RPC Command Interface

Remote Procedure Call (RPC) protocol is a very simple protocol defining a few data types and commands. NVMe/virtio-blk SNAP, like other standard SPDK applications, supports JSON-based RPC protocol commands, to control any resources create/delete/query/modify commands easily from CLI.

The mlnx_snap application supports executing all standard SPDK RPC commands, in addition to an extended SNAP-specific command set. SPDK standard commands are executed by the standard **spdk_rpc.py** tool, while the SNAP-specific command set extension is executed by an equivalent **snap_rpc.py** tool.

Full **spdk_rpc.py** command set documentation can be found in the SPDK official documentation site.

Full **snap_rpc.py** extended command is detailed later in this chapter.

8.12.3.3.2 PCIe Function Management Commands

8.12.3.3.2.1 Emulation Managers Discovery

Emulated PCIe functions are managed through IB devices called "emulation managers". The emulation managers are ordinary IB devices (e.g. mlx5_0, mlx5_1, etc.) with special privileges to also control PCIe communication and device emulations towards the host operating system. Numerous emulation managers may co-exist, each with its own set of capabilities.

The list of emulation managers with their capabilities can be queried using the following command:

```
snap_rpc.py emulation_managers_list
```

Appendix SPDK Configuration includes additional information.

8.12.3.3.2.2 Emulation Devices Configuration (Hotplug)

As mentioned above, each emulation manager holds a list of the emulated PCI functions it controls. The PCI functions may be approached later by either their function index (in emulation manager's list) or their PCIe BDF number (e.g. 88:00.2) as enumerated by the host OS. Some PCIe functions configured at the FW configuration stage are considered "static" (i.e. always present).

In addition, users can dynamically add detachable functions to that list at runtime (and to the host's PCIe devices list accordingly). These functions are called "Hotplugged" PCIe functions.

After a new PCIe function is plugged, it is shown on host PCIe devices list until either explicitly unplugged or the system goes through a cold reboot. Hot-plugged PCIe function will remain persistent even after SNAP process termination.

Some OSs automatically start to communicate with the new function after it is plugged. Some continue to communicate with the function (for a certain time) even after it is signaled to be unplugged. Therefore, users must always keep an open controller (of a matching type) over any existing configured PCIe function (see NVMe Controller Management and Virtio-blk Controller Management for more details).

The following command hotplugs a new PCIe function to the system:

```
snap_rpc.py emulation_device_attach emu_manager {nvme,virtio_blk} [--id ID] [--vid VID] [--ssid SSID] [--ssvid
SSVID] [--revid REVID] [--class_code CLASS_CODE] [--bdev_type {spdk,none}] [--bdev BDEV] [--num_queues NUM_QUEUES]
[--queue_depth QUEUE_DEPTH][--total_vf TOTAL_VF] [--num_msix NUM_MSIX]
```

The following command hot-unplugs a new PCIe function from the system:

```
snap_rpc.py emulation_device_detach <emu_manager> {nvme,virtio_blk} [-d PCI_BDF / -i PCI_INDEX / --vuid VUID]
```

Mandatory parameters:

- **emu_manager** – emulation manager

- **{nvme,virtio_blk}** – device type

Optional arguments:

- **--pci_bdf** – PCIe BDF identifier
- **--pci_index** – PCIe index identifier
- **--vuid** – PCIe VUID identifier
- **--force** – forcefully remove device (not recommended)

> ⚠ At least one identifier must be provided to describe the PCIe function to be detached.

> ⚠
> - Before calling **emulation_device_detach**, you need to uninstall or disable the controller from the Windows side (Device Manager → Storage Controllers).
> - Once a PCIe function is unplugged from the host system (when calling **emulation_device_detach**), its controller will be deleted implicitly also.

The following command lists all existing functions (either static or hotplugged):

```
snap_rpc.py emulation_functions_list
```

8.12.3.3.3 NVMe Emulation Management Commands

8.12.3.3.3.1 NVMe Subsystem

The NVMe subsystem, as described in the NVMe specification, is a logical entity which encapsulates sets of NVMe backends (or namespaces) and connections (or controllers). NVMe subsystems are extremely useful when working with multiple NVMe controllers, and especially when using NVMe virtual functions.

Each NVMe subsystem is defined by its serial number (SN), model number (MN), and qualified name (NQN). After creation, each subsystem also gets a unique index number.

The following example creates a new NVMe subsystem with a default generated NQN:

```
snap_rpc.py subsystem_nvme_create <serial_number> <model_number>
```

Mandatory parameters:

- **serial_number** – subsystem serial number
- **model_number** – subsystem model number

Optional arguments:

- **--nqn** – subsystem qualified name (auto-generated if not provided)
- **--nn** – maximal namespace ID allowed in the subsystem (default 0xFFFFFFFE; range 1-0xFFFFFFFE)

- **--mnan** – maximal number of namespaces allowed in the subsystem (default 1024; range 1-0xFFFFFFFE)

The following command deletes an NVMe subsystem:

```
snap_rpc.py subsystem_nvme_delete <NQN>
```

Where **<NQN>** is the subsystem's NQN.

The following command lists all NVMe subsystems:

```
snap_rpc.py subsystem_nvme_list
```

8.12.3.3.3.2 NVMe Controller

Each NVMe device (e.g. NVMe PCIe entry) exposed to the host, whether it is a PF or VF, must be backed by NVMe controller, which is responsible for all protocol communication with the host's driver. Every new NVMe controller must also be linked to NVMe subsystem. After creation, NVMe controllers can be addressed using either their name (e.g., "NvmeEmu0pf0") or both their subsystem NQN and controller ID.

The following command opens a new NVMe controller:

```
snap_rpc.py controller_nvme_create mlx5_0 [--pf_id ID / --pci_bdf / --vuid VUID] [--subsys_id ID / --nqn NQN]
```

Mandatory parameters:

- **emu_manager** – emulation manager

Optional parameters:

- **--vf_id VF_ID** – PCIe VF index to start emulation on (if the controller is destined to be opened on a VF). **--pf_id** must also be set for the command to take effect.
- **--conf** – JSON configuration file path to be used to provide an extended set of configuration parameters. Full information concerning the different parameters of the configuration file can be found under appendix "JSON File Format".
- **--nr_io_queues** – I/O queues maximal number (default 32, range 0-32)
- **--mdts** – maximum data transfer size (default 4, range 1-6)
- **--max_namespaces** – maximum number of namespaces for this controller (default 1024, range 1-0xFFFFFFFE)
- **--quirks** – bitmask to enable specific NVMe driver quirks to work with non-NVMe spec compliant drivers. For more information, refer to appendix "JSON File Format".
- **--mem {static,pool}** – use memory from a global pool or from dedicated buffers. See "Mem-pool" for more information.

The following command deletes an existing NVMe controller:

```
snap_rpc.py controller_nvme_delete [--name NAME / --subnqn SUBNQN --cntlid ID / --vuid VUID]
```

Optional arguments:

- **-c NAME**, **--name NAME** – controller Name. Must be set if **--nqn** and **--cntlid** are not set.
- **-n SUBNQN**, **--subnqn SUBNQN** – NVMe subsystem (NQN). Must be set if **--name** is not set.
- **-i CNTLID**, **--cntlid CNTLID** – controller identifier in NVMe subsystem. Must be set if **--name** is not set.

The following command lists all NVMe controllers:

```
snap_rpc.py controller_list --type nvme
```

Optional arguments:

- **-t {nvme,virtio_blk,virtio_net}**, **--type {nvme,virtio_blk,virtio_net}** – controller type

8.12.3.3.3.3 NVMe Backend (Namespace)

NVMe Namespaces are the representors of a continuous range of LBAs in the local/remote storage device (previously configured in "Backend Configuration" section). Each Namespace must be linked to a controller and have a unique identifier (NSID) across the entire NVMe subsystem (e.g. 2 namespaces cannot share the same NSID, even though linked to different controllers).

SNAP application uses SPDK block device framework as backends for its NVMe namespaces. Therefore, they should be configured in advance. For more information about SPDK block devices, see SPDK BDEV documentation and appendix "SPDK Configuration".

The following command attaches a new namespace to an existing NVMe controller:

```
snap_rpc.py controller_nvme_namespace_attach [--ctrl CTRL / --subnqn SUBNQN --cntlid ID] <bdev_type> <bdev> <nsid>
```

Mandatory parameters:

- **bdev_type** – block device type
- **bdev** – block device to use as backend
- **nsid** – namespace ID

Optional parameters:

- **-c CTRL**, **--ctrl CTRL** – controller name. Must be set if **--nqn** and **--cntlid** are not set.
- **-n SUBNQN**, **--subnqn SUBNQN** – NVMe subsystem (NQN). Must be set if **--ctrl** is not set.
- **-i CNTLID**, **--cntlid CNTLID** – controller identifier in NVMe subsystem. Must be set if **--ctrl** is not set.
- **-q QN**, **--qn QN** – QN of remote target which provides this namespace
- **-p PROTOCOL**, **--protocol PROTOCOL** – protocol used
- **-g NGUID**, **--nguid NGUID** – namespace globally unique identifier
- **-e EUI64**, **--eui64 EUI64** – namespace EUI-64 identifier
- **-u UUID**, **--uuid UUID** – namespace UUID

In full-offload mode, backends are acquired from the remote storage automatically and no manual configuration is required.

The following command detaches a namespace from a controller:

```
snap_rpc.py controller_nvme_namespace_detach [--ctrl CTRL / --subnqn SUBNQN --cntlid ID] <nsid>
```

Mandatory parameters:

- **nsid** – namespace ID

Optional parameters:

- **-c CTRL** , **--ctrl CTRL** – controller name. Must be set if **--nqn** and **--cntlid** are not set.
- **-n SUBNQN** , **--subnqn SUBNQN** – NVMe subsystem (NQN). Must be set if **--ctrl** is not set.
- **-i CNTLID** , **--cntlid CNTLID** – controller identifier in NVMe subsystem. Must be set if **--ctrl** is not set.

The following command lists a namespace of a controller:

```
snap_rpc.py controller_nvme_namespace_list [--ctrl CTRL / --subnqn SUBNQN --cntlid ID]
```

Optional parameters:

- **-c CTRL** , **--ctrl CTRL** – controller name. Must be set if **--nqn** and **--cntlid** are not set.
- **-n SUBNQN** , **--subnqn SUBNQN** – NVMe subsystem (NQN). Must be set if **--ctrl** is not set.
- **-i CNTLID** , **--cntlid CNTLID** – controller identifier in NVMe subsystem. Must be set if **--ctrl** is not set.

8.12.3.3.4 Virtio-blk Emulation Management Commands

8.12.3.3.4.1 Virtio-blk Controller

Each virtio-blk device (e.g. virtio-blk PCI entry) exposed to the host, whether it is PF or VF, must be backed by a virtio-blk controller. Virtio-blk is considered a limited storage protocol (compared to NVMe, for instance).

Due to protocol limitations:

- Trying to use a virtio-blk device (e.g. probe virtio-blk driver on host) without an already functioning virtio-blk controller, may cause the host server to hang until such controller is opened successfully (no timeout mechanism exists)
- Upon creation of a virtio-blk controller, a backend device must already exist

The following command creates a new virtio-blk controller:

```
snap_rpc.py controller_virtio_blk_create <emu_manager> [-d PCI_BDF / --pf_id PF_ID / --vuid VUID]
```

Mandatory parameters:

- **emu_manager** – emulation manager

Optional parameters:

- **--vf_id** – PCIe VF index to start emulation on, if controller is destined to be opened on VF
- **--num_queues** – number of queues (default 64, range 2-256)

- **--queue_depth** – queue depth (default 128, range 1-256)
- **--size_max** –- maximal SGE data transfer size (default 4096, range 1 – MAX_UINT16). See virtio specification for more information.
- **--seg_max** – maximal SGE list length (default 1, range 1-queue_depth). See virtio specification for more information.
- **--bdev_type** – block device type (spdk/none). Note that opening a controller with **none** backend means to open it with a backend of size of 0.
- **--bdev** – SPDK block device to use as backend
- **--serial** – serial number for the controller
- **--force_in_order** - force I/O in-order completions. Note that this value is required to ensure future virtio-blk controllers always successfully recover after an application crash.
- **--suspend** – create controller is in a SUSPENDED state (must be explicitly resumed later)
- **--mem {static,pool}** – use memory from a global pool or from dedicated buffers. See "Advanced Features" for more information.

The following command deletes a virtio-blk controller:

```
snap_rpc.py controller_virtio_blk_delete [-c NAME / --vuid VUID]
```

Mandatory arguments:

- **name** – controller name

Optional arguments:

- **--f** , **--force** – force controller deletion

The following command lists all virtio-blk controllers:

```
snap_rpc.py controller_list --type virtio_blk
```

Optional arguments:

- **-t {nvme,virtio_blk,virtio_net}** , **--type {nvme,virtio_blk,virtio_net}** – controller type

Virtio-blk controllers can also be suspended and resumed. While suspended, the controller stops receiving new requests from the host driver and only finishes handling of requests already in flight (without prompting any IO errors back to the driver).

The following command suspends/resumes virtio-blk controller:

```
snap_rpc.py controller_virtio_blk_suspend <name>
snap_rpc.py controller_virtio_blk_resume <name>
```

Mandatory arguments:

- **name** – controller name

8.12.3.3.4.2 Virtio-blk Backend Management

Like NVMe, virtio-blk also uses SPDK block devices framework as its backend devices, but since virtio-blk is a limited storage protocol as opposed to NVMe, whose backend management abilities are limited as well:

- Virtio-blk protocol supports only one backend device
- Virtio-blk protocol does not support administration commands to add backends, thus all backend attributes are communicated to the host virtio-blk driver over PCIe BAR and must be accessible during driver probing. For that reason, backends can only be changed when PCIe function is not in use by any host storage driver.

For these reasons, when the host driver is active, all backend management operations must occur only when the controller is in suspended state.

The following command attaches a new backend to a controller:

```
snap_rpc.py controller_virtio_blk_bdev_attach <ctrl_name> {spdk} <bdev_name>
```

Mandatory arguments:

- **name** – controller name
- **{spdk}** – block device type
- **bdev** – block device to use as backend

Optional arguments:

- **--size_max** – maximal SGE data transfer size (no hard limit). See Virtio specification for more information.
- **--seg_max** – maximal SGE list length (no hard limit). See Virtio specification for more information.

The following command detaches a backend from a controller:

```
snap_rpc.py controller_virtio_blk_bdev_detach <ctrl_name>
```

Mandatory arguments:

- **ctrl_name** – controller name

> ⚠ Destruction of SPDK block devices using SPDK block devices' API is considered a **controller_virtio_blk_bdev_detach** and is bound to the same limitations.

The following command lists a backend detail of a controller:

```
snap_rpc.py controller_virtio_blk_bdev_list <name>
```

Mandatory arguments:

- **name** – controller name

8.12.3.3.5 Debug and Statistics

BlueField and virito-blk SNAP provide a set of commands which help customers retrieve performance and debug statistics about the opened emulated devices. The statistics are provided at the SNAP controller level (whether for NVMe or Virtio-blk).

8.12.3.3.5.1 IO Statistics

The following commands are available to measure how many successful/failed IO operations were executed by the controller.

These commands have minimal effect on BlueField SNAP performance and can therefore be used to sample statistics while the controller performs high bandwidth IO operations.

```
snap_rpc.py controller_nvme_get_iostat [-c CTRL_NAME]
snap_rpc.py controller_virtio_blk_get_iostat [-c CTRL_NAME]
```

These commands have minimal effect on BlueField SNAP performance and can therefore be used to sample statistics while the controller performs high-bandwidth IO operations.

Mandatory arguments:

- **CTRL_NAME** – controller name

NVMe/Virtio IO statistics:

- **read_ios** – number of read commands handled
- **completed_read_ios** – number of read commands completed successfully
- **err_read_ios** – number of read commands completed with error
- **write_ios** – number of write commands handled
- **completed_write_ios** – number of write commands completed successfully
- **err_write_ios** – number of write commands completed with error
- **flush_ios** – number of flush commands handled
- **completed_flush_ios** – number of flush commands completed successfully
- **err_flush_ios** – number of flush commands completed with error

Virtio IO specific statistics:

- **fatal_ios** – number of commands dropped and never completed
- **outstanding_in_ios** – number of outstanding IOs at a given moment
- **outstanding_in_bdev_ios** – number of outstanding IOs at a given moment, pending backend handling
- **outstanding_to_host_ios** – number of outstanding IOs at a given moment, pending DMA handling

8.12.3.3.5.2 Debug Statistics

The following commands are available to examine the controller and queues with more detailed status and information.

When queried frequently, these commands may impact performance and should therefore be called for debug purposes only.

```
snap_rpc.py controller_nvme_get_debugstat [-c NAME]
snap_rpc.py controller_virtio_blk_get_debugstat [-c NAME]
```

8.12.3.3.6 Initialization Scripts

The default initialization scripts **/etc/mlnx_snap/spdk_rpc_init.conf** and **/etc/mlnx_snap/snap_rpc_init.conf** allow users to control the startup configuration.

These scripts, which are used for the out-of-box configuration, may be modified by the user to control the SNAP initialization:

- The **spdk_rpc_init.conf** may be modified with the SPDK commands listed under the SPDK Configuration appendix.
- The **snap_rpc_init.conf** may be modified with the **snap_rpc** commands described throughout this chapter (SNAP Commands).

8.12.3.4 Advanced Features

8.12.3.4.1 Performance Optimization

⚠ Tuning MLNX SNAP for the best performance may require additional resources from the system (CPU, memory) and may affect SNAP controller scalability.

8.12.3.4.1.1 Increasing Number of Used Arm Cores

By default, MLNX SNAP uses 4 Arm cores, with core mask 0xF0. The core mask is configurable in **/etc/default/mlnx_snap** (parameter **CPU_MASK**) for best performance (i.e., **CPU_MASK=0xFF**).

⚠ As SNAP is an SPDK based application, it constantly polls the CPU and therefore occupies 100% of the CPU it runs on.

8.12.3.4.1.2 Disabling Mem-pool

When mem-pool is enabled, that reduces the memory footprint but decreases the overall performance.

To configure the controller to not use mem-pool, set **MEM_POOL_SIZE=0** in **/etc/default/mlnx_snap**.

See section "Mem-pool" for more information.

8.12.3.4.1.3 Maximizing Single IO Transfer Data Payload

Increasing datapath staging buffer sizes improves performance for larger block sizes (>4K):

- For NVMe, this can be controlled by increasing the MDTS value either in the JSON file or the RPC parameter. For more information regarding MDTS, refer to the NVMe specification. The default value is 4 (64K buffer), and the maximum value is 6 (256K buffer).
- For virtio-blk, this can be controlled using the **seg_max** and **size_max** RPC parameters. For more information regarding these parameters, refer to the VirtIO-blk specification. No hard-maximum limit exists.

8.12.3.4.1.4 Increasing Emulation Manager MTU

The default MTU for the emulation manager network interface is 1500. Increasing MTU to over 4K on the emulation manager (e.g., MTU=4200) also enables the SNAP application to transfer larger amount of data in a single Host→DPU memory transactions, which may improve performance.

8.12.3.4.1.5 Optimizing Number of Queues and MSIX Vector (virtio-blk only)

SNAP emulated queues are spread evenly across all configured PFs (static and dynamic) and defined VFs per PF (whether functions are being used or not). This means that the larger the total number of functions SNAP is configured with (either PFs or VFs), the less queues and MSIX resources each function will be assigned which would affect its performance accordingly. Therefore, it is recommended to configure in Firmware Configuration the minimal number of PFs and VFs per PF desired for that specific system.

Another consideration is matching between MSIX vector size and the desired number of queues. The standard virtio-blk kernel driver uses an MSIX vector to get events on both control and data paths. When possible, it assigns exclusive MSIX for each virtqueue (e.g., per CPU core) and reserves an additional MSIX for configuration changes. If not possible, it uses a single MSIX for all virtqueues. Therefore, to ensure best performance with virtio-blk devices, the condition **VIRTIO_BLK_EMULATION_NUM_MSIX** > **virtio_blk_controller.num_queues** must be applied.

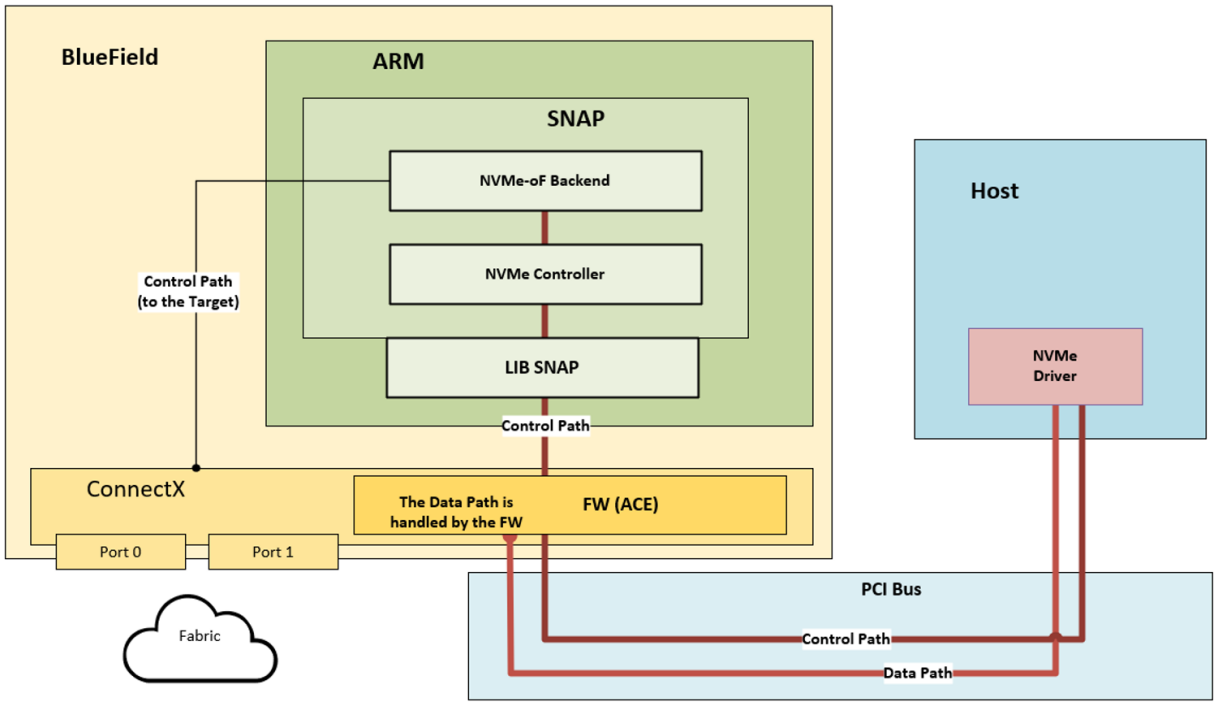
> ⚠ The total number of MSIXs is limited on BlueField-2 cards, so MSIX reservation considerations may apply when running with multiple devices. For more information, refer to this FAQ.

8.12.3.4.2 NVMe-RDMA Full Offload Mode

The NVMe-RDMA full offload mode allows reducing the Arm cores CPU cost by offloading the data-path directly to the firmware/hardware. If the user needs to control the data plane or the backend this mode does not allow it.

In full offload mode the control plane is handled at the SW level, while the data plane is being handled at the FW level and requires no SW interaction. For that reason, the user has no control over the backend devices. Thus they are detected automatically and no namespace management commands are required.

The NVMe-RDMA architecture:



In this mode, a remote target parameter must be provided using a JSON configuration file (a JSON file example can be found in **/etc/mlnx_snap/mlnx_snap_offload.json.example**) and the NVMe controller can detect and connect to the relevant backends by itself.

As the SNAP application does not participate in the datapath and needs fewer resources, it is recommended to reduce CPU_MASK to a single core (i.e., **CPU_MASK=0x80**). Refer to "Increasing Number of Used Arm Cores" for **CPU_MASK** configuration.

After configuration is done, users must create the NVMe subsystem and (offloaded) controller. Note that **snap_rpc.py controller_nvme_namespace_attach** is not required and **--rdma_device mlx5_2** is provided to mark the relevant RDMA interface for the connection.

The following example creates an NVMe full-offload controller:

```
# snap_rpc.py subsystem_nvme_create  "Mellanox_NVMe_SNAP" "Mellanox NVMe SNAP Controller"
# snap_rpc.py controller_nvme_create mlx5_0 --subsys_id 0 --pci_bdf 88:00.2 --nr_io_queues 32 --mdts 4 -c /etc/
mlnx_snap/mlnx_snap.json --rdma_device mlx5_2
```

This is the matching JSON file example:

```
{
  "ctrl": {
    "offload": true,
  },
  "backends": [
    {
      "type": "nvmf_rdma",
      "name": "testsubsystem",
      "paths": [
        {
          "addr": "1.1.1.1",
          "port": 4420,
          "ka_timeout_ms": 15000,
          "hostnqn": "r-nvmx03"
```

```
        }
      ]
    }
  ]
}
```

Full offload mode requires that the provided RDMA device (given in **--rdma_device** parameter) supports RoCE transport (typically SF interfaces). Full offload mode for virtio-blk is not supported.

The discovered namespace ID may be remapped to get another ID when exposed to the host in order to comply with firmware limitations.

8.12.3.4.3 SR-IOV

SR-IOV configuration depends on the kernel version:

- Optimal configuration may be achieved with a new kernel in which the **sriov_drivers_autoprobe sysfs** entry exists in **/sys/bus/pci/devices/<BDF>/**
- Otherwise, the minimal requirement may be met if the **sriov_totalvfs sysfs** entry exists in **/sys/bus/pci/devices/<BDF>/**

SR-IOV configuration needs to be done on both the host and DPU side, marked in the following example as [HOST] and [ARM] respectively. This example assumes that there is 1 VF on static virtio-blk PF 86:00.3 (NVMe flow is similar), and that a **Malloc0** SPDK BDEV exists.

8.12.3.4.3.1 Optimal Configuration

```
[ARM] snap_rpc.py controller_virtio_blk_create mlx5_0 -d 86:00.3 --bdev_type none
[HOST] modprobe -v virtio-pci && modprobe -v virtio-blk
[HOST] echo 0 > /sys/bus/pci/devices/0000:86:00.3/sriov_drivers_autoprobe
[HOST] echo 1 > /sys/bus/pci/devices/0000:86:00.3/sriov_numvfs [ARM] snap_rpc.py controller_virtio_blk_create
mlx5_0 --pf_id 0 --vf_id 0 --bdev_type spdk --bdev Malloc0
\* Continue by binding the VF PCIe function to the desired VM. *\
```

After configuration is finished, no disk is expected to be exposed in the hypervisor. The disk only appears in the VM after the PCIe VF is assigned to it using the virtualization manager. If users want to use the device from the hypervisor, they simply need to bind the PCIe VF manually.

8.12.3.4.3.2 Minimal Requirement

```
[ARM] snap_rpc.py controller_virtio_blk_create mlx5_0 -d 86:00.3 --bdev_type none
[HOST] modprobe -v virtio-pci && modprobe -v virtio-blk
[HOST] echo 1 > /sys/bus/pci/devices/0000:86:00.3/sriov_numvfs
\* the host now hangs until configuration is performed on the DPU side *\ [ARM] snap_rpc.py
controller_virtio_blk_create mlx5_0 --pf_id 0 --vf_id 0 --bdev_type spdk --bdev Malloc0
\* Host is now released *\
\* Continue by binding the VF PCIe function to the desired VM. *\
```

Hotplug PFs do not support SR-IOV.

It is recommended to add **pci=assign-busses** to the boot command line when creating more than 127 VFs.

Without this option, the following errors may appear from host, and the virtio driver will not probe these devices.

```
pci 0000:84:00.0: [1af4:1041] type 7f class 0xffffff
pci 0000:84:00.0: unknown header type 7f, ignoring device
```

8.12.3.4.4 Zero Copy (SNAP-direct)

Zero-copy is supported on SPDK 21.07 and higher.

The SNAP-direct feature allows SNAP applications to transfer data directly from the host memory to remote storage without using any staging buffer inside the DPU.

SNAP enables the feature according to the SPDK BDEV configuration only when working against an SPDK NVMe-oF RDMA block device.

To configure the controller to use Zero Copy, set the following in **/etc/default/mlnx_snap**:

- For virtio-blk:

```
VIRTIO_BLK_SNAP_ZCOPY=1
```

- For NVMe:

```
NVME_SNAP_ZCOPY=1
```

8.12.3.4.5 NVMe/TCP Zero Copy

NVMe/TCP Zero Copy is implemented as a custom NVDA_TCP transport in SPDK NVMe initiator and it is based on a new XLIO socket layer implementation.

The implementation is different for Tx and Rx:

- The NVMe/TCP Tx Zero Copy is similar between RDMA and TCP in that the data is sent from the host memory directly to the wire without an intermediate copy to Arm memory
- The NVMe/TCP Rx Zero Copy allows achieving partial zero copy on the Rx flow by eliminating copy from socket buffers (XLIO) to application buffers (SNAP). But data still must be DMA'ed from Arm to host memory.

To enable NVMe/TCP Zero Copy, use **SPDK v22.05.nvda --with-xlio**.

For more information about XLIO including limitations and bug fixes, refer to the NVIDIA Accelerated IO (XLIO) Documentation.

To configure the controller to use NVMe/TCP Zero Copy, set the following in **/etc/default/mlnx_snap**:

```
EXTRA_ARGS="-u -mem-size 1200 -wait-for-rpc"
NVME_SNAP_TCP_RX_ZCOPY=1
SPDK_XLIO_PATH=/usr/lib/libxlio.so
MIN_HUGEMEM=4G
```

To connect using NVDA_TCP transport:

- If **/etc/mlnx_snap/spdk_rpc_init.conf** is not being used, add the following at the start of the file in the given order:

  ```
  sock_set_default_impl -i xlio
  framework_start_init
  ```

  When the **mlnx_snap** service is started, run the following command:

  ```
  [ARM] spdk_rpc.py bdev_nvme_attach_controller -b <NAME> -t NVDA_TCP -f ipv4 -a <IP> -s <PORT> -n <SUBNQN>
  ```

- If **/etc/mlnx_snap/spdk_rpc_init.conf** is not being used, once the service is started, run the following commands in the given order:

  ```
  [ARM] spdk_rpc.py sock_set_default_impl -i xlio
  [ARM] spdk_rpc.py framework_start_init
  [ARM] spdk_rpc.py bdev_nvme_attach_controller -b <NAME> -t NVDA_TCP -f ipv4 -a <IP> -s <PORT> -n <SUBNQN>
  ```

NVDA_TCP transport is fully interoperable with other implementations based on the NVMe/TCP specifications.

NVDA_TCP limitations:

- SPDK multipath is not supported
- NVMe/TCP data digest is not supported
- SR-IOV is not supported

8.12.3.4.6 Robustness and Recovery

As SNAP is a standard user application running on the DPU OS, it is vulnerable to system interferences, like closing SNAP application gracefully (i.e., stopping mlnx_snap service), killing SNAP process brutally (i.e., running **kill -9**), or even performing full OS restart to DPU. If there are exposed devices already in use by host drivers when any of these interferences occur, that may cause the host drivers/application to malfunction.

To avoid such scenarios, the SNAP application supports a "Robustness and Recovery" option. So, if the SNAP application gets interrupted for any reason, the next instance of the SNAP application will be able to resume where the previous instance left off.

This functionality can be enabled under the following conditions:

- Only virtio-blk devices are used (this feature is currently not supported for NVMe protocol)

- By default, SNAP application is programmed to survive any kind of "graceful" termination, including controller deletion, service restart, and even (graceful) Arm reboot. If extended protection against brutal termination is required, such as sending SIGKILL to SNAP process or performing brutal Arm shutdown, the **--force_in_order** flag must be added to the **snap_rpc.py controller_virtio_blk_create** command.

  > ⚠ The **force_in_order** flag may impact performance if working with remote targets as it may cause high rates of out-of-order completions or if different queues are served in different rates.

- It is the user's responsibility to open the recovered virtio-blk controller with the exact same characteristics as the interrupted virtio-blk controller (same remote storage device, same BAR parameters, etc.)

8.12.3.4.7 Mem-pool

By default, SNAP application pre-allocates all required memory buffers in advance.

A great amount of allocated memory may be required when using:

- Large number of controllers (as with SR-IOV)
- Large number of queues per controller
- High queue-depth
- Large mdfs (for NVMe) or **seg_max** and **size_max** (for virtio-blk)

To reduce the memory footprint of the application, users may choose to use mem-pool (a shared memory buffer pool) instead. However, using mem-pool may decrease overall performance.

To configure the controller to use mem-pool rather than private ones:

1. In **/etc/default/mlnx_snap**, set the parameter **MEM_POOL_SIZE** to a non-zero value. This parameter accepts K/M/G notations (e.g., **MEM_POOL_SIZE=100M**). If K/M/G notation is not specified, the value defaults to bytes.
2. Users must choose the right value for their needs—a value too small may cause longer starvations, while a value too large consumes more memory. As a rule of thumb, typical usage may choose to set it as a minimum (num_devices*4MB, 512MB).
3. Upon controller creation, add the option **--mempool**. For example:

```
snap_rpc.py controller_nvme_create mlx5_0 –subsys_id 0 –pf_id 0 --mem pool
```

> ⚠ The per controller mem-pool configuration is independent from all others. Users can set some controllers to work with mem-pool and other controllers to work without it.

8.12.3.4.8 Virtio-blk Transitional Device (0.95)

SNAP supports virtio-blk transitional devices. Virtio transitional devices refer to devices supporting drivers conforming to modern specification and legacy drivers (conforming to legacy 0.95 specifications).

To configure virtio-blk PCIe functions to be transitional devices, special firmware configuration parameters must be applied:

- **VIRTIO_BLK_EMULATION_PF_PCI_LAYOUT** (0: MODERN / 1: TRANSITIONAL) – configures transitional device support for PFs

  > ⚠ This parameter is currently not supported.

- **VIRTIO_BLK_EMULATION_VF_PCI_LAYOUT** (0: MODERN / 1: TRANSITIONAL) – configures transitional device support for underlying VFs

  > ⚠ This parameter is currently not supported.

- **VIRTIO_EMULATION_HOTPLUG_TRANS** (True/False) – configures transitional device support for hot-plugged virtio-blk devices

To use virtio-blk transitional devices, Linux boot parameters must be set on the host:

- If the kernel version is older than 5.1, set the following Linux boot parameter on the host OS:

  ```
  intel_iommu=off
  ```

- If **virtio_pci** is built-in from host OS, set the following Linux boot parameter:

  ```
  virtio_pci.force_legacy=1
  ```

- If **virtio_pci** is a kernel module rather than built-in from host OS, use force legacy to load the module:

  ```
  modprobe -rv virtio_pci
  modprobe -v virtio_pci force_legacy=1
  ```

For hot-plugged functions, additional configuration must be applied during SNAP hotplug operation:

```
# snap_rpc.py emulation_device_attach mlx5_0 virtio_blk -- transitional_device --bdev_type spdk --bdev BDEV
```

8.12.3.4.9 Virtio-blk Live Migration

Live migration is a standard process supported by QEMU which allows system administrators to pass devices between virtual machines in a live running system. For more information, refer to QEMU VFIO device Migration documentation.

Live migration is supported for SNAP virtio-blk devices. It can be activated using a driver with proper support (e.g., NVIDIA's proprietary VDPA-based Live Migration Solution). For more info, refer to TBD.

If the physical function (PF) has been removed, for instance, using vDPA provisioning virtio-blk PF with the command:

```
python ./app/vfe-vdpa/vhostmgmt mgmtpf -a 0000:af:00.3
```

It is advisable to confirm and restore the presence of controllers in SNAP before attempting to re-add them using the command:

```
python dpdk-vhost-vfe/app/vfe-vdpa/vhostmgmt vf -v /tmp/sock-blk-0 -a 0000:59:04.5
```

8.12.3.5 Appendix – Live Upgrade

The following procedure is designed for live deployment of small software bug fixes or modifications made in the SNAP application. Using this procedure for other purposes (e.g., bumping SNAP service to a new version on top of an older BFB image) may cause SNAP to malfunction.

To live upgrade SNAP, 2 SNAP processes must be opened in parallel.

All system resources (e.g., hugepages, memory) must be sufficient to temporarily support 2 SNAP application instances operating in parallel during the upgrade procedure.

8.12.3.5.1 Passing virtio-blk Controller's Management Between SNAP Processes

1. Open 2 SNAP processes simultaneously on the Arm.

   This requires changing the SPDK RPC server path.

   For lower downtime, it is highly recommended to run each process on a different CPU mask.

   For SNAP Process 1, run:

```
./mlnx_snap_emu -m 0xf0 -r /var/tmp/spdk.sock1
```

For SNAP Process 2, run:

```
./mlnx_snap_emu -m 0x0f -r /var/tmp/spdk.sock2
```

2. Connect to the same bdev with both processes (i.e., with Malloc device).
   For SNAP Process 1, run:

```
spdk_rpc.py -s /var/tmp/spdk.sock1 bdev_malloc_create -b Malloc1 1024 512
```

   For SNAP Process 2, run:

```
spdk_rpc.py -s /var/tmp/spdk.sock2 bdev_malloc_create -b Malloc1 1024 512
```

3. Open a virtio-blk controller on the SNAP Process 1:

```
snap_rpc.py -s /var/tmp/spdk.sock1 controller_virtio_blk_create mlx5_0 --pf_id 0 --bdev_type spdk --bdev
Malloc1 --num_queues 16
```

4. Load virtio-blk driver on the host side and start using it.
5. Delete the virtio-blk controller instance from SNAP Process 1 and immediately open a virtio-blk controller on SNAP Process 2:

```
snap_rpc.py -s /var/tmp/spdk.sock1 controller_virtio_blk_delete VblkEmu0pf0 --force &&  snap_rpc.py -s /
var/tmp/spdk.sock2 controller_virtio_blk_create mlx5_0 --pf_id 0 --bdev_type spdk --bdev Malloc1  --
num_queues 16
```

8.12.3.5.2 Full "Live Upgrade" Procedure

Assuming there exists a fully configured SNAP service is already running on the system:

1. Create a local copy of SNAP binary file (e.g., under **/tmp** folder):

```
cp /usr/bin/mlnx_snap_emu /tmp/
```

2. For all active virtio-blk controllers, follow management passing procedure as described in section "Passing virtio-blk Controller's Management Between SNAP Processes".
3. Stop original SNAP service.

```
systemctl stop mlnx_snap
```

4. Upgrade SNAP service.
   - If installed from binary, use Linux official installation framework (**apt** / **yum**)
   - If installed from sources, follow the same installation process as done originally
5. Repeat management passing procedure, this time to move back control from the local copy into the official (updated) version of SNAP service.

8.12.3.6 Appendix – Host OS Configuration

With Linux environment on host OS, additional kernel boot parameters may be required to support SNAP related features:

- To use SR-IOV, **intel_iommu=on iommu=ptmust** be added
- To use PCIe hotplug, **pci=realloc** must be added
- When using SR-IOV, **pci=assign-busses** must be added

To view boot parameter values, use the command **cat /proc/cmdline**.

8.12.3.7 Appendix – SPDK Configuration

SPDK backend (BDEV) management commands:

```
spdk_rpc.py bdev_nvme_attach_controller -b <name> -t rdma -a <ip> -f ipv4 -s <port> -n <nqn>
spdk_rpc.py bdev_nvme_detach_controller <name>
spdk_rpc.py bdev_null_create <name> <size_mb> <blk_size>
spdk_rpc.py bdev_null_delete <name>
spdk_rpc.py bdev_aio_create <filepath> <name> <blk_size>
spdk_rpc.py bdev_aio_delete <name>
```

For more information, please refer to SPDK BDEV documentation.

8.12.3.8 Appendix – Firmware Configuration

Before configuring mlnx_snap, the user must ensure all FW configuration requirements are met. By default, mlnx_snap is disabled, and needs to be enabled by running both common mlnx-snap configuration, and additional protocol-specific configuration depending on the expected usage of the application (e.g. Hotplug, SR-IOV, UEFI boot, etc).

After all configuration is finished, power-cycling the host is required for these changes to take effect.

> ⚠ To verify that all configuration requirements are satisfied, users may query the current/next configuration by running the following:
>
> ```
> mlxconfig -d /dev/mst/mt41686_pciconf0 -e query
> ```

8.12.3.8.1 Basic Configuration

1. (Optional) Reset all previous configuration.

   ```
   [dpu] mst start
   [dpu] mlxconfig -d /dev/mst/mt41686_pciconf0 reset
   ```

   > ⚠ This will return your product to its default configurations. Do this only if you were not able to get SNAP to work.

2. Set general basic parameters.
   On BlueField-2:

   ```
   [dpu] mlxconfig -d /dev/mst/mt41686_pciconf0 s INTERNAL_CPU_MODEL=1
   ```

   On BlueField:

```
[dpu] sudo mlxconfig -d /dev/mst/mt41682_pciconf0 s INTERNAL_CPU_MODEL=1 PF_BAR2_ENABLE=1 PF_BAR2_SIZE=1
```

3. When using RDMA/RoCE transport, additional parameters must be configured:

```
[dpu] mlxconfig -d /dev/mst/mt41686_pciconf0 s PER_PF_NUM_SF=1
[dpu] mlxconfig -d /dev/mst/mt41686_pciconf0 s PF_SF_BAR_SIZE=8 PF_TOTAL_SF=2
[dpu] mlxconfig -d /dev/mst/mt41686_pciconf0.1 s PF_SF_BAR_SIZE=8 PF_TOTAL_SF=2
```

8.12.3.8.2 System Configuration Parameters

Parameter	Description	Possible Values	Comments
SRIOV_EN	Enable SR-IOV	0/1	
NUM_OF_VFS	Number of VFs per emulated PF	[0-127]	
NUM_PF_MSIX	Number of MSIX assigned to emulated PF	[0-63]	
NUM_VF_MSIX	Number of MSIX assigned to emulated VF	[0-63]	
PCI_SWITCH_EMULATION_ENABLE	Enable PCIe switch for emulated PFs	0/1	
PCI_SWITCH_EMULATION_NUM_PORT	Max number of emulated PFs	[0-32]	Single port is reserved for all static PFs

SRIOV_EN is valid only for static PFs

8.12.3.8.3 NVMe Configuration

Parameter	Description	Possible Values	Comments
NVME_EMULATION_ENABLE	Enable NVMe device emulation	0/1	
NVME_EMULATION_NUM_PF	Number of static emulated NVMe PFs	[0-2]	
NVME_EMULATION_NUM_MSIX	Number of MSIX assigned to emulated NVMe PF/VF	[0-63]	
NVME_EMULATION_NUM_VF	Number of VFs per emulated NVMe PF	[0-127]	If not 0, overrides NUM_OF_VFS ; valid only when SRIOV_EN=1
EXP_ROM_NVME_UEFI_x86_ENABLE	Enable NVMe UEFI exprom driver	0/1	Used for UEFI boot process

8.12.3.8.4 Virtio-blk Configuration

Due to virtio-blk protocol limitations, using bad configuration while working with static virtio-blk PFs may cause the host server OS to fail on boot.

Before continuing, make sure you have configured:

- A working channel to access Arm even when the host is shut down. Setting such channel is out of the scope of this document. Please refer to "NVIDIA BlueField BSP documentation" for more details.
- Add the following line to **/etc/mlnx_snap/snap_rpc_init.conf** :

```
controller_virtio_blk_create mlx5_0 --pf_id 0 --bdev_type none
```

For more information, please refer to section "Virtio-blk Controller Management".

Parameter	Description	Possible Values	Comments
VIRTIO_BLK_EMULATION_ENABLE	Enable virtio-blk device emulation	0/1	
VIRTIO_BLK_EMULATION_NUM_PF	Number of static emulated virtio-blk PFs	[0-2]	See WARNING above
VIRTIO_BLK_EMULATION_NUM_MSIX	Number of MSIX assigned to emulated virtio-blk PF/VF	[0-63]	
VIRTIO_BLK_EMULATION_NUM_VF	Number of VFs per emulated virtio-blk PF	[0-127]	If not 0, overrides NUM_OF_VFS ; valid only when SRIOV_EN=1
EXP_ROM_VIRTIO_BLK_UEFI_x86_ENABLE	Enable virtio-blk UEFI exprom driver	0/1	Used for UEFI boot process

8.12.3.9 Appendix – JSON File Format

This section is relevant only for the following cases:

- Using legacy mode in which the user prefers to not use the recommended SNAP commands, but to use the JSON file format
- NVMe-RDMA full offload mode in which the configuration is only possible with the JSON file format

The configuration parameters are divided into two categories: Controller and backends.

8.12.3.9.1 Configuration File Examples

8.12.3.9.1.1 Legacy Mode Configuration

> For the non-full offload mode, it is recommended to use the SNAP RPC commands (described in SNAP Commands) and not the legacy mode of the JSON file format described in this section.

```
{
  "ctrl": {
    "func_num": 0,
    "rdma_device": "mlx5_2",
    "sqes": 0x6,
    "cqes": 0x4,
    "cq_period": 3,
    "cq_max_count": 6,
    "nr_io_queues": 32,
    "mn": "Mellanox BlueField NVMe SNAP Controller",
    "sn": "MNC12",
    "mdts": 4,
    "oncs": 0,
    "offload": false,
    "max_namespaces": 1024,
    "quirks": 0x0,
    "version": "1.3.0"
  },
  "backends": [
    {
      "type": "spdk_bdev",
      "paths": [
        {

        }
      ]
    }
  ]
}
```

8.12.3.9.1.2 NVMe-RDMA Full Offload Mode Configuration

> For NVMe-RDMA full offload mode, users can only use the JSON file format (and not the SNAP RPC commands).

```
{
  "ctrl": {
    "func_num": 0,
    "rdma_device": "mlx5_2",
    "sqes": 0x6,
    "cqes": 0x4,
    "cq_period": 3,
    "cq_max_count": 6,
    "nr_io_queues": 32,
    "mn": "Mellanox BlueField NVMe SNAP Controller",
    "sn": "MNC12",
    "mdts": 4,
    "oncs": 0,
    "offload": true,
    "max_namespaces": 1024,
    "quirks": 0x0
    "version": "1.3.0"
  },
  "backends": [
    {
      "type": "nvmf_rdma",
      "name": "testsubsystem",
      "paths": [
        {
          "addr": "1.1.1.1",
          "port": 4420,
          "ka_timeout_ms": 15000,
          "hostnqn": "r-nvmx03"
        }
      ]
    }
  ]
}
```

8.12.3.9.2 Configuration Parameters

8.12.3.9.2.1 Controller Parameters

Parameters in SNAP JSON configuration file. Default file is located in `/etc/mlnx_snap/mlnx_snap.json`.

Parameter	Description	Legal Values	Default
`rpc_server`	Describes the RPC server socket for passing through RPC commands. Relevant only when using vendor-specific RPC commands from host.	Any	`""`
`offload`	Enable full-offload mode	`true`, `false`	`false`
`nr_io_queues`	Maximum number of I/O queues. ⚠ The actual number of queues is limited by number of queues supported by FW.	≥0	`32`
`mn`	Model number	String (up to 40 chars)	`"MLX NVMe Ctrl"`
`sn`	Serial number	String (up to 20 chars)	`"MNC12"`
`nn`	Number of namespaces (NN) indicates the maximum value of a valid NSID for the NVM subsystem. If the `mnan` field is cleared to `0h`, then this field also indicates the maximum number of namespaces supported by the NVM subsystem.	0-0xFFFFFFFE	`0xFFFFFFFE`
`mnan`	Maximum number of allowed namespaces (MNAN) supported by the NVM subsystem	1-0xFFFFFFFE	`1024`
`mdts`	Max data transfer size. This value is in units of the minimum memory page size (CAP.MPSMIN) and is reported as a power of two (2^n). A value of 0h indicates that there is no maximum data transfer size.	1-6	`4`

Parameter	Description	Legal Values	Default
`quirks`	Bitmask for enabling specific NVMe driver quirks in order to work with non-NVMe spec compliant drivers. • Bit 0 – send namespace change async events even if driver does not explicitly request them via the SET_FTRS command ⚠ Enable this if the NVMe driver knows to handle namespace change but does not use SET_FTRS. CentOS 7.5 inbox driver does this. • Bit 1 – send new namespace change events even if previous ones are not yet cleared by the driver ⚠ CentOS 7.5 inbox driver requires this. • Bit 2 – force Number of Namespaces (NN) on Identify controller to dynamically track and indicate both the maximum value of a valid NSID and the maximum number of namespaces supported by the NVM controller. There is no limitation on namespaces NSIDs on controller level. • Bit 3 – force OACS to enable namespace management bit ⚠ VMWare driver requires this bit to be set.	`0x0` - `0x3`	`0x0`
`max_namespaces`	Limit number of available namespaces	Any	`1024`

8.12.3.9.2.2 Backend Parameters

Theses parameters are used to define the backend server.

⚠ Even though a list of backends can be configured, currently only a single backend is supported.

Parameter	Description	Legal Values	Default
`type`	Backend type: • `"memdisk"` – RAM based local storage • `"nvmf_rdma"` – NVMe-oF over RDMA remote storage • `"posix_io"` – file-based storage • `"spdk_bdev"` – SPDK block devices	`nvmf_rdma`, `spdk_bdev`	`"spdk_bdev"`
`name`	Depends on backend type: • `"nvmf_rdma"` – remote subsystem name • `"memdisk"` / `"spdk_bdev"` – unused • `"posix_io"` – backend filename	Any	Null

Parameter	Description	Legal Values	Default
size_mb	Represents the desired size (in MB) of the opened backend ⚠ Relevant only for memdisk / posix_io backends.	Any	Unused
block_order	Represents the desired block size (in logarithmic scale) of the opened backend ⚠ Relevant only for memdisk / posix_io backends.	9, 12	Unused

Path Section

This section is relevant only if backend type is set to `nvmf_rdma`. For each backend, a list of paths can be specified using the following parameters:

Parameter	Description	Legal Values	Default
addr	Target IPv4 address	String in a.b.c.d format	"192.168.101.2"
port	Target port number	1024-65534	4420
ka_timeout_ms	Keepalive timeout in msec	>0	15000
Nqn	Host NQN	String up to 223-char long	"nqn.2014-08.org.nvmexpress:uuid:11111111-2222-3333-4444-555555555555"

8.12.3.10 Appendix – Frequently Asked Questions

8.12.3.10.1 How do I enable SNAP?

Please refer to section "SNAP Installation".

8.12.3.10.2 How do I configure SNAP to support VirtIO-blk?

Please refer to section "Virtio-blk Configuration".

8.12.3.10.3 How do I configure SNAP to work with both ports (for the same or for multiple targets)?

Assumptions:

- The remote target is configured with nqn "Test" and 1 namespace, and it exposes it through the 2 RDMA interfaces 1.1.1.1/24 and 2.2.2.1/24
- The RDMA interfaces are 1.1.1.2/24 and 2.2.2.2/24

Non-offload mode configuration:

1. Create the SPDK BDEVS. Run:

```
spdk_rpc.py bdev_nvme_attach_controller -b Nvme0 -t rdma -a 1.1.1.1 -f ipv4 -s 4420 -n Test
spdk_rpc.py bdev_nvme_attach_controller -b Nvme1 -t rdma -a 2.2.2.1 -f ipv4 -s 4420 -n Test
```

2. Create NVMe controller. Run:

```
snap_rpc.py controller_nvme_create mlx5_0 --subsys_id 0 -c /etc/mlnx_snap/mlnx_snap.json --rdma_device
mlx5_2
```

3. Attach the namespace twice, one through each port. Run:

```
snap_rpc.py controller_nvme_namespace_attach -c NvmeEmu0pf0 spdk Nvme0n1 1
snap_rpc.py controller_nvme_namespace_attach -c NvmeEmu0pf0 spdk Nvme1n1 2
```

At this stage, you should see **/dev/nvme0n1** and **/dev/nvme0n2** on the host "nvme list", both of which are mapped to the same remote disk via 2 different ports.

Full-offload mode configuration:

> ⚠ Full-offload mode currently allows users to connect to multiple remote targets in parallel (but not to the same remote target through different paths).

1. Create 2 separate JSON full-offload configuration files (see section "NVMe-RDMA Full Offload Mode Configuration"). Each describe a connection to remote target via different RDMA interface.
2. Configure 2 separate NVMe device entries to be exposed to the host either as hot-plugged PCIe functions or “static” ones (see section "Firmware Configuration").
3. Create 2 NVMe controllers, one per RDMA interface. Run:

```
snap_rpc.py subsystem_nvme_create Mellanox_NVMe_SNAP "Mellanox NVMe SNAP Controller"
snap_rpc.py controller_nvme_create mlx5_0 --subsys_id 0 --pf_id 0 -c /etc/mlnx_snap/mlnx_snap_p0.json --
rdma_device mlx5_2
snap_rpc.py subsystem_nvme_create  Mellanox_NVMe_SNAP "Mellanox NVMe SNAP Controller"
snap_rpc.py controller_nvme_create mlx5_0 --subsys_id 1 --pf_id 1 -c /etc/mlnx_snap/mlnx_snap_p1.json --
rdma_device mlx5_3
```

> ⚠ NVMe controllers may also share the same NVMe subsystem. In this case, users must make sure all namespaces in all remote targets have a distinct NSID.

At this stage, you should see **/dev/nvme0n1** and **/dev/nvme1n1** on the host **nvme list**.

8.12.3.10.4 How do I configure offload mode? Which protocols are supported?

Please refer to section "NVMe-RDMA Full Offload Mode Configuration".

For more information on full offload, please refer to section "NVMe-RDMA Full Offload Mode".

8.12.3.10.5 How do I configure Firmware for SNAP?

Please refer to section "Firmware Configuration".

8.12.3.10.6 Can I work with custom SPDK on Arm?

MLNX SNAP is natively compiled against NVIDIA's internal branch of SPDK. It is possible to work with different SPDK versions, under the following conditions:

- mlnx-snap sources must be recompiled against the new SPDK sources
- The new SPDK version changes do not break any external SPDK APIs

Integration process:

1. Build SPDK (and DPDK) with shared libraries.

```
[spdk.git] ./configure --prefix=/opt/mellanox/spdk-custom --disable-tests --disable-unit-tests --without-
crypto --without-fio --with-vhost --without-pmdk --without-rbd --with-rdma --with-shared --with-iscsi-
initiator --without-vtune --without-isal
[spdk.git] make && sudo make install
[spdk.git] cp -r dpdk/build/lib/* /opt/mellanox/spdk-custom/lib/
[spdk.git] cp -r dpdk/build/include/* /opt/mellanox/spdk-custom/include/
```

> ⚠ It is also possible to install DPDK in that directory but copying suffices.

> ⚠ Only the flag **with-shared** is mandatory

2. Build SNAP against the new SPDK.

```
[mlnx-snap.src] ./configure --with-snap --with-spdk=/opt/mellanox/spdk-custom --without-gtest --prefix=/usr
[mlnx-snap.src] make -j8 && sudo make install
```

3. Append additional custom libraries to the mlnx-snap application. Set **LD_PRELOAD="/opt/mellanox/spdk/lib/libspdk_custom_library.so"**.

> ⚠ Additional SPDK/DPDK libraries required by **libspdk_custom_library.so** might also need to be attached to **LD_PRELOAD**.

> ⚠ **LD_PRELOAD** setting can be added to **/etc/default/mlnx_snap** for persistent work with the mlnx_snap system service.

4. Run application.

8.12.3.10.7 Can I replace my backend storage at runtime?

NVMe protocol has an embedded support for backends (namespaces) attach/detach at runtime.

To change backend storage during runtime for NVMe, run:

```
snap_rpc.py controller_nvme_namespace_detach -c NvmeEmu0pf0 1
```

```
snap_rpc.py controller_nvme_namespace_attach -c NvmeEmu0pf0 spdk nvme0n1 1
```

Virtio-blk does not have similar support in its protocol's specification. Therefore, detaching while running IO results in error on any IO received between the request to detach and attach.

To change backend storage at runtime for virtio-blk, run:

```
snap_rpc.py controller_virtio_blk_bdev_detach VblkEmu0pf0
snap_rpc.py controller_virtio_blk_bdev_attach VblkEmu0pf0 spdk nvme0n1
```

8.12.3.10.8 I'm suffering from low performance after updating to latest mlnx-snap. How can I fix it?

After adding the option to work with a large number of controllers, resource considerations had to be considered. It was necessary to pay special attention to the MSIX resource, which is limited to ~1K across the whole BlueField-2 card. Therefore, new PCI functions are now opened with limited resources by default (specifically, MSIX is set to 2).

User may choose to assign more resources for a specific function, as detailed in the following:

1. Increase the number of MSIX allowed to be assigned to a function (power-cycle may be required for changes to take effect):

   ```
   [dpu] mlxconfig -d /dev/mst/mt41686_pciconf0 s VIRTIO_BLK_EMULATION_NUM_MSIX=63
   ```

2. Hotplug virtio-blk PF with the increased value of MSIX.

   ```
   [dpu] snap_rpc.py emulation_device_attach mlx5_0 virtio_blk --num_msix=63
   ```

3. Open the controller with increased number of queues (1 queue per MSIX, and leave another free MSIX for configuration interrupts):

   ```
   [dpu] snap_rpc.py controller_virtio_blk_create mlx5_0 --pf_id 0 --bdev_type spdk --bdev Null0 --
   num_queues=62
   ```

For more information, please refer to section "Performance Optimization".

8.12.3.11 SNAP-3 Release Notes

The release note pages provide information for NVIDIA® BlueField®-2 SNAP software such as changes and new features, software known issues, and bug fixes.

8.12.3.11.1 Changes and New Features

- N/A

8.12.3.11.2 Known Issues

The following are known limitations of this NVMe/virtio-blk SNAP software version.

Ref #	Issue
4712565	Description: Windows2019 is not supported on the OpenEuler BFB in SNAP 3.8.0-13.
	Workaround: N/A
	Keywords: Windows 19; OpenEuler BFB
	Discovered in version: 3.8.0-13
–	Description: NVMeTCP XLIO is currently not supported when running 64K page size kernels on the DPU Arm cores (as is the case for CentOS 8.x, Rocky 8.x, or openEuler 20.x).
	Workaround: N/A
	Keywords: 64K page size; NVMeTCP XLIO
	Discovered in version: 3.6.0
–	Description: When running with virtio-blk and virtio-net protocols in parallel, performance may be negatively impacted.
	Workaround: N/A
	Keywords: Performance
	Discovered in version: 3.7.2
2957317	Description: Due to an upstream kernel bug that exists in some Linux kernel distributions, the command **emulation_device_detach** times out, which causes any inflight traffic to hang.
	Workaround: It is recommended to ensure that all inflight traffic on the device is stopped before performing a hotunplug.
	Keywords: PCIe Hotplug
	Discovered in version: 3.6.0
3046440	Description: NVMe full-offload mode does not work properly over the first generation of BlueField SoCs
	Workaround: N/A
	Keywords: NVMe full-offload mode
	Discovered in version: 3.6.0
2879262	Description: Due to a kernel bug that exists in some Linux kernel distributions, configuring large number of virtio queues along with a small number of MSIX may lead the kernel to a soft lock-up (on top of causing significant performance degradation).
	Workaround: It is recommended that to keep virtio-blk controller's **--num_queues** value in **snap_rpc.py controller_virtio_blk_create** is smaller than the value of **VIRTIO_BLK_EMULATION_NUM_MSIX** (which is configured through **mlxconfig**).
	Keywords: Virtio-blk; kernel hang
	Discovered in version: 3.6.0
–	Description: SPDK multipath is supported only with NVMe over RDMA (and not with NVMe over TCP).
	Workaround: N/A
	Keywords: SPDK; NVMe
	Discovered in version: 3.6.0

Ref #	Issue
3055119	Description: Windows driver does not work with Virtio-blk SNAP-Direct feature.
	Workaround: To disable the feature when working with Windows OS, user must set `VIRTIO_BLK_SNAP_ZCOPY=0` in `/etc/default/mlnx_snap`.
	Keywords: Windows
	Discovered in version: 3.5.0
–	Description: NVMe multipath features cannot be obtained when using SNAP in full-offload mode configuration
	Workaround: N/A
	Keywords: NVMe full-offload mode; multipath
	Discovered in version: 3.4.0
–	Description: After each PCIe device hot-plug, a matching controller must be immediately opened. Specifically, hot-unplugging the device before a controller is created may cause the host kernel driver to malfunction on some Linux distributions.
	Workaround: N/A
	Keywords: Hot-plug; controller
	Discovered in version: 3.3.0
–	Description: SR-IOV on hot-plugged PFs is not supported
	Workaround: N/A
	Keywords: PCIe Hotplug
	Discovered in version: 3.2.0
–	Description: Any PCIe emulated device exposed to the host must have a matching controller opened on it in mlnx_snap service prior to loading its kernel driver. This includes virtio-net devices too.
	Workaround: N/A
	Keywords: VF; PF; virtio-net; kernel driver
	Discovered in version: 3.1.0

–	Description: It is not possible to attach block devices using the same nsid to different NVMe controllers which are linked to the same NVMe subsystem. For example, the following commands will result with an error as both controllers are attached with NSID 1:
	`snap_rpc.py controller_nvme_namespace_attach NvmeEmu2pf0 spdk Null0 1`
	`snap_rpc.py controller_nvme_namespace_attach NvmeEmu2pf1 spdk Null1 1`

	Workaround: N/A
	Keywords: Block device; controller
	Discovered in version: 3.0.0
–	Description: mlnx_snap NVMe controller supports an admin queue with a maximum size of 1024 towards the host.
	Workaround: N/A
	Keywords: Admin queue; controller
	Discovered in version: 3.0.0

Ref #	Issue
–	Description: The DPU expansion ROM includes NVMe and virtio-blk UEFI drivers certified by NVIDIA, which should be used by the BIOS. Any other BIOS drivers are not guaranteed to work properly.
	Workaround: N/A
	Keywords: BIOS; certified drivers
	Discovered in version: 3.0.0
–	Description: Legacy interrupts are not supported.
	Workaround: N/A
	Keywords: Block device; controller
	Discovered in version: 3.0.0

8.12.3.11.2.1 Windows OS Known Issues

⚠ The following are not BlueField SNAP limitations.

Ref #	Issue
3543249	Description: When using hotplugged PCIe devices, after all devices are plugged, the host must be rebooted for Windows to detect all devices.
	Workaround: N/A
	Keywords: Hotplug
	Discovered in version: 3.7.4
3521378	Description: For a successful **emulation_device_detach** RPC command, it is recommended to use **directio=1** (**O_DIRECT**) with virtio-blk controller created on hot-plugged emulation. ⚠ If **directio=0** is used, the IO must be stopped manually. Otherwise, **emulation_device_detach** may fail.
	Workaround: N/A
	Keywords: Virtio-blk; RPC
	Discovered in version: 3.7.4
2957317	Description: Setting virtio-blk emulation on bare metal will end with server crash.
	Workaround: Set the **seg_max** flag of the virtio-blk controller to at least 16 (default is 1) using the following RPC: `controller_virtio_blk_create … --seg_max 16`
	Keywords: Virtio-blk; bare-metal; seg_max
	Discovered in version: 3.7.2

Ref #	Issue
3056533	Description: When using NVMe driver in Windows, if I/O is not completed for more than 120 seconds, Windows starts ignoring the NVMe device and its disks disappear.
	Workaround: N/A
	Keywords: NVMe device disappears
	Discovered in version: 3.6.1
N/A	Description: There is a Windows driver known issue that it may crash when attaching multiple namespaces simultaneously. Users must attach namespaces one-by-one, and verify each namespace is discovered by the OS before attaching a new one.
	Workaround: N/A
	Keywords: Attaching multiple namespaces simultaneously
	Discovered in version: 3.4.0
N/A	Description: There is a known Windows NVMe driver bug which causes Windows initiators to crash if the NVMe driver is started and no target is up and ready. Therefore, if users work with Windows OS on top of the emulated NVMe device, they must make sure that mlnx_snap NVMe controller is connected to the remote target before running the driver on the host side.
	Workaround: N/A
	Keywords: Windows initiators crash
	Discovered in version: 3.1.0
N/A	Description: There is a known Windows driver bug in which namespaces hotplug is not supported. On newer Windows builds, NVMe controller quirks must be set to 0x5. For more information, please see section "Controller Parameters".
	Workaround: N/A
	Keywords: Namespaces hotplug
	Discovered in version: 3.1.0

8.12.3.11.2.2 NVMe Linux Driver Limitations

Ref #	Issue
-	Description: When using SRIOV with VFs sharing the same driver as their PF (sriov_driver_autoprobe=1) - unprobing the driver may take a long time, and some admin commands might get timeout
	Workaround: N/A
	Keywords: SRIOV; driver.
	Discovered in version: 3.8.0-8

8.12.3.11.2.3 Virtio-blk Linux Driver Limitations

Ref #	Issue
3066750	Description: Driver does not support PCIe function level reset (FLR). Running FLR during IO causes the IO (and kernel) to hang.

Ref #	Issue
	Workaround: N/A
	Keywords: PCIe function; hang
	Discovered in version: 3.6.1
2879262	Description: When working with a large number of virtqueues (≥ 64) over a single MSIX, the host kernel might experience soft lockup. Specifically, setting **--num_queues** to a high number, which is also higher than the configured **--num_msix** value, might cause this issue.
	Workaround:
	Keywords: Kernel; hang; virtqueues
	Discovered in version: 3.6.1
2957317	Description: In Linux kernel version 5.4.0-91-generic and above, the command **emulation_device_detach** times out if I/O traffic is running.
	Workaround: N/A
	Keywords: Command time out
	Discovered in version: 3.6.1

8.12.3.11.2.4 Virtio-blk Transitional Device Limitations

Ref #	Issue
3231721	Description: When using **emulation_device_attach** RPC to hot plug a virtio-blk transitional device, the capacity and block size attributes must be provided for this hot-plugged virtio-blk transitional device.
	Workaround: Use the **--bdev_type spdk** and **--bdev spdk_bdev** options to provide a bdev to the hot-plugged virtio-blk transitional device when using **emulation_device_attach** RPC.
	Keywords: Hot plugging virtio-blk transitional device
	Discovered in version: 3.7.0
–	Description: Legacy/transitional drivers do not require syncing with the device upon driver initialization. Therefore, it is highly recommended that the SNAP controller is opened on the PCIe function before the driver becomes operational. If the driver becomes operational before the controller, controller configuration options would be very limited.
	Workaround: N/A
	Keywords: Legacy; SNAP controller; SNAP driver
	Discovered in version: 3.7.0
–	Description: Legacy/transitional device support naturally includes backends with 512B block size. Using backends with any other block size (e.g., 4K) can only be achieved when SNAP controller is opened before driver is activated.
	Workaround: N/A
	Keywords: Legacy; backend block size
	Discovered in version: 3.7.0

8.13 DOCA Telemetry Service Guide

Contents:

This guide provides instructions on how to use the DOCA Telemetry Service (DTS) container on top of hosts or NVIDIA® BlueField® DPU.

8.13.1 1. Introduction

DOCA Telemetry Service (DTS) provides real-time system and workload metrics for BlueField platforms and supported host systems from built-in providers and from external telemetry applications.

Release notes are available at the bottom of the page.

8.13.1.1 1.1. Data Export Options

DTS supports the following export mechanisms:

- Prometheus Endpoint (pull)
- Fluent Bit (push)
- OpenTelemetry (push)
- Prometheus Remote Write (push)
- Storage

DTS can export telemetry collected by application utilizing the DOCA Telemetry Exporter API library.

DTS can store collected data in binary files located at **/opt/mellanox/doca/services/telemetry/data** directory. By default, data writing is disabled. Binary files can be read using the clx_read binary packaged in collectx-clxapidev linux package.

As DTS may generate a large volume of telemetry, output can be filtered using the **fieldset** or **counterset** options. For more details, see Counterset and Fieldset files.

8.13.2 2. Service Deployment

DTS is available as a built-in service on BlueField DPUs and as a container image on NVIDIA NGC for hosts.

Since DTS does not monitor all changes related to firmware, networking, firewall settings, or other system components, it is strongly recommended (and in some cases required) to restart DTS after any server configuration changes to ensure proper operation.

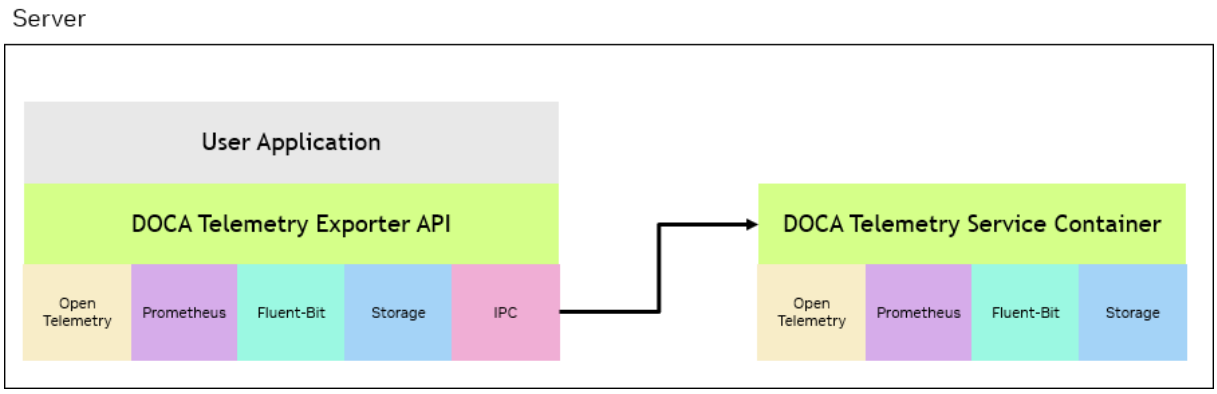
8.13.2.1 2.1. Built-in DOCA Service Image (BlueField)

DOCA Telemetry Service is included by default in the BlueField DPU image. Each DPU image ships with a fixed version of the service, providing out-of-the-box support for applications using the DOCA Telemetry Exporter library.

8.13.2.2 2.2. DOCA Telemetry Exporter Integration

Applications using the DOCA Telemetry Exporter API can leverage DTS Inter-Process Communication (IPC) to export collected telemetry data through DTS.

The export modules available in DTS are also integrated into the DOCA Telemetry Exporter. Both solutions support decoupling telemetry collection from export configuration and allow data to be exported either directly from the library (e.g., to Prometheus or OpenTelemetry) or indirectly via DTS using IPC.



8.13.2.3 2.3. DOCA Service on NGC

In addition to the built-in image, updated DTS versions are available on NVIDIA NGC. This is useful if you need to upgrade to a newer version than the one bundled with the BlueField OS. Refer to the container's NGC page for service-specific configuration and deployment steps.

For more information about the deployment of DOCA containers on top of the BlueField DPU, refer to *NVIDIA DOCA Container Deployment Guide*.

8.13.2.4 2.4. DPU Deployment

On BlueField DPUs, DTS starts automatically during boot using the configuration defined in:

```
/etc/kubelet.d/doca_telemetry_standalone.yaml
```

- To disable auto-start, remove the YAML file from this path
- Changes to the YAML file are applied on system restart

DTS file location: **/opt/mellanox/doca/services/telemetry/**

- Container volume mounts:
  - **config/**
  - **data/**
  - **ipc_sockets/**
- Backup artifacts:
  - **doca_telemetry_service_${version}_arm64.tar.gz** – DTS container image

- **doca_telemetry_standalone.yaml** – backup of the default boot YAML

8.13.2.5 2.5. Host Deployment

DTS supports deployment on **x86_64** and **aarch64** host architectures. All exporters and providers run from a single Docker container.

1. Initialize and configure host DTS with the desired DTS version:

```
export DTS_IMAGE=nvcr.io/nvidia/doca/doca_telemetry:<desired-DTS-host-version>
docker run -v "/opt/mellanox/doca/services/telemetry/config:/config" --rm --name doca-telemetry-init -it
$DTS_IMAGE /bin/bash -c "DTS_CONFIG_DIR=host /usr/bin/telemetry-init.sh"
```

2. Launch the DTS container:

```
docker run -d --net=host --uts=host --ipc=host --pid=host\
           --privileged \
           --device=/dev/mst/ \
           --gpus all \
           -v "/opt/mellanox/doca/services/telemetry/config:/config" \
           -v "/opt/mellanox/doca/services/telemetry/ipc_sockets:/tmp/ipc_sockets" \
           -v "/opt/mellanox/doca/services/telemetry/data:/data" \
           -v "/opt/mellanox/doca/services/telemetry/log:/var/log" \
           -v "/usr/lib64/mft:/usr/lib64/mft" \
           --rm --name doca-telemetry -it $DTS_IMAGE /usr/bin/telemetry-run.sh
```

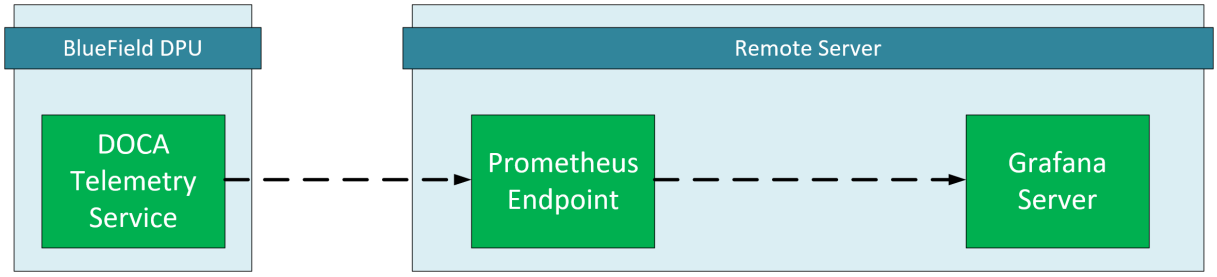
8.13.2.6 2.6. Deploying with Grafana Monitoring

This chapter provides an overview and deployment configuration of DOCA Telemetry Service with Grafana.

8.13.2.6.1 2.6.1. Grafana Deployment Prerequisites

- BlueField DPU running DOCA Telemetry Service.
- Optional remote server to host Grafana and Prometheus.
- Prometheus installed on the host machine. Please refer to the Prometheus website for more information.
- Grafana installed on the host machine. Please refer to Grafana Labs website for more information.

8.13.2.6.2 2.6.2. Grafana Deployment Configuration



8.13.2.6.2.1 2.6.2.1. DTS Configuration (DPU Side)

Configuring DTS to export the sysfs counter using the Prometheus plugin:

Sysfs is used as an example, other counters are available.

1. Make sure the sysfs counter is enabled.

```
vim /opt/mellanox/doca/services/telemetry/config/dts_config.ini
enable-provider=sysfs
```

2. Enable Prometheus exporter by setting the **prometheus** address and port.

```
vim /opt/mellanox/doca/services/telemetry/config/dts_config.ini
prometheus=http://0.0.0.0:9100
```

DTS must be restarted to apply changes.

8.13.2.6.2.2 2.6.2.2. Prometheus Configuration (Remote Server)

Please download Prometheus for your platform.

Prometheus is configured via command-line flags and a configuration file, **prometheus.yml**.

1. Open the **prometheus.yml** file and configure the DPU as the endpoint target.

```
vim prometheus.yml
# metrics_path defaults to '/metrics'
# scheme defaults to 'http'.

static_configs:
- targets: ["<dpu-ip>:<prometheus-port>"]
```

   Where:

   - **<dpu-ip>** is the DPU IP address. Prometheus reaches to this IP to pull data.
   - **<prometheus-port>** the exporter port that set in DTS configuration.

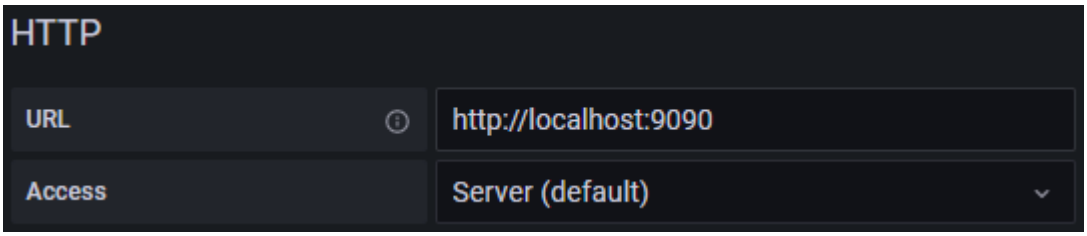
2. Run Prometheus server:

```
./prometheus --config.file="prometheus.yml"
```

8.13.2.6.2.3 2.6.2.3. Grafana Configuration (Remote Server)

Please download and install Grafana for your platform.

1. Setup Grafana. Please refer to Install Grafana guide in Grafana documentation.
2. Log into the Grafana dashboard at http://localhost:3000.
3. Add Prometheus as data source by navigating to Settings → Data sources → Add data source → Prometheus.
4. Configure the Prometheus data source. Under the HTTP section, set the Prometheus server address.

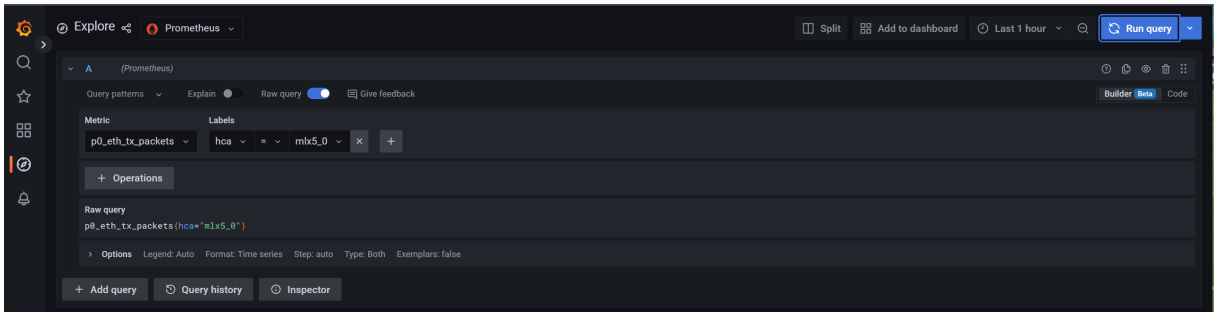


5. Save and test.

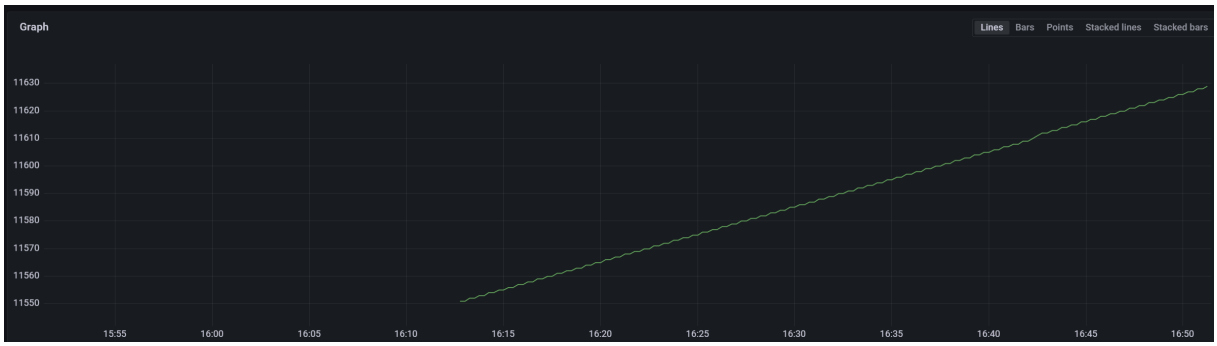
8.13.2.6.3 2.6.3. Exploring Telemetry Data

Go to the Explore page on the left-hand side, and choose a Prometheus provider.

Choose a metric to display and specify a label. The label can be used to filter out data based on the source and HCA devices.



Graph display after selecting a metric and specifying a label to filter by:



8.13.3 3. DTS Configuration

DTS configuration is located at **/opt/mellanox/doca/services/telemetry/config**.

This directory is initialized during service startup. It contains the main configuration file (**dts_config.ini**) and a folder for Fluent Bit configuration (**fluent_bit_configs**). The **dts_config.ini** file is used to control telemetry provider behavior, exporter settings, sampling rates, and data output. Fluent Bit integration is discussed separately in the Fluent Bit section.

Any changes to **dts_config.ini** automatically trigger a DTS restart within 60 seconds to apply updates.

8.13.3.1 3.1. Initialization Scripts (BlueField Deployment)

The DTS container's **initContainers** section of the **.yaml** file executes two initialization scripts:

- **/usr/bin/telemetry-init.sh** – Generates default configuration files only if the **config** directory is empty
- **/usr/bin/enable-fluent-forward.sh** – Configures the Fluent Bit forwarding destination (host and port). This script only runs when both parameters are provided and will overwrite the **fluent_bit_configs** directory with a new **.exp** configuration file.

8.13.3.2 3.2. Enabling Fluent Bit Forwarding

To enable Fluent Bit forwarding, add the destination IP and port to the command line in the **initContainers** section of your deployment YAML:

```
command: ["/bin/bash", "-c", "/usr/bin/telemetry-init.sh && /usr/bin/enable-fluent-forward.sh -i=127.0.0.1
-p=24224"]
```

The IP and port shown above are examples. See the "Fluent Bit" section for manual configuration guidance.

8.13.3.3 3.3. Generating Configuration

The **config** directory starts empty by default. The initialization process will generate default configuration files during first launch.

8.13.3.4 3.4. Resetting Configuration

To reset DTS to its default state:

1. Delete all files from **/opt/mellanox/doca/services/telemetry/config/** .
2. Restart the DTS container to regenerate default configuration files.

8.13.3.5 3.5. Sampling Interval

The sampling interval for most telemetry providers is controlled by the **update** parameter in **dts_config.ini** :

```
# Sampling interval for providers (milliseconds)
update=1000
```

Export frequency may differ due to internal buffer logic. Data is exported once a page of counters is filled. You can enforce a time-based flush using the **sync-time-limit** parameter:

```
# Timeout for forced page rotation in seconds
sync-time-limit=10000
```

8.13.3.6 3.6. Enabling Providers

To enable a telemetry provider, uncomment its **enable-provider** line in **dts_config.ini**. For example:

```
enable-provider=ethtool
```

See the "Providers" section for a full list of available telemetry providers.

8.13.3.6.1 3.6.1. Remote Collection (BlueField Only)

Some providers may not function correctly within a container and require remote execution. Enable this mode as follows:

1. Start the DOCA privileged executor (DPE):

   ```
   systemctl start dpe
   ```

2. Prefix the provider name with **grpc.**. For example, the following line configures remote collection of the **sysfs.fw_pages** component:

   ```
   enable-provider=grpc.sysfs.fw_pages
   ```

3. For provider-specific configuration, apply the same prefix. Building upon the previous example:

   ```
   grpc.sysfs.fw_pages.mlx5_1=ignore
   ```

8.13.3.7 3.7. Enabling Data Output

To enable writing collected data to disk, uncomment the following line in **dts_config.ini**:

```
output=/data
```

8.13.3.8 3.8. Enabling IPC with External Applications

To enable inter-process communication (IPC) between DTS and a non-containerized application, refer to the section "Using IPC with Non-container Application" in *DOCA Telemetry Exporter*.

IPC requires write access to the DTS socket. Since it is created by root, the client may need to run as root.

8.13.3.9 3.9. Level Labels

DTS supports metadata static labels for telemetry exporters such as Prometheus and OpenTelemetry. Labels can be configured globally or per device, and are defined in **/opt/mellanox/doca/services/telemetry/config/level_labels.ini**.

8.13.3.9.1 3.9.1. Sample `level_labels.ini`

The following is an example `level_labels.ini` file, defining global and per-device labels:

```
# lines starting with hash are ignored
# 2 level labels are supported - global and device
# global labels will be added to all metrics
# device labels will be added to metrics matching the device description

[global_labels]
# global labels to be added to all metrics
# format:
# label_name=label_value
hostname=myhost
ip=127.127.127.127

[device_labels]
# device labels will be added to metrics related to the device
# format:
# device_index_label=device_label_name1|device_label_value1|device_label_name2|device_label_value2
# where device_index_label is either an RDMA device name or PCI address
mlx5_0=rail|0|network|compute
mlx5_1=rail|1|network|compute
mlx5_2=rail|2|network|storage

[source_id_mapping]
# device label mapping for the DOCA Exporter API (metrics API and events API)
# applications using the DOCA Exporter API send data with a source_id;
# this section maps each source_id to a device label lookup:
# the system reads the index field value from the data, resolves it to a
# device via [device_labels], and attaches that device's labels
# format:
# source_id=index_label|device_type
# supported device_types: hca, netif, mst

[device_mapping]
# optional: mapping of device names to network interfaces and mst devices
# this is done internally by DTS, but can be overridden here
# add clear_auto_detected_mapping=true to clear the auto-detected mapping
# format:
# device_name=mst_device_name|net_interface_names
# in case of multiple net interfaces, separate them with comma

[data_types_mapping]
# internal data types, don't change this section if you don't know what you are doing
# format:
# data_type|index|device_type_name
ethtool_event=device_name|netif
ppcc_eth=device_name|mst
ifconfig_event=device_name|netif
amber_event=device_name|mst
pci_=device_name|hca
```

8.13.3.9.2 3.9.2. Enabling Level Labels

1. Confirm that `level_labels.ini` is properly configured.
2. Add the following line to `dts_config.ini`:

```
level-labels-file=/config/level_labels.ini
```

8.13.3.9.3 3.9.3. Monitoring and Reloading

DTS continuously monitors `level_labels.ini`. When a change is detected:

- The current telemetry process exits automatically
- DTS restarts within 60 seconds and loads the updated labels

This ensures label changes take effect without requiring manual restarts.

8.13.3.10 3.10. Device Labels

In addition to the level labels file mentioned above, DTS can attach optional **device labels** to exported metrics so downstream systems (Prometheus, OpenTelemetry) can correlate telemetry with physical hardware. Both options are **off by default**, independent of each other, and apply to all configured time-series exporters (Prometheus scrape, OpenTelemetry, Remote Write).

Configure them in **`dts_config.ini`**.

8.13.3.10.1 3.10.1. PCI Label

`add-pci-address-label` adds a **`pci`** label to metrics that are tied to a network device (for example, series that include **`hca`**, **`device_name`**, or a network interface name).

The value is the device's **PCI BDF address** (for example **`0000:21:00.0`**), discovered from the host where DTS is running. This is useful for inventory, alerting, and joining metrics across nodes when device names differ.

```
add-pci-address-label=true
```

8.13.3.10.2 3.10.2. NUMA Node Label

`add-numa-node-label` adds a **`numa_node`** label to the same device-scoped metrics. This is more relevant for Vera Rubin where DTS runs on Bluefield4 in socket direct mode.

The value is the **host NUMA index** associated with the device, resolved via DOCA API when representors are available. On a plain host without representors mapping, the label is omitted rather than causing an error.

```
add-numa-node-label=true
```

8.13.3.10.3 3.10.3. Example

With both options enabled on a DPU with active representors, a scraped Prometheus series might look like:

```
port_xmit_bytes{data_type="counters", source="node-a", hca="mlx5_0", port="1", pci="0006:01:00.0", numa_node="0"}
12345 1782397595000
```

On a host deployment with PCI labeling enabled but no NUMA mapping, you would see **`pci`** only:

```
port_xmit_bytes{data_type="counters", source="node-b", hca="mlx5_0", port="1", pci="0000:21:00.0"}  9876
1782397595000
```

These labels work alongside other DTS labeling mechanisms such as **`level-labels-file`**; they do not replace static labels you define there.

8.13.4 4. Telemetry Providers

DTS supports on-board data collection from `sysf`, `ethtool`, and `tc` providers. Fluent and Prometheus aggregator providers can collect the data from other applications.

Other providers are available based on different conditions (e.g., specific container mounts or host only such as `amber`, `ppcc_eth`, etc). Such providers are described with their dependencies in their corresponding sections.

8.13.4.1 4.1. Hardware & Network Providers

This category includes providers that interface directly with physical and virtual networking components. These tools collect low-level statistics (such as traffic volume, packet drops, and congestion control metrics) from network interfaces, eSwitches, and hardware ports.

8.13.4.1.1 4.1.1. Sysfs

The `sysfs` provider includes several components: `ib_port`, `hw_port`, `devices`, `ib_mr_cache`, `eth`, `hwmon`, and `bf_ptm`. The schema for `sysfs` counters defines the exact counters available for each component.

Sysfs provider is enabled by default on BlueField.

By default, the following components are disabled when the `sysfs` provider is enabled:

- `ib_mr_cache`
- `rate`
- `bf_ptm`

You can enable the provider and selectively disable components. For example, to disable `eth`:

```
enable-provider=sysfs
disable-provider=sysfs.eth
```

`ib_port` and `ib_hw` are state counters collected per port and are only collected when the port is in an active state.

The `hwmon` component is supported only on BlueField-2 and BlueField-3.

To filter which counters are collected, you may use a `.cset` (counterset) file. This file must not contain aliases. Place the `.cset` file in the config directory (or any accessible directory in the container), and specify the path in the configuration:

```
sysfs-cset-path=/config/path/to/my.cset
```

Classic sysfs port counters can be filtered by the virtual port number that appears in the sysfs path under `/ports/<port>/`.

Use `sysfs-allowed-ports` to collect only selected ports:

```
sysfs-allowed-ports=60,62,64-72
```

Use **sysfs-excluded-ports** to skip selected ports:

```
sysfs-excluded-ports=63,70-72
```

Sysfs counters that cannot be collected yet, or are explicitly marked as uncollected/invalid, emit **-1** by default. As it may conflict with export protocol guidelines (I.e. exporting negative value for a non-zero counter), users can emit **0** instead of **-1** by setting:

```
sysfs-invalid-counter-value=0
```

Only -1 and 0 are supported for this parameter.

8.13.4.1.1.1 4.1.1.1. Counters List

- **ib_port** counters:

```
{hca_name}:{port_num}:ib_port_state
{hca_name}:{port_num}:VL15_dropped
{hca_name}:{port_num}:excessive_buffer_overrun_errors
{hca_name}:{port_num}:link_downed
{hca_name}:{port_num}:link_error_recovery
{hca_name}:{port_num}:local_link_integrity_errors
{hca_name}:{port_num}:multicast_rcv_packets
{hca_name}:{port_num}:multicast_xmit_packets
{hca_name}:{port_num}:port_rcv_constraint_errors
{hca_name}:{port_num}:port_rcv_data
{hca_name}:{port_num}:port_rcv_errors
{hca_name}:{port_num}:port_rcv_packets
{hca_name}:{port_num}:port_rcv_remote_physical_errors
{hca_name}:{port_num}:port_rcv_switch_relay_errors
{hca_name}:{port_num}:port_xmit_constraint_errors
{hca_name}:{port_num}:port_xmit_data
{hca_name}:{port_num}:port_xmit_discards
{hca_name}:{port_num}:port_xmit_packets
{hca_name}:{port_num}:port_xmit_wait
{hca_name}:{port_num}:symbol_error
{hca_name}:{port_num}:unicast_rcv_packets
{hca_name}:{port_num}:unicast_xmit_packets
```

- **ib_hw** counters:

```
{hca_name}:{port_num}:hw_state
{hca_name}:{port_num}:hw_duplicate_request
{hca_name}:{port_num}:hw_implied_nak_seq_err
{hca_name}:{port_num}:hw_lifespan
{hca_name}:{port_num}:hw_local_ack_timeout_err
{hca_name}:{port_num}:hw_out_of_buffer
{hca_name}:{port_num}:hw_out_of_sequence
{hca_name}:{port_num}:hw_packet_seq_err
{hca_name}:{port_num}:hw_req_cqe_error
{hca_name}:{port_num}:hw_req_cqe_flush_error
{hca_name}:{port_num}:hw_req_remote_access_errors
{hca_name}:{port_num}:hw_req_remote_invalid_request
{hca_name}:{port_num}:hw_resp_cqe_error
{hca_name}:{port_num}:hw_resp_cqe_flush_error
{hca_name}:{port_num}:hw_resp_local_length_error
{hca_name}:{port_num}:hw_resp_remote_access_errors
{hca_name}:{port_num}:hw_rnr_nak_retry_err
{hca_name}:{port_num}:hw_rx_atomic_requests
{hca_name}:{port_num}:hw_rx_dct_connect
{hca_name}:{port_num}:hw_rx_icrc_encapsulated
{hca_name}:{port_num}:hw_rx_read_requests
{hca_name}:{port_num}:hw_rx_write_requests
```

- **ib_mr_cache** counters:

```
{hca_name}:mr_cache:size_{n}:cur
{hca_name}:mr_cache:size_{n}:limit
{hca_name}:mr_cache:size_{n}:miss
{hca_name}:mr_cache:size_{n}:size
```

- **eth** counters:

```
{hca_name}:{device_name}:eth_collisions
{hca_name}:{device_name}:eth_multicast
{hca_name}:{device_name}:eth_rx_bytes
{hca_name}:{device_name}:eth_rx_compressed
{hca_name}:{device_name}:eth_rx_crc_errors
{hca_name}:{device_name}:eth_rx_dropped
{hca_name}:{device_name}:eth_rx_errors
{hca_name}:{device_name}:eth_rx_fifo_errors
{hca_name}:{device_name}:eth_rx_frame_errors
{hca_name}:{device_name}:eth_rx_length_errors
{hca_name}:{device_name}:eth_rx_missed_errors
{hca_name}:{device_name}:eth_rx_nohandler
{hca_name}:{device_name}:eth_rx_over_errors
{hca_name}:{device_name}:eth_rx_packets
{hca_name}:{device_name}:eth_tx_aborted_errors
{hca_name}:{device_name}:eth_tx_bytes
{hca_name}:{device_name}:eth_tx_carrier_errors
{hca_name}:{device_name}:eth_tx_compressed
{hca_name}:{device_name}:eth_tx_dropped
{hca_name}:{device_name}:eth_tx_errors
{hca_name}:{device_name}:eth_tx_fifo_errors
{hca_name}:{device_name}:eth_tx_heartbeat_errors
{hca_name}:{device_name}:eth_tx_packets
{hca_name}:{device_name}:eth_tx_window_errors
```

- **devices** counters

```
{hca_name}:current_link_width
{hca_name}:current_link_speed
{hca_name}:max_link_speed
{hca_name}:max_link_width
```

- BlueField-2 **hwmon** counters:

```
{hwmon_name}:{l3cache}:CYCLES
{hwmon_name}:{l3cache}:HITS_BANK0
{hwmon_name}:{l3cache}:HITS_BANK1
{hwmon_name}:{l3cache}:MISSES_BANK0
{hwmon_name}:{l3cache}:MISSES_BANK1
{hwmon_name}:{pcie}:IN_C_BYTE_CNT
{hwmon_name}:{pcie}:IN_C_PKT_CNT
{hwmon_name}:{pcie}:IN_NP_BYTE_CNT
{hwmon_name}:{pcie}:IN_NP_PKT_CNT
{hwmon_name}:{pcie}:IN_P_BYTE_CNT
{hwmon_name}:{pcie}:IN_P_PKT_CNT
{hwmon_name}:{pcie}:OUT_C_BYTE_CNT
{hwmon_name}:{pcie}:OUT_C_PKT_CNT
{hwmon_name}:{pcie}:OUT_NP_BYTE_CNT
{hwmon_name}:{pcie}:OUT_NP_PKT_CNT
{hwmon_name}:{pcie}:OUT_P_PKT_CNT
{hwmon_name}:{tile}:MEMORY_READS
{hwmon_name}:{tile}:MEMORY_WRITES
{hwmon_name}:{tile}:MSS_NO_CREDIT
{hwmon_name}:{tile}:VICTIM_WRITE
{hwmon_name}:{tilenet}:CDN_DIAG_C_OUT_OF_CRED
{hwmon_name}:{tilenet}:CDN_REQ
{hwmon_name}:{tilenet}:DDN_REQ
{hwmon_name}:{tilenet}:NDN_REQ
{hwmon_name}:{trio}:TDMA_DATA_BEAT
{hwmon_name}:{trio}:TDMA_PBUF_MAC_AF
{hwmon_name}:{trio}:TDMA_RT_AF
{hwmon_name}:{trio}:TPIO_DATA_BEAT
{hwmon_name}:{triogen}:TX_DAT_AF
{hwmon_name}:{triogen}:TX_DAT_AF
```

- BlueField-3 **hwmon** counters:

```
{hwmon_name}:{llt}:GDC_BANK0_RD_REQ
{hwmon_name}:{llt}:GDC_BANK1_RD_REQ
{hwmon_name}:{llt}:GDC_BANK0_WR_REQ
{hwmon_name}:{llt}:GDC_BANK1_WR_REQ
{hwmon_name}:{llt_miss}:GDC_MISS_MACHINE_RD_REQ
{hwmon_name}:{llt_miss}:GDC_MISS_MACHINE_WR_REQ
{hwmon_name}:{mss}:SKYLIB_DDN_TX_FLITS
{hwmon_name}:{mss}:SKYLIB_DDN_RX_FLITS
```

- BlueField-3 **bf_ptm** counters:

```
bf:ptm:active_power_profile
bf:ptm:atx_power_available
bf:ptm:core_temp
bf:ptm:ddr_temp
bf:ptm:error_state
bf:ptm:power_envelope
bf:ptm:power_throttling_event_count
bf:ptm:power_throttling_state
bf:ptm:thermal_throttling_event_count
bf:ptm:thermal_throttling_state
```

```
bf:ptm:throttling_state
bf:ptm:total_power
bf:ptm:vr0_power
bf:ptm:vr1_power
```

- **rate** counters – calculated counters showing the rate of raw counters of other components. Such counters are identified by a **_rate** suffix which correspond to the original raw counter name they track.

8.13.4.1.1.2 4.1.1.2. Port Counters

The following parameters are located in **/sys/class/infiniband/mlx5_0/ports/1/counters**.

Counter	Description	InfiniBand Spec Name	Group
port_rcv_data	The total number of data octets, divided by 4, (counting in double words, 32 bits), received on all VLs from the port.	**PortRcvData**	Informative
port_rcv_packets	Total number of packets (this may include packets containing Errors. This is 64 bit counter.	**PortRcvPkts**	Informative
port_multicast_rcv_packets	Total number of multicast packets, including multicast packets containing errors.	**PortMultiCastRcvPkts**	Informative
port_unicast_rcv_packets	Total number of unicast packets, including unicast packets containing errors.	**PortUnicastRcvPkts**	Informative
port_xmit_data	The total number of data octets, divided by 4, (counting in double words, 32 bits), transmitted on all VLs from the port.	**PortXmitData**	Informative
port_xmit_packets **port_xmit_packets_64**	Total number of packets transmitted on all VLs from this port. This may include packets with errors. This is 64 bit counter.	**PortXmitPkts**	Informative
port_rcv_switch_relay_errors	Total number of packets received on the port that were discarded because they could not be forwarded by the switch relay.	**PortRcvSwitchRelayErrors**	Error
port_rcv_errors	Total number of packets containing an error that were received on the port.	**PortRcvErrors**	Informative
port_rcv_constraint_errors	Total number of packets received on the switch physical port and are discarded.	**PortRcvConstraintErrors**	Error
local_link_integrity_errors	The number of times that the count of local physical errors exceeded the threshold specified by **LocalPhyErrors**.	**LocalLinkIntegrityErrors**	Error
port_xmit_wait	The number of ticks during which the port had data to transmit but no data was sent during the entire tick (either because of insufficient credits or because of lack of arbitration).	**PortXmitWait**	Informative
port_multicast_xmit_packets	Total number of multicast packets transmitted on all VLs from the port. This may include multicast packets with errors.	**PortMultiCastXmitPkts**	Informative

Counter	Description	InfiniBand Spec Name	Group
port_unicast_xmit_packets	Total number of unicast packets transmitted on all VLs from the port. This may include unicast packets with errors.	PortUnicastXmitPkts	Informative
port_xmit_discards	Total number of outbound packets discarded by the port because the port is down or congested.	PortXmitDiscards	Error
port_xmit_constraint_errors	Total number of packets not transmitted from the switch physical port.	PortXmitConstraintErrors	Error
port_rcv_remote_physical_errors	Total number of packets marked with the EBP delimiter received on the port.	PortRcvRemotePhysicalErrors	Error
symbol_error	Total number of minor link errors detected on one or more physical lanes.	SymbolErrorCounter	Error
VL15_dropped	Number of incoming VL15 packets dropped due to resource limitations (e.g., lack of buffers) of the port.	VL15Dropped	Error
link_error_recovery	Total number of times the Port Training state machine has successfully completed the link error recovery process.	LinkErrorRecoveryCounter	Error
link_downed	Total number of times the Port Training state machine has failed the link error recovery process and downed the link.	LinkDownedCounter	Error

8.13.4.1.1.3 4.1.1.3. Hardware Counters

The hardware counters, found under `/sys/class/infiniband/mlx5_0/ports/1/hw_counters/`, are counted per function and exposed on the function. Some counters are not counted per function. These counters are commented with a relevant comment.

Counter	Description	Group
duplicate_request	Number of received packets. A duplicate request is a request that had been previously executed.	Error
implied_nak_seq_err	Number of times the requested decided an ACK. with a PSN larger than the expected PSN for an RDMA read or response.	Error
lifespan	The maximum period in ms which defines the aging of the counter reads. Two consecutive reads within this period might return the same values	Informative
local_ack_timeout_err	The number of times QP's ack timer expired for RC, XRC, DCT QPs at the sender side. The QP retry limit was not exceeded, therefore it is still recoverable error.	Error
np_cnp_sent	The number of CNP packets sent by the Notification Point when it noticed congestion experienced in the RoCEv2 IP header (ECN bits).	Informative

Counter	Description	Group
np_ecn_marked_roce_packets	The number of RoCEv2 packets received by the notification point which were marked for experiencing the congestion (ECN bits where '11' on the ingress RoCE traffic) .	Informative
out_of_buffer	The number of drops occurred due to lack of WQE for the associated QPs.	Error
out_of_sequence	The number of out of sequence packets received.	Error
packet_seq_err	The number of NAK sequence error packets received. The QP retry limit was not exceeded.	Error
req_cqe_error	The number of times requester detected CQEs completed with errors.	Error
req_cqe_flush_error	The number of times requester detected CQEs completed with flushed errors.	Error
req_remote_access_errors	The number of times requester detected remote access errors.	Error
req_remote_invalid_request	The number of times requester detected remote invalid request errors.	Error
resp_cqe_error	The number of times responder detected CQEs completed with errors.	Error
resp_cqe_flush_error	The number of times responder detected CQEs completed with flushed errors.	Error
resp_local_length_error	The number of times responder detected local length errors.	Error
resp_remote_access_errors	The number of times responder detected remote access errors.	Error
rnr_nak_retry_err	The number of received RNR NAK packets. The QP retry limit was not exceeded.	Error
rp_cnp_handled	The number of CNP packets handled by the Reaction Point HCA to throttle the transmission rate.	Informative
rp_cnp_ignored	The number of CNP packets received and ignored by the Reaction Point HCA. This counter should not raise if RoCE Congestion Control was enabled in the network. If this counter raise, verify that ECN was enabled on the adapter. See HowTo Configure DCQCN (RoCE CC) values for ConnectX-4 (Linux).	Error

Counter	Description	Group
rx_atomic_requests	The number of received ATOMIC request for the associated QPs.	Informative
rx_dct_connect	The number of received connection request for the associated DCTs.	Informative
rx_read_requests	The number of received READ requests for the associated QPs.	Informative
rx_write_requests	The number of received WRITE requests for the associated QPs.	Informative
rx_icrc_encapsulated	The number of RoCE packets with ICRC errors.	Error
roce_adp_retrans	Counts the number of adaptive retransmissions for RoCE traffic	Informative
roce_adp_retrans_to	Counts the number of times RoCE traffic reached timeout due to adaptive retransmission	Informative
roce_slow_restart	Counts the number of times RoCE slow restart was used	Informative
roce_slow_restart_cnps	Counts the number of times RoCE slow restart generated CNP packets	Informative
roce_slow_restart_trans	Counts the number of times RoCE slow restart changed state to slow restart	Informative
roce_slow_restart	Counts the number of times RoCE traffic reached timeout due to adaptive retransmission	Informative

8.13.4.1.1.4 4.1.1.4. Debug Status Counters

The following parameters are located in **/sys/class/net/<interface>/debug** .

Parameter	Description	Default
lro_timeout	Sets the LRO timer period value in usecs which will be used as LRO session expiration time. For example: # cat /sys/class/net/eth2/debug/lro_timeout Actual timeout: 32 Supported timeout: 8 16 32 1024 | 32 |	
link_down_reason	Link down reason will allow the user to query the reason which is preventing the link from going up. For example: $ cat /sys/class/net/ethXX/debug/link_down_reason	

monitor_opcode: 0x0
status_message: The port is Active. Refer to *NVIDIA's Adapters Programmer's Reference Manual (PRM)* for all possible options (PDDR register). | N/A |

8.13.4.1.1.5 4.1.1.5. Power Thermal Counters

The **bf_ptm** component collects BlueField-3 power thermal counters using remote collection. It is disabled by default and can be enabled as follows:

1. Load kernel module **mlxbf-ptm**:

```
modprobe -v mlxbf-ptm
```

2. Enable component using remote collection:

```
enable-provider=grpc.sysfs.bf_ptm
```

8.13.4.1.2 4.1.2. Ethtool Counters

Ethtool counters provide a comprehensive list of hardware and software statistics generated by the Ethtool utility on a per-device basis. These counters are essential for monitoring traffic flow, offload efficiency, and identifying network or device errors.

8.13.4.1.2.1 4.1.2.1. Counter Groups

Counters are categorized by their origin location within the datapath:

- Ring: Software ring counters.
- Software Port: An aggregation of software ring counters.
- vPort: Traffic and drop counters related to internal steering or buffer availability.
  - Troubleshooting: Errors here often indicate issues internal to the BlueField DPU.
  - Scope: Includes standard Ethernet traffic (including raw) and RDMA/RoCE traffic.
- Physical Port: Counters for the physical port connecting the BlueField DPU to the network.
  - Troubleshooting: Errors here usually indicate physical link issues (cable/transceiver) or external network switch issues.
  - Standards: Includes IEEE 802.3, RFC 2863, RFC 2819, RFC 3635, flow control, and FEC counters.
  - Note: Physical port counters are not exposed to virtual machines.
- Priority Port: A subset of physical port counters, broken down per priority level per port.

8.13.4.1.2.2 4.1.2.2. Counter Types

Within each group, counters are classified by their function:

- Traffic Informative: General traffic volume tracking used for load estimation and debugging.
- Traffic Acceleration [A]: Tracks traffic processed by NVIDIA hardware offloads or drivers.
- Error Counters: Tracks dropped packets or malformed frames. Increments in these counters typically indicate a problem.

8.13.4.1.2.3 4.1.2.3. Acceleration Mechanism Counters

Specific hardware offload mechanisms have dedicated counters to verify efficiency:

- TCP Segmentation Offload (TSO): Tracks outbound throughput improvements where the kernel buffers multiple packets into a single large buffer, which the BlueField hardware then splits into individual packets.
- Large Receive Offload (LRO): Tracks inbound throughput improvements where multiple incoming packets of a single stream are aggregated into a single buffer.
- CQE Compress: Tracks the compression of Completion Queue Events (CQE), used to spare PCIe bandwidth and improve performance.
- CHECKSUM: Tracks TCP checksum calculations performed by the BlueField hardware. Statuses include:
  - **CHECKSUM_UNNECESSARY** : No checksum needed.
  - **CHECKSUM_NONE** : No acceleration was used.
  - **CHECKSUM_COMPLETE** : Device verified the checksum for the entire packet.
  - **CHECKSUM_PARTIAL** : Device provided partial checksum validation.

8.13.4.1.2.4 4.1.2.4. Configuration Parameters

To configure which interfaces are monitored and what data is collected, use the following parameters in your configuration file.

4.1.2.4.1. Interface Selection

Define which network interfaces to monitor using a comma-separated list.

```
ethtool-interfaces=enp1s0f0,enp1s0f1,enp1s0f2
```

By default, the system uses "contains" matching (any interface name containing the string is matched). To force an exact match, set:

```
ethtool-interfaces-match-method=exact
```

4.1.2.4.2. Fieldset Customization (Filtering)

To customize exactly which fields are included in the collected data, point to a fieldset configuration file (**.fset**).

```
ethtool-fieldset-file=/config/ethtool.fset
```

The file should reside in the DTS config folder. This example filters for Receive (Rx) and Transmit (Tx) pause control counters:

```
[ethtool_event_*]
rx_pause_ctrl_phy
tx_pause_ctrl_phy
```

8.13.4.1.2.5 4.1.2.5. Ring/Software Port Counters

The following counters are available per ring or software port.

These counters provide information on the amount of traffic accelerated by the BlueField. The counters tally the accelerated traffic in addition to the standard counters which tally that (i.e. accelerated traffic is counted twice).

The counter names in the table below refers to both ring and port counters. the notation for ring counters includes the `[i]` index without the braces. the notation for port counters does not include the `[i]` . a counter name `rx[i]_packets` will be printed as `rx0_packets` for ring 0 and `rx_packets` for the software port

Counter	Description	Type
`rx[i]_packets`	The number of packets received on ring i.	Informative
`rx[i]_bytes`	The number of bytes received on ring i.	Informative
`tx[i]_packets`	The number of packets transmitted on ring i.	Informative
`tx[i]_bytes`	The number of bytes transmitted on ring i.	Informative
`tx[i]_tso_packets`	The number of TSO packets transmitted on ring i [A].	Acceleration
`tx[i]_tso_bytes`	The number of TSO bytes transmitted on ring i [A].	Acceleration
`tx[i]_tso_inner_packets`	The number of TSO packets which are indicated to be carry internal encapsulation transmitted on ring i [A]	Acceleration
`tx[i]_tso_inner_bytes`	The number of TSO bytes which are indicated to be carry internal encapsulation transmitted on ring i [A].	Acceleration
`rx[i]_lro_packets`	The number of LRO packets received on ring i [A].	Acceleration
`rx[i]_lro_bytes`	The number of LRO bytes received on ring i [A].	Acceleration
`rx[i]_csum_unnecessary`	Packets received with a `CHECKSUM_UNNECESSARY` on ring i [A].	Acceleration

Counter	Description	Type
rx[i]_csum_none	Packets received with CHECKSUM_NONE on ring i [A].	Acceleration
rx[i]_csum_complete	Packets received with a CHECKSUM_COMPLETE on ring i [A].	Acceleration
rx[i]_csum_unnecessary_inner	Packets received with inner encapsulation with a CHECK_SUM UNNECESSARY on ring i [A].	Acceleration
tx[i]_csum_partial	Packets transmitted with a CHECKSUM_PARTIAL on ring i [A].	Acceleration
tx[i]_csum_partial_inner	Packets transmitted with inner encapsulation with a CHECKSUM_PARTIAL on ring i [A].	Acceleration
tx[i]_csum_none	Packets transmitted with no hardware checksum acceleration on ring i.	Informative
tx[i]_stopped tx_queue_stopped	Events where SQ was full on ring i. If this counter is increased, check the amount of buffers allocated for transmission.	Error
tx[i]_wake tx_queue_wake	Events where SQ was full and has become not full on ring i.	Error
tx[i]_dropped tx_queue_dropped	Packets transmitted that were dropped due to DMA mapping failure on ring i. If this counter is increased, check the amount of buffers allocated for transmission.	Error
rx[i]_wqe_err	The number of wrong opcodes received on ring i.	Error
tx[i]_nop	The number of no WQEs (empty WQEs) inserted to the SQ (related to ring i) due to the reach of the end of the cyclic buffer. When reaching near to the end of cyclic buffer the driver may add those empty WQEs to avoid handling a state the a WQE start in the end of the queue and ends in the beginning of the queue. This is a normal condition.	Informative
rx[i]_mpwqe_frag	The number of WQEs that failed to allocate compound page and hence fragmented MPWQE's (multipacket WQEs) were used on ring i. If this counter raise, it may suggest that there is no enough memory for large pages, the driver allocated fragmented pages. This is not abnormal condition.	Informative
rx[i]_mpwqe_filler_cqes	The number of filler CQEs events that where issued on ring i.	Informative

Counter	Description	Type
rx[i]_cqe_compress_blks	The number of receive blocks with CQE compression on ring i [A].	Acceleration
rx[i]_cqe_compress_pkts	The number of receive packets with CQE compression on ring i [A].	Acceleration
rx[i]_cache_reuse	The number of events of successful reuse of a page from a driver's internal page cache	Acceleration
rx[i]_cache_full	The number of events of full internal page cache where driver can't put a page back to the cache for recycling (page will be freed)	Acceleration
rx[i]_cache_empty	The number of events where cache was empty - no page to give. driver shall allocate new page	Acceleration
rx[i]_cache_busy	The number of events where cache head was busy and cannot be recycled. driver allocated new page	Acceleration
rx[i]_xmit_more	The number of packets sent with xmit_more indication set on the skbuff (no doorbell)	Acceleration
tx[i]_cqes	The number of completions received on the CQ of TX ring.	Informative
ch[i]_poll	The number of invocations of NAPI poll of channel.	Informative
ch[i]_arm	The number of times the NAPI poll function completed and armed the completion queues on channel	Informative
ch[i]_aff_change	The number of times the NAPI poll function explicitly stopped execution on a CPU due to a change in affinity, on channel.	Informative
rx[i]_congst_umr	The number of times an outstanding UMR request is delayed due to congestion, on ring.	Error
ch[i]_events	The number of hard interrupt events on the completion queues of channel.	Informative
rx[i]_mpwqe_filler_strides	The number of strides consumed by filler CQEs on ring.	Informative
rx[i]_xdp_tx_xmit	The number of packets forwarded back to the port due to XDP program XDP_TX action (bouncing). these packets are not counted by other software counters. These packets are counted by physical port and vPort counters.	Informative

Counter	Description	Type
rx[i]_xdp_tx_full	The number of packets that should have been forwarded back to the port due to XDP_TX action but were dropped due to full tx queue. these packets are not counted by other software counters. These packets are counted by physical port and vPort counters You may open more rx queues and spread traffic rx over all queues and/or increase rx ring size.	Error
rx[i]_xdp_tx_err	The number of times an XDP_TX error such as frame too long and frame too short occurred on XDP_TX ring of RX ring.	Error
rx[i]_xdp_tx_cqes rx_xdp_tx_cqe	The number of completions received on the CQ of the XDP-TX ring.	Informative
rx[i]_xdp_drop	The number of packets dropped due to XDP program XDP_DROP action. these packets are not counted by other software counters. These packets are counted by physical port and vPort counters.	Informative
rx[i]_xdp_redirect	The number of times an XDP redirect action has been triggered on ring.	Acceleration
tx[i]_xdp_xmit	The number of packets redirected to the interface (due to XDP redirect). These packets are not counted by other software counters. These packets are counted by physical port and vPort counters.	Informative
tx[i]_xdp_full	The number of packets redirected to the interface (due to XDP redirect) but were dropped due to the Tx queue being full. These packets are not counted by other software counters. Users may enlarge Tx queues.	Informative
tx[i]_xdp_err	The number of packets redirected to the interface (due to XDP redirect) but were dropped due to an error (e.g., frame too long and frame too short).	Error
tx[i]_xdp_cqes	The number of completions received for packets redirected to the interface (due to XDP redirect) on the CQ.	Informative
rx[i]_cache_waive	The number of cache evacuation. This can occur due to page move to another NUMA node or page was pfmemalloc-ed and should be freed as soon as possible.	Acceleration

8.13.4.1.2.6 4.1.2.6. vPort Counters

Counters on the eswitch port that is connected to the vNIC.

Counter	Description	Type
rx_vport_unicast_packets	Unicast packets received, steered to a port including raw Ethernet QP/ DPDK traffic, excluding RDMA traffic	Informative
rx_vport_unicast_bytes	Unicast bytes received, steered to a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
tx_vport_unicast_packets	Unicast packets transmitted, steered from a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative

Counter	Description	Type
tx_vport_unicast_bytes	Unicast bytes transmitted, steered from a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
rx_vport_multicast_packets	Multicast packets received, steered to a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
rx_vport_multicast_bytes	Multicast bytes received, steered to a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
tx_vport_multicast_packets	Multicast packets transmitted, steered from a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
tx_vport_multicast_bytes	Multicast bytes transmitted, steered from a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
rx_vport_broadcast_packets	Broadcast packets received, steered to a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
rx_vport_broadcast_bytes	Broadcast bytes received, steered to a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
tx_vport_broadcast_packets	Broadcast packets transmitted, steered from a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
tx_vport_broadcast_bytes	Broadcast packets transmitted, steered from a port including raw Ethernet QP/DPDK traffic, excluding RDMA traffic	Informative
rx_vport_rdma_unicast_packets	RDMA unicast packets received, steered to a port (counters counts RoCE/UD/RC traffic) [A]	Acceleration
rx_vport_rdma_unicast_bytes	RDMA unicast bytes received, steered to a port (counters counts RoCE/UD/RC traffic) [A]	Acceleration
tx_vport_rdma_unicast_packets	RDMA unicast packets transmitted, steered from a port (counters counts RoCE/UD/RC traffic) [A]	Acceleration
tx_vport_rdma_unicast_bytes	RDMA unicast bytes transmitted, steered from a port (counters counts RoCE/UD/RC traffic) [A]	Acceleration
rx_vport_ rdma_multicast_packets	RDMA multicast packets received, steered to a port (counters counts RoCE/UD/RC traffic) [A]	Acceleration
rx_vport_ rdma_multicast_bytes	RDMA multicast bytes received, steered to a port (counters counts RoCE/UD/RC traffic) [A]	Acceleration
tx_vport_ rdma_multicast_packets	RDMA multicast packets transmitted, steered from a port (counters counts RoCE/UD/RC traffic) [A]	Acceleration
tx_vport_ rdma_multicast_bytes	RDMA multicast bytes transmitted, steered from a port (counters counts RoCE/UD/RC traffic) [A]	Acceleration
rx_steer_missed_packets	Number of packets received by the NIC but discarded due to not matching any flow in the NIC flow table.	Error

Counter	Description	Type
rx_packets	Representor only: packets received, that were handled by the hypervisor.	Informative
rx_bytes	Representor only: bytes received, that were handled by the hypervisor.	Informative
tx_packets	Representor only: packets transmitted which have been handled by the hypervisor.	Informative
tx_bytes	Representor only: bytes transmitted which have been handled by the hypervisor.	Informative

8.13.4.1.2.7 4.1.2.7. Physical Port Counters

The physical port counters are the counters on the external port connecting adapter to the network. This measuring point holds information on standardized counters like IEEE 802.3, RFC2863, RFC 2819, RFC 3635 and additional counters like flow control, FEC and more.

Counter	Description	Type
rx_packets_phy	The number of packets received on the physical port. This counter doesn't include packets that were discarded due to FCS, frame size and similar errors.	Informative
tx_packets_phy	The number of packets transmitted on the physical port.	Informative
rx_bytes_phy	The number of bytes received on the physical port, including Ethernet header and FCS.	Informative
tx_bytes_phy	The number of bytes transmitted on the physical port.	Informative
rx_multicast_phy	The number of multicast packets received on the physical port.	Informative
tx_multicast_phy	The number of multicast packets transmitted on the physical port.	Informative
rx_broadcast_phy	The number of broadcast packets received on the physical port.	Informative
tx_broadcast_phy	The number of broadcast packets transmitted on the physical port.	Informative
rx_crc_errors_phy	The number of dropped received packets due to frame check sequence (FCS) error on the physical port. If this counter is increased in high rate, check the link quality using rx_symbol_error_phy and rx_corrected_bits_phy counters below.	Error
rx_in_range_len_errors_phy	The number of received packets dropped due to length/type errors on a physical port.	Error

Counter	Description	Type
rx_out_of_range_len_phy	The number of received packets dropped due to length greater than allowed on a physical port. If this counter is increasing, it implies that the peer connected to the adapter has a larger MTU configured. Using same MTU configuration shall resolve this issue.	Error
rx_oversize_pkts_phy	The number of dropped received packets due to length which exceed MTU size on a physical port. If this counter is increasing, it implies that the peer connected to the adapter has a larger MTU configured. Using same MTU configuration shall resolve this issue.	Error
rx_symbol_err_phy	The number of received packets dropped due to physical coding errors (symbol errors) on a physical port.	Error
rx_mac_control_phy	The number of MAC control packets received on the physical port.	Informative
tx_mac_control_phy	The number of MAC control packets transmitted on the physical port.	Informative
rx_pause_ctrl_phy	The number of link layer pause packets received on a physical port. If this counter is increasing, it implies that the network is congested and cannot absorb the traffic coming from to the adapter.	Informative
tx_pause_ctrl_phy	The number of link layer pause packets transmitted on a physical port. If this counter is increasing, it implies that the NIC is congested and cannot absorb the traffic coming from the network.	Informative
rx_unsupported_op_phy	The number of MAC control packets received with unsupported opcode on a physical port.	Error
rx_discards_phy	The number of received packets dropped due to lack of buffers on a physical port. If this counter is increasing, it implies that the adapter is congested and cannot absorb the traffic coming from the network.	Error
tx_discards_phy	The number of packets which were discarded on transmission, even no errors were detected. the drop might occur due to link in down state, head of line drop, pause from the network, etc.	Error
tx_errors_phy	The number of transmitted packets dropped due to a length which exceed MTU size on a physical port.	Error
rx_undersize_pkts_phy	The number of received packets dropped due to length which is shorter than 64 bytes on a physical port. If this counter is increasing, it implies that the peer connected to the adapter has a non-standard MTU configured or malformed packet had arrived.	Error
rx_fragments_phy	The number of received packets dropped due to a length which is shorter than 64 bytes and has FCS error on a physical port. If this counter is increasing, it implies that the peer connected to the adapter has a non-standard MTU configured.	Error
rx_jabbers_phy	The number of received packets d due to a length which is longer than 64 bytes and had FCS error on a physical port.	Error
rx_64_bytes_phy	The number of packets received on the physical port with size of 64 bytes.	Informative

Counter	Description	Type
rx_65_to_127_bytes_phy	The number of packets received on the physical port with size of 65 to 127 bytes.	Informative
rx_128_to_255_bytes_phy	The number of packets received on the physical port with size of 128 to 255 bytes.	Informative
rx_256_to_511_bytes_phy	The number of packets received on the physical port with size of 256 to 512 bytes.	Informative
rx_512_to_1023_bytes_phy	The number of packets received on the physical port with size of 512 to 1023 bytes.	Informative
rx_1024_to_1518_bytes_phy	The number of packets received on the physical port with size of 1024 to 1518 bytes.	Informative
rx_1519_to_2047_bytes_phy	The number of packets received on the physical port with size of 1519 to 2047 bytes.	Informative
rx_2048_to_4095_bytes_phy	The number of packets received on the physical port with size of 2048 to 4095 bytes.	Informative
rx_4096_to_8191_bytes_phy	The number of packets received on the physical port with size of 4096 to 8191 bytes.	Informative
rx_8192_to_10239_bytes_phy	The number of packets received on the physical port with size of 8192 to 10239 bytes.	Informative
link_down_events_phy	The number of times where the link operative state changed to down. In case this counter is increasing it may imply on port flapping. You may need to replace the cable/transceiver.	Error
rx_out_of_buffer	Number of times receive queue had no software buffers allocated for the adapter's incoming traffic.	Error
module_bus_stuck	The number of times that module's I2C bus (data or clock) short-wire was detected. You may need to replace the cable/transceiver.	Error
module_high_temp	The number of times that the module temperature was too high. If this issue persists, you may need to check the ambient temperature or replace the cable/transceiver module.	Error

Counter	Description	Type
module_bad_shorted	The number of times that the module cables were shorted. You may need to replace the cable/transceiver module.	Error
module_unplug	The number of times that module was ejected.	Informative
rx_buffer_passed_thres_phy	The number of events where the port receive buffer was over 85% full.	Informative
tx_pause_storm_warning_events	The number of times the device was sending pauses for a long period of time.	Informative
tx_pause_storm_error_events	The number of times the device was sending pauses for a long period of time, reaching time out and disabling transmission of pause frames. on the period where pause frames were disabled, drop could have been occurred.	Error
rx[i]_buff_alloc_err / rx_buff_alloc_err	Failed to allocate a buffer to received packet (or SKB) on port (or per ring)	Error
rx_bits_phy	This counter provides information on the total amount of traffic that could have been received and can be used as a guideline to measure the ratio of errored traffic in rx_pcs_symbol_err_phy and rx_corrected_bits_phy.	Informative
rx_pcs_symbol_err_phy	This counter counts the number of symbol errors that wasn't corrected by FEC correction algorithm or that FEC algorithm was not active on this interface. If this counter is increasing, it implies that the link between the NIC and the network is suffering from high BER, and that traffic is lost. You may need to replace the cable/transceiver. The error rate is the number of rx_pcs_symbol_err_phy divided by the number of rx_phy_bits on a specific time frame.	Error
rx_corrected_bits_phy	The number of corrected bits on this port according to active FEC (RS/FC). If this counter is increasing, it implies that the link between the NIC and the network is suffering from high BER. The corrected bit rate is the number of rx_corrected_bits_phy divided by the number of rx_phy_bits on a specific time frame	Error
phy_raw_errors_lane[l]	This counter counts the number of physical raw errors per lane [l] index. The counter counts errors before FEC corrections. If this counter is increasing, it implies that the link between the NIC and the network is suffering from high BER, and that traffic might be lost. You may need to replace the cable/transceiver. Please check in accordance with rx_corrected_bits_phy.	Error

8.13.4.1.2.8 4.1.2.8. Priority Port Counters

The following counters are physical port counters that being counted per L2 priority (0-7).

`p` in the counter name represents the priority.

Counter	Description	Type
`rx_prio[p]_bytes`	The number of bytes received with priority p on the physical port.	Informative
`rx_prio[p]_packets`	The number of packets received with priority p on the physical port.	Informative
`tx_prio[p]_bytes`	The number of bytes transmitted on priority p on the physical port.	Informative
`tx_prio[p]_packets`	The number of packets transmitted on priority p on the physical port.	Informative
`rx_prio[p]_pause`	The number of pause packets received with priority p on a physical port. If this counter is increasing, it implies that the network is congested and cannot absorb the traffic coming from the adapter.	Informative
`rx_prio[p]_pause_duration`	The duration of pause received (in microSec) on priority p on the physical port. The counter represents the time the port did not send any traffic on this priority. If this counter is increasing, it implies that the network is congested and cannot absorb the traffic coming from the adapter.	Informative
`rx_prio[p]_pause_transition`	The number of times a transition from Xoff to Xon on priority p on the physical port has occurred.	Informative
`tx_prio[p]_pause`	The number of pause packets transmitted on priority p on a physical port. If this counter is increasing, it implies that the adapter is congested and cannot absorb the traffic coming from the network.	Informative
`tx_prio[p]_pause_duration`	The duration of pause transmitter (in microSec) on priority p on the physical port.	Informative
`rx_prio[p]_buf_discard`	The number of packets discarded by device due to lack of per host receive buffers.	Informative
`rx_prio[p]_cong_discard`	The number of packets discarded by device due to per host congestion.	Informative
`rx_prio[p]_marked`	The number of packets ecn marked by device due to per host congestion.	Informative
`rx_prio[p]_discard`	The number of packets discarded by device due to lack of receive buffers.	Informative

8.13.4.1.2.9 4.1.2.9. Device Counters

Counter	Description	Type
rx_pci_signal_integrity	Counts physical layer PCIe signal integrity errors, the number of transitions to recovery due to Framing errors and CRC (dlp and tlp). If this counter is raising, try moving the adapter card to a different slot to rule out a bad PCIe slot. Validate that you are running with the latest firmware available and latest server BIOS version.	Error
tx_pci_signal_integrity	Counts physical layer PCIe signal integrity errors, the number of transition to recovery initiated by the other side (moving to recovery due to getting TS/EIEOS). If this counter is raising, try moving the adapter card to a different slot to rule out a bad PCI slot. Validate that you are running with the latest firmware available and latest server BIOS version.	Error
outbound_pci_buffer_overflow	The number of packets dropped due to pci buffer overflow. If this counter is raising in high rate, it might indicate that the receive traffic rate for a host is larger than the PCIe bus and therefore a congestion occurs.	Informative
outbound_pci_stalled_rd	The percentage (in the range 0...100) of time within the last second that the NIC had outbound non-posted reads requests but could not perform the operation due to insufficient posted credits.	Informative
outbound_pci_stalled_wr	The percentage (in the range 0...100) of time within the last second that the NIC had outbound posted writes requests but could not perform the operation due to insufficient posted credits.	Informative
outbound_pci_stalled_rd_events	The number of seconds where outbound_pci_stalled_rd was above 30%.	Informative
outbound_pci_stalled_wr_events	The number of seconds where outbound_pci_stalled_wr was above 30%.	Informative
dev_out_of_buffer	The number of times the device owned queue had not enough buffers allocated.	Error

8.13.4.1.2.10 4.1.2.10. Full List of Counters

```
# ethtool -S eth5

NIC statistics:
rx_packets: 10
rx_bytes: 3420
tx_packets: 18
tx_bytes: 1296
tx_tso_packets: 0
tx_tso_bytes: 0
tx_tso_inner_packets: 0
tx_tso_inner_bytes: 0
tx_added_vlan_packets: 0
tx_nop: 0
rx_lro_packets: 0
rx_lro_bytes: 0
rx_ecn_mark: 0
rx_removed_vlan_packets: 0
```

```
rx_csum_unnecessary: 0
rx_csum_none: 0
rx_csum_complete: 10
rx_csum_unnecessary_inner: 0
rx_xdp_drop: 0
rx_xdp_redirect: 0
rx_xdp_tx_xmit: 0
rx_xdp_tx_full: 0
rx_xdp_tx_err: 0
rx_xdp_tx_cqe: 0
tx_csum_none: 18
tx_csum_partial: 0
tx_csum_partial_inner: 0
tx_queue_stopped: 0
tx_queue_dropped: 0
tx_xmit_more: 0
tx_recover: 0
tx_cqes: 18
tx_queue_wake: 0
tx_udp_seg_rem: 0
tx_cqe_err: 0
tx_xdp_xmit: 0
tx_xdp_full: 0
tx_xdp_err: 0
tx_xdp_cqes: 0
rx_wqe_err: 0
rx_mpwqe_filler_cqes: 0
rx_mpwqe_filler_strides: 0
rx_buff_alloc_err: 0
rx_cqe_compress_blks: 0
rx_cqe_compress_pkts: 0
rx_page_reuse: 0
rx_cache_reuse: 0
rx_cache_full: 0
rx_cache_empty: 2688
rx_cache_busy: 0
rx_cache_waive: 0
rx_congst_umr: 0
rx_arfs_err: 0
ch_events: 75
ch_poll: 75
ch_arm: 75
ch_aff_change: 0
ch_eq_rearm: 0
rx_out_of_buffer: 0
rx_if_down_packets: 15
rx_steer_missed_packets: 0
rx_vport_unicast_packets: 0
rx_vport_unicast_bytes: 0
tx_vport_unicast_packets: 0
tx_vport_unicast_bytes: 0
rx_vport_multicast_packets: 2
rx_vport_multicast_bytes: 172
tx_vport_multicast_packets: 12
tx_vport_multicast_bytes: 936
rx_vport_broadcast_packets: 37
rx_vport_broadcast_bytes: 9270
tx_vport_broadcast_packets: 6
tx_vport_broadcast_bytes: 360
rx_vport_rdma_unicast_packets: 0
rx_vport_rdma_unicast_bytes: 0
tx_vport_rdma_unicast_packets: 0
tx_vport_rdma_unicast_bytes: 0
rx_vport_rdma_multicast_packets: 0
rx_vport_rdma_multicast_bytes: 0
tx_vport_rdma_multicast_packets: 0
tx_vport_rdma_multicast_bytes: 0
tx_packets_phy: 0
rx_packets_phy: 0
rx_crc_errors_phy: 0
tx_bytes_phy: 0
rx_bytes_phy: 0
tx_multicast_phy: 0
tx_broadcast_phy: 0
rx_multicast_phy: 0
rx_broadcast_phy: 0
rx_in_range_len_errors_phy: 0
rx_out_of_range_len_phy: 0
rx_oversize_pkts_phy: 0
rx_symbol_err_phy: 0
tx_mac_control_phy: 0
rx_mac_control_phy: 0
rx_unsupported_op_phy: 0
rx_pause_ctrl_phy: 0
tx_pause_ctrl_phy: 0
rx_discards_phy: 0
tx_discards_phy: 0
tx_errors_phy: 0
rx_undersize_pkts_phy: 0
rx_fragments_phy: 0
rx_jabbers_phy: 0
rx_64_bytes_phy: 0
rx_65_to_127_bytes_phy: 0
rx_128_to_255_bytes_phy: 0
rx_256_to_511_bytes_phy: 0
rx_512_to_1023_bytes_phy: 0
rx_1024_to_1518_bytes_phy: 0
rx_1519_to_2047_bytes_phy: 0
rx_2048_to_4095_bytes_phy: 0
rx_4096_to_8191_bytes_phy: 0
rx_8192_to_10239_bytes_phy: 0
link_down_events_phy: 0
rx_prio0_bytes: 0
rx_prio0_packets: 0
tx_prio0_bytes: 0
```

```
tx_prio0_packets: 0
rx_prio1_bytes: 0
rx_prio1_packets: 0
tx_prio1_bytes: 0
tx_prio1_packets: 0
rx_prio2_bytes: 0
rx_prio2_packets: 0
tx_prio2_bytes: 0
tx_prio2_packets: 0
rx_prio3_bytes: 0
rx_prio3_packets: 0
tx_prio3_bytes: 0
tx_prio3_packets: 0
rx_prio4_bytes: 0
rx_prio4_packets: 0
tx_prio4_bytes: 0
tx_prio4_packets: 0
rx_prio5_bytes: 0
rx_prio5_packets: 0
tx_prio5_bytes: 0
tx_prio5_packets: 0
rx_prio6_bytes: 0
rx_prio6_packets: 0
tx_prio6_bytes: 0
tx_prio6_packets: 0
rx_prio7_bytes: 0
rx_prio7_packets: 0
tx_prio7_bytes: 0
tx_prio7_packets: 0
module_unplug: 0
module_bus_stuck: 0
module_high_temp: 0
module_bad_shorted: 0
ch0_events: 9
ch0_poll: 9
ch0_arm: 9
ch0_aff_change: 0
ch0_eq_rearm: 0
ch1_events: 23
ch1_poll: 23
ch1_arm: 23
ch1_aff_change: 0
ch1_eq_rearm: 0
ch2_events: 8
ch2_poll: 8
ch2_arm: 8
ch2_aff_change: 0
ch2_eq_rearm: 0
ch3_events: 19
ch3_poll: 19
ch3_arm: 19
ch3_aff_change: 0
ch3_eq_rearm: 0
ch4_events: 8
ch4_poll: 8
ch4_arm: 8
ch4_aff_change: 0
ch4_eq_rearm: 0
ch5_events: 8
ch5_poll: 8
ch5_arm: 8
ch5_aff_change: 0
ch5_eq_rearm: 0
rx0_packets: 0
rx0_bytes: 0
rx0_csum_complete: 0
rx0_csum_unnecessary: 0
rx0_csum_unnecessary_inner: 0
rx0_csum_none: 0
rx0_xdp_drop: 0
rx0_xdp_redirect: 0
rx0_lro_packets: 0
rx0_lro_bytes: 0
rx0_ecn_mark: 0
rx0_removed_vlan_packets: 0
rx0_wqe_err: 0
rx0_mpwqe_filler_cqes: 0
rx0_mpwqe_filler_strides: 0
rx0_buff_alloc_err: 0
rx0_cqe_compress_blks: 0
rx0_cqe_compress_pkts: 0
rx0_page_reuse: 0
rx0_cache_reuse: 0
rx0_cache_full: 0
rx0_cache_empty: 448
rx0_cache_busy: 0
rx0_cache_waive: 0
rx0_congst_umr: 0
rx0_arfs_err: 0
rx0_xdp_tx_xmit: 0
rx0_xdp_tx_full: 0
rx0_xdp_tx_err: 0
rx0_xdp_tx_cqes: 0
rx1_packets: 10
rx1_bytes: 3420
rx1_csum_complete: 10
rx1_csum_unnecessary: 0
rx1_csum_unnecessary_inner: 0
rx1_csum_none: 0
rx1_xdp_drop: 0
rx1_xdp_redirect: 0
rx1_lro_packets: 0
rx1_lro_bytes: 0
rx1_ecn_mark: 0
rx1_removed_vlan_packets: 0
```

```
rx1_wqe_err: 0
rx1_mpwqe_filler_cqes: 0
rx1_mpwqe_filler_strides: 0
rx1_buff_alloc_err: 0
rx1_cqe_compress_blks: 0
rx1_cqe_compress_pkts: 0
rx1_page_reuse: 0
rx1_cache_reuse: 0
rx1_cache_full: 0
rx1_cache_empty: 448
rx1_cache_busy: 0
rx1_cache_waive: 0
rx1_congst_umr: 0
rx1_arfs_err: 0
rx1_xdp_tx_xmit: 0
rx1_xdp_tx_full: 0
rx1_xdp_tx_err: 0
rx1_xdp_tx_cqes: 0
rx2_packets: 0
rx2_bytes: 0
rx2_csum_complete: 0
rx2_csum_unnecessary: 0
rx2_csum_unnecessary_inner: 0
rx2_csum_none: 0
rx2_xdp_drop: 0
rx2_xdp_redirect: 0
rx2_lro_packets: 0
rx2_lro_bytes: 0
rx2_ecn_mark: 0
rx2_removed_vlan_packets: 0
rx2_wqe_err: 0
rx2_mpwqe_filler_cqes: 0
rx2_mpwqe_filler_strides: 0
rx2_buff_alloc_err: 0
rx2_cqe_compress_blks: 0
rx2_cqe_compress_pkts: 0
rx2_page_reuse: 0
rx2_cache_reuse: 0
rx2_cache_full: 0
rx2_cache_empty: 448
rx2_cache_busy: 0
rx2_cache_waive: 0
rx2_congst_umr: 0
rx2_arfs_err: 0
rx2_xdp_tx_xmit: 0
rx2_xdp_tx_full: 0
rx2_xdp_tx_err: 0
rx2_xdp_tx_cqes: 0
...
tx0_packets: 1
tx0_bytes: 60
tx0_tso_packets: 0
tx0_tso_bytes: 0
tx0_tso_inner_packets: 0
tx0_tso_inner_bytes: 0
tx0_csum_partial: 0
tx0_csum_partial_inner: 0
tx0_added_vlan_packets: 0
tx0_nop: 0
tx0_csum_none: 1
tx0_stopped: 0
tx0_dropped: 0
tx0_xmit_more: 0
tx0_recover: 0
tx0_cqes: 1
tx0_wake: 0
tx0_cqe_err: 0
tx1_packets: 5
tx1_bytes: 300
tx1_tso_packets: 0
tx1_tso_bytes: 0
tx1_tso_inner_packets: 0
tx1_tso_inner_bytes: 0
tx1_csum_partial: 0
tx1_csum_partial_inner: 0
tx1_added_vlan_packets: 0
tx1_nop: 0
tx1_csum_none: 5
tx1_stopped: 0
tx1_dropped: 0
tx1_xmit_more: 0
tx1_recover: 0
tx1_cqes: 5
tx1_wake: 0
tx1_cqe_err: 0
tx2_packets: 0
tx2_bytes: 0
tx2_tso_packets: 0
tx2_tso_bytes: 0
tx2_tso_inner_packets: 0
tx2_tso_inner_bytes: 0
tx2_csum_partial: 0
tx2_csum_partial_inner: 0
tx2_added_vlan_packets: 0
tx2_nop: 0
tx2_csum_none: 0
tx2_stopped: 0
tx2_dropped: 0
tx2_xmit_more: 0
tx2_recover: 0
tx2_cqes: 0
tx2_wake: 0
tx2_cqe_err: 0
...
```

8.13.4.1.3 4.1.3. Traffic Control Info

The following TC objects are supported and reported regarding the ingress filters:

- Filters
    - flower
- Actions
    - mirred
    - tunnel_key

The info is provided as one of the following events:

- Basic filter event
- Flower/IPv4 filter event
- Flower/IPv6 filter event
- Basic action event
- Mirred action event
- Tunnel_key/IPv4 action event
- Tunnel_key/IPv6 action event

General notes:

- Actions always belong to a filter, so action events share the filter event's ID via the **event_id** data member
- Basic filter event only contains textual *kind* (so users can see which real life objects' support they are lacking)
- Basic action event only contains textual *kind* and some basic common statistics if available

8.13.4.1.4 4.1.4. Network Interfaces

ifconfig collects network interface data. To enable, set:

```
enable-provider=ifconfig
```

If the Prometheus endpoint is enabled and autodetection of fset indexes is disabled, add the following configuration to cache every collected network interface and arrange the index according to their names:

```
prometheus-fset-indexes=name
```

Metrices are collected for each network interface as follows:

```
name
rx_packets
tx_packets
rx_bytes
tx_bytes
rx_errors
tx_errors
rx_dropped
tx_dropped
multicast
collisions
rx_length_errors
rx_over_errors
rx_crc_errors
rx_frame_errors
rx_fifo_errors
```

```
rx_missed_errors
tx_aborted_errors
tx_carrier_errors
tx_fifo_errors
tx_heartbeat_errors
tx_window_errors
rx_compressed
tx_compressed
rx_nohandler
```

8.13.4.1.5 4.1.5. vNIC Provider

To enable the virtual NIC counters provider:

```
enable-provider=vnic
```

Counters are provided for all the valid Ethernet devices in the **devlink dev** output.

Counters are retrieved using **devlink health diagnose <device_name> reporter vnic** .

The exact list of counters depends on the driver version.

The following is an example of a counter list:

```
total_error_queues
send_queue_priority_update_flow
comp_eq_overrun
async_eq_overrun
cq_overrun
invalid_command
quota_exceeded_command
generated_pkt_steering_fail
handled_pkt_steering_fail
```

8.13.4.1.6 4.1.6. PPCC_ETH Provider

Programmable congestion control counters are based on algorithms defined by the user, although default algorithms are also available.

Counters are collected per MST device and algorithm (algo_slot) parameters.

MST device names can be found under **/dev/mst/** . Ensure **/dev/mst** is mounted into the DTS container from the host and that the container runs privileged.

The counter list depends on the installed MFT version.

/usr/lib64/mft or **/usr/lib/mft** must be mounted to the DTS container to retrieve the counter list corresponding to the installed MFT version. If not mounted, DTS uses its internal version of the counter list.

The parameters below are optional.

The provider supports automatic detection of available Ethernet MST devices for collection. To override this behavior, specify a comma-separated list of device names:

```
ppcc_eth_devices=mt41692_pciconf0,mt41692_pciconf0.1
```

User can narrow down the counters collection to a single algo_slot index by adding:

```
ppcc_algo_slot=1
```

The following algorithm parameters can be configured if needed. When not specified, they use the listed defaults:

```
ppcc_algo_param_index=0
ppcc_local_port=1
ppcc_pnat=0
ppcc_lp_msb=0
```

For more details, consult the official PPCC documentation.

Some **algo_slots** are not implemented. Devices with no counters are ignored. If no valid devices are detected, the provider is disabled.

8.13.4.2 4.2. GPU & Performance Providers

This category focuses on the computational health and utilization of the system's processing units. These providers collect vital performance, thermal, and architectural metrics directly from NVIDIA GPUs and BlueField Arm cores.

8.13.4.2.1 4.2.1. NVIDIA System Management Interface

The **nvidia-smi** provider collects GPU and GPU process information provided by the NVIDIA system management interface.

This provider is supported only on hosts with installed GPUs. All GPU cards supported by **nvidia-smi** are supported by this provider.

The counter list is GPU dependent. Additionally, per-process information is collected for the first 20 (by default) **nvidia_smi_max_processes** processes.

Counters can be either collected as string data "as is" in **nvidia-smi** or converted to numbers when **nvsmi_with_numeric_fields** is set.

To enable **nvidia-smi** provider and change parameters, set:

```
enable-provider=nvidia_smi

# Optional parameters:
#nvidia_smi_max_processes=20
#nvsmi_with_numeric_fields=1
```

8.13.4.2.2 4.2.2. NVIDIA Data Center GPU Manager

The **dcgm** provider collects GPU information provided by the NVIDIA data center GPU manager (DCGM) API.

This provider is supported only on hosts with installed GPUs, and requires running the **nv-hostengine** service (refer to DCGM documentation for details).

DCGM counters are split into several groups by context:

- GPU – basic GPU information (always)
- COMMON – common fields that can be collected from all devices
- PROF – profiling fields
- ECC – ECC errors

- NVLINK / NVSWITCH / VGPU – fields depending on the device type

To enable DCGM provider and counter groups, set:

```
enable-provider=dcgm

dcgm_events_enable_common_fields=1
#dcgm_events_enable_prof_fields=0
#dcgm_events_enable_ecc_fields=0
#dcgm_events_enable_nvlink_fields=0
#dcgm_events_enable_nvswitch_fields=0
#dcgm_events_enable_vgpu_fields=0
```

8.13.4.2.3 4.2.3. BlueField Performance

The **bfperf** provider collects calculated performance counters of BlueField Arm cores. It requires the executable **bfperf_pmc**, which is integrated in the DOCA BFB bundle of BlueField-3, as well as an active DPE.

To enable BlueField performance provider, set:

```
enable-provider=bfperf
```

When running, the **bfperf** provider is expected to recurrently reset the counters of the **sysfs.hwmon** component. Consider disabling it if **bfperf** is enabled.

8.13.4.3 4.3. Infrastructure Providers

Providers in this category monitor advanced protocols and broader data center infrastructure components. They capture deep diagnostic data, RDMA traffic events, and operational metrics for complex storage backends like NFS and Lustre.

8.13.4.3.1 4.3.1. Storage Provider

The Storage Provider collects and reports storage-related metrics for NFS mounts, local block devices, and Lustre filesystems.

To enable the provider, add the following to your configuration file:

```
enable-provider=storage
```

If no specific provider options are configured, the provider will automatically collect metrics from all available volumes across all types (NFS, local storage, and Lustre).

8.13.4.3.1.1 4.3.1.1. Platform-specific Requirements

4.3.1.1.1. BlueField (DPU)

When running the telemetry service on a BlueField DPU via kubelet, you must add **hostPID: true** and **hostIPC: true** to the pod spec. This ensures the storage provider can access host resources correctly.

4.3.1.1.2. Lustre Support (All Platforms)

The provider reads Lustre client states from the `llite` directory. You must mount the host Lustre `llite` path into the container (e.g., `/sys/kernel/debug/lustre/llite` -> `/sys/kernel/debug/lustre/llite`). For Lustre client versions below 2.10, the path resides under `/proc/fs/lustre/llite/`.

8.13.4.3.1.2 4.3.1.2. Storage Provider Configuration Options

Option	Description
`storage-nfs-volumes`	The NFS volumes to monitor. Accepts `all`, a comma-separated list (e.g., `vol1,vol2`), or leave empty to disable NFS.
`storage-devices`	The local block devices to monitor. Accepts `all` or a comma-separated list.
`storage-lustre-volumes`	The Lustre volumes to monitor. Accepts `all` or a comma-separated list. (Requires Lustre `llite` mount).
`storage-lustre-llite-dir`	The path to the Lustre `llite` directory inside the container. Default: `/sys/kernel/debug/lustre/llite`.

8.13.4.3.1.3 4.3.1.3. Metrics by Type

Metrics are exposed under a Mounts base scope, with each volume or device appearing as its own scope.

4.3.1.3.1. NFS Metrics

Tracks App Read/Write (bytes), Driver Read/Write (bytes/sec), Operations (32 NFS ops including lookup, read, write, create), and RPC queue/RTT/exe times (seconds).

Counter	Type	Unit
App Read	uint64	bytes
App Write	uint64	bytes
Driver Read	uint64	bytes/sec
Driver Write	uint64	bytes/sec
Operations (32 NFS ops: lookup, read, write, create, remove, etc.)	uint64 array	—
Read RPC queue	double	seconds
Read RPC RTT	double	seconds
Read RPC exe	double	seconds
Write RPC queue	double	seconds
Write RPC RTT	double	seconds
Write RPC exe	double	seconds

4.3.1.3.2. Lustre Metrics

Tracks Read/Write (bytes), Page Faults (count), Operations (16 Lustre ops), and Latency for Read/Write/Open/Close (seconds).

Counter	Type	Unit
Read	uint64	bytes
Write	uint64	bytes
Operations (16 Lustre ops: read_bytes, write_bytes, open, close, etc.)	uint64 array	—
Page Faults	uint64	count
Read Latency	double	seconds
Write Latency	double	seconds
Open Latency	double	seconds
Close Latency	double	seconds

4.3.1.3.3. Local Devices Metrics

Tracks Read/Write (bytes/sec) for Physical Device IO (PCIe NVMe), NVMe-oF, partitions, and RAID.

Counter	Type	Unit
Read	uint64	bytes/sec
Write	uint64	bytes/sec

8.13.4.3.2 4.3.2. Amber Provider

Amber data for both InfiniBand and Ethernet MST devices in amBER format.

/dev/mst should be accessible. Ensure **/dev/mst** is mounted into the DTS container from the host and that the container runs privileged.

The following configuration parameters are optional:

```
amber_devices=mt41692_pciconf0,mt41692_pciconf0.1   # Default:all, or a comma-separated list of devices under /dev/
mst
amber_update_interval_sec=30                        # Sample rate for collection amber counters
amber_labels=Device_FW_Version,MAC_Address          # string fields to filter, to reduce number of labels per amber
time series. If value is "none", strings fields will not be collected
amber_stop_collecting_on_command_error=true         # Use this option to control error handling: true (default) to
stop collecting from failing devices and avoid CPU overhead from persistent errors,
                                                    # or `false` to keep retrying and resume metric collection
after transient errors.
amber_service_stagger_msec=800                      # Amber service startup delay per device in ms
```

8.13.4.3.3 4.3.3. Diagnostic Data

Diagnostic data is comprised of two providers which gather diagnostic data counters from network interface cards (NICs). These providers support the same counters (as defined in a YAML file), but they differ in usage and collection frequency:

- Low frequency provider is defined in **dts_config.ini** and is controlled by DTS collection loop.
- High frequency provider is expected to run ad-hoc remotely (via REST API) or from **clx_ad_hoc_runner_config.ini** , and operates in a distinct flow for a limited duration.

Both providers get the counter set from a YAML file - see below.

8.13.4.3.3.1 4.3.3.1. Prerequisites

- Firmware version 28.43.1000 onwards for ConnectX-7 or 32.43.1000 onwards for BlueField-3.
- **fwctl** driver should be installed and loaded (see instructions in NVIDIA MLNX_OFED Documentation v24.07-0.6.1.0 for doca version < 3.1, or see instructions in DOCA-Telemetry-Diag for doca version >= 3.1), which is packaged in doca-ofed profile. Ensure that **fwctl** and **mlx5_fwctl** drivers are loaded with:

```
lsmod | grep fwctl
```

At least one **/dev/fwctl/fwctl*** device node must also be accessible inside the DTS container. Verify with:

```
ls -l /dev/fwctl/fwctl*
```

If no device node is visible, expose **/dev/fwctl** to the container.

8.13.4.3.3.2 4.3.3.2. Diagnostic Data Low Frequency

To enable the diagnostic data low frequency provider, set in **dts_config.ini**:

```
enable-provider=diagnostic_data_low_freq
```

Configure the counter set YAML file (all default files are available in the mounted folder host **/opt/mellanox/doca/services/telemetry/config/diagnostic_data_configs**).

If there is no such folder, you must update the config folder by running **telemetry-init.sh**. See "DTS Deployment" section for information.

```
diagnostic-data-yml-file=/config/diagnostic_data_configs/all-single-port.yml
```

To configure the diagnostic data timestamp collection type, set the following:

```
diagnostic-data-timestamp-collection-type=<method>
```

Where **<method>** can be one of the following:

- **no_counters** – Do not collect timestamp counters. This is the default method.
- **start_and_end** – Collect sample start and end timestamps
- **per_counter** – Collect every counter collection timestamp

To configure the clock firmware should use when collecting timestamps, set the following:

```
diagnostic-data-timestamp-source=<clock>
```

Where **<clock>** can be one of the following:

- **RTC** – Real-time clock (default clock used)
- **FRC** – Free-running clock

8.13.4.3.3.3 4.3.3.3. Diagnostic Data High Frequency

The Diagnostic Data High Frequency provider supports higher sampling frequencies with sub-millisecond resolution, enabling detailed and precise telemetry collection. This provider is intended for ad-hoc, short-term use because of the high data volume, unlike standard DTS providers configured in **/opt/mellanox/doca/services/telemetry/config/dts_config.ini**.

While the standard DTS flow functions as an endless collect-export loop, High Frequency Telemetry (HFT) operates as an external flow triggered by an HFT HTTP API or by HFT configuration file.

8.13.4.3.3.4 4.3.3.4. Providers Compatibility

Both low and high frequency providers can run concurrently. The low frequency provider samples at the DTS standard frequency (defined in **dts_config.ini**), and the high frequency provider samples counters based on the HFT configuration file (**clx_ad_hoc_runner_config.ini**).

To allow both providers to run concurrently, verify that the counters, the timestamp collection type, and the timestamp collection source are identical. Otherwise, when the high frequency provider starts sampling, the low frequency provider hangs until the end of the HFT session.

8.13.4.3.3.5 4.3.3.5. HFT HTTP API

The HFT HTTP API allows to start and stop remotely high frequency sampling sessions at scale on devices running DTS, it also allows to get and delete the data as needed.

The API is disabled by default. To enable it, add the following parameter to DTS configuration file **/opt/mellanox/doca/services/telemetry/config/dts_config.ini**.

```
enable-http-api=true
```

The default port is 9117. To modify it, set the **http-port** option with the desired port number.

The general expected API usage is:

1. Start an HFT ad hoc collection, wait for it to stop or manually stop it. By default, the collected telemetry is stored locally to binary files in the filesystem.
2. Get the binary data files. The binary data can be read using the **/opt/mellanox/collectx/bin/clx_read** app, packaged in **collectx-clxapidev**, a DOCA dependency package.
3. Delete the collection from DTS host. In any case, DTS runs a configurable retention policy for the collected data files which can be overriden in **dts_config.ini**:

```
# by size - set the max size limit of the collection files in bytes. Default is 1GB
ad-hoc-runner-max-size-bytes=1073741824
# by time - set the max age of collection files in seconds. Default is 1 week
ad-hoc-runner-max-age-seconds=604800
# set the minimum time for collection files to exist. Default is 10 minutes
ad-hoc-runner-min-age-seconds=600
```

4.3.3.5.1. Summary

Method	URL	Description	Query Parameters	Has Request Body
`POST`	`/ad-hoc-collection/start`	Start an ad hoc collection session. Returns collection ID.	-	Yes
`POST`	`/ad-hoc-collection/stop`	Stop active ad hoc collection session	-	No
`POST`	`/ad-hoc-collection/hft-counterset`	Post an HFT counter set YAML file	`filename`	Yes
`GET`	`/ad-hoc-collection`	Get compressed folder with collection data	`collection_id` , `delta`	No
`GET`	`/ad-hoc-collection/list`	List collection folders	-	No
`GET`	`/ad-hoc-collection/status`	Show status of collections	`collection_id`	No
`DELETE`	`/ad-hoc-collection`	Remove collection data permanently	`collection_id`	No

Query parameters are key-value pairs appended to the URL using `?` and separated by `&`.
Example: `/URL?key1=value1&key2=value2`.

Request body is a JSON object containing key-value pairs passed in the body of the request.
Example: `{"key1": "value1", "key2": "value2"}`.

Response body is the JSON-formatted reply returned by the server. The Success Response section of each endpoint specifies the expected response format.

Status code:

- `200` indicates a successful request
- Other status codes indicate errors

4.3.3.5.2. POST /ad-hoc-collection/start

Start ad hoc collection.

4.3.3.5.2.1. Request Body

Without a request body, this call will trigger an HFT collection for 30 seconds, in resolution of 10K samples per second, and will write the telemetry to binary data files.

Request body is passed as JSON with key-value pairs. all JSON values must be of string type.

The following table describes supported parameters in the request body:

Key	Default Value	Description
`sample-time-us`	`100000`	Sampling time in microseconds for SW iteration. one SW iteration collects all FW samples

Key	Default Value	Description
`num-iterations`	`300`	Number of iterations
`start-time`	Current time	Start time for collection. UTC epoch timestamp (in microseconds)
`end-time`	Calculated based on `start-time`, `num-iterations`, and `sample-time-us`	End time for collection. UTC epoch timestamp (in microseconds)
`file-write`	`true`	Whether to write binary files
`busy-wait-sampling`	`false`	Whether to use busy wait sampling instead of sleep
`prometheus-endpoint`	-	Prometheus endpoint URL
`fluentbit-config-dir`	-	Fluentbit configuration directory
`open-telemetry-receiver`	-	OpenTelemetry receiver URL
`remote-write-receiver`	-	Prometheus Remote write receiver URL
`hft-counterset`	-	Counterset to use. Should be The filename provided to /ad-hoc-collection/hft-counterset.
`provider.diagnostic-data-num-samples`	`1000`	Number of diagnostic data samples (FW parameter)
`provider.diagnostic-data-sample-period-nsec`	`100000`	Sample period in nanoseconds (FW parameter)
`provider.diagnostic-data-yml-file`	`/config/diagnostic_data_configs/all-single-port.yml`	Configuration file path. Conflicts with hft-counterset
`provider.diagnostic-data-timestamp-collection-type`	`start_and_end`	Type of timestamp collection: [`no_counters`, `start_and_end`, `per_counter`]
`provider.diagnostic-data-timestamp-source`	`RTC`	Source for timestamps: [`RTC`, `FRC`]

4.3.3.5.2.2. Success Response

```
{
  "status": "success",
```

```
  "message": "Collection created",
  "collection_id": 2
}
```

The **collection_id** parameter is generated automatically by this API and used in the GET and DELETE requests.

4.3.3.5.3. POST /ad-hoc-collection/stop

Stop active collection.

4.3.3.5.3.1. Success Response

```
{
  "status": "success",
  "message": "Collection stopped successfully"
}
```

4.3.3.5.4. POST /ad-hoc-collection/hft-counterset

Posts an HFT counterset that can later be used by the **POST /ad-hoc-collection/start** endpoint via the **hft-counterset** parameter.

Two methods are supported for posting a counterset to this endpoint:

- By providing a JSON payload
- By uploading a YAML file (using the format used by the diagnostic data high-frequency provider)

4.3.3.5.4.1. YAML File Request

The request is expected to be a multipart/form-data containing a valid counterset YAML file.

```
curl -X POST localhost:9117/ad-hoc-collection/hft-counterset?filename=my_cx7_hft_cset -F
"file=@my_cx7_hft_cset.yml"
```

4.3.3.5.4.2. JSON Request

JSON of key-value strings pairs, keys are the counter names and the value is a string representation of the hexadecimal data ID of that counter.

For example, the following JSON contains two counters

```
curl -X POST localhost:9117/ad-hoc-collection/hft-counterset?filename=my_cx7_hft_cset -d '{"port_rx_bytes_0":
"0x1020000100000000", "port_rx_packets_0": "0x1020000300000000"}'
```

4.3.3.5.4.3. Query Parameters

Parameter	Description
filename	Name to assign to the requested counterset

4.3.3.5.4.4. Success Response

```
{
  "status": "success",
  "message": "HFT counterset saved successfully",
  "filename": "my_cset_filename",
  "overriden": false
```

```
}
```

`overriden` is `true` if there was already a counterset with this name and it was replaced.

4.3.3.5.5. GET /ad-hoc-collection

This request is feasible for collections that were made with `file-write: true` and were not removed from the system (by DELETE /ad-hoc-collection or the retention policy).

4.3.3.5.5.1. Query Parameters

Parameter	Description
`collection_id`	Required collection ID
`delta`	When `delta=true`, the response contains only the delta—that is, the difference in data files written since the previous call with `delta=true`, using the same `collection_id` and `X-Client-ID` header. This option is applicable only when a collection is in progress. The data schema is provided in the first call with `delta=true` and remains consistent across subsequent delta requests.

4.3.3.5.5.2. Success Response

Content of `tar.gz` file. when extracted, a `collection_data` folder should appear in the working directory. Example with bash:

```
$ curl -s localhost:9117/ad-hoc-collection?collection_id=2 -o response.tar.gz
$ tar xf response.tar.gz
$ tree collection_data
collection_data/
    2025
        0401
            c-237-169-100-103
                diagnostic_data_1743529761189538.bin
                diagnostic_data_1743529761802460.bin
                diagnostic_data_1743529762402948.bin
                diagnostic_data_1743529763003488.bin
                diagnostic_data_1743529763604078.bin
                diagnostic_data_1743529764204606.bin
                diagnostic_data_1743529764805022.bin
                diagnostic_data_1743529765405535.bin
                diagnostic_data_1743529766005931.bin
    schema
        schema_c4d0317ee6cecbab995bfbcc237c76b2.json
```

Example with `delta=true`:

```
$ curl -s -H "X-Client-ID: my_app" "localhost:9117/ad-hoc-collection?collection_id=3&delta=true" -o
response1.tar.gz
$ curl -s -H "X-Client-ID: my_app" "localhost:9117/ad-hoc-collection?collection_id=3&delta=true" -o
response2.tar.gz
$ curl -s -H "X-Client-ID: my_app" "localhost:9117/ad-hoc-collection?collection_id=3&delta=true" -o
response3.tar.gz
$ tar xf response1.tar.gz
$ tar xf response2.tar.gz
$ tar xf response3.tar.gz
$ tree collection_data/
collection_data/
    2025
        0709
            swx-ray02
                diagnostic_data_1752050796207691.bin
                diagnostic_data_1752050796512800.bin
                diagnostic_data_1752050796949445.bin
                diagnostic_data_1752050797235709.bin
                diagnostic_data_1752050797524730.bin
                diagnostic_data_1752050797826312.bin
    schema
        schema_300a8aceac1d63c968ae837f366c9db4.json
```

4.3.3.5.6. GET /ad-hoc-collection/list

List the collections available on the file system, to be gathered by GET /ad-hoc-collection.

4.3.3.5.6.1. Success Response

```
{
  "status": "success",
  "num_collections": 3,
  "collection_ids": [1,2,3]
}
```

4.3.3.5.7. DELETE /ad-hoc-collection

Delete data of an ad-hoc collection. Note that in addition to this API there is a configurable retention policy.

4.3.3.5.7.1. Query Parameters

Parameter	Description	Notes
collection_id	The collection ID to delete	0 means delete all

4.3.3.5.7.2. Success Response

```
{
  "status": "success",
  "message": "Collection removed"
}
```

8.13.4.3.3.6 4.3.3.6. HFT Configuration File

The configuration file is mounted to DTS container, the host path is **/opt/mellanox/doca/services/telemetry/config/clx_ad_hoc_runner_config.ini**

This configuration file specifies the HFT session's timing parameters, provider settings, and export configurations. HFT allows data export to endpoints or protocols distinct from those used in the standard DTS collection loop. The DTS configuration references the HFT configuration file as an "ad hoc runner," enabling DTS to monitor its status via the following directive:

```
ad-hoc-runner-file=/config/clx_ad_hoc_runner_config.ini
```

The HFT configuration file serves as both the trigger and definition for HFT sessions. Modifying the file stops the current session and applies the new configuration for the next session. Deleting the file stops any pending HFT sessions entirely.

4.3.3.6.1. HFT File Parameters

This table provides the details of the HFT parameters. Refer to section "DOCA Telemetry Service Guide#HFT Configuration File Example" for more helpful tips.

Option	Description
start-time	HFT session start time. If not used, the session starts immediately. UTC epoch timestamp (in microseconds) or human readable format: HH:MM:SS / HH:MM

Option	Description
`end-time`	HFT session end time. Ignored if `start-time` is missing. If not used, `end-time` is calculated using `num-iterations`. UTC epoch timestamp (in microseconds). or human readable format: HH:MM:SS / HH:MM
`num-iterations`	Number of iterations. If not used, `start-time` and `end-time` are required, and the number of iterations is calculated
`sample-time-us`	Time interval between iterations (in microseconds). Used with `num-iterations` or `start-time` / `end-time` to determine session length. distinct from `provider.diagnostic-data-sample-period-nsec`, which controls firmware sampling within each iteration
`provider`	Provider to use. Should be `diagnostic_data_high_freq`
`file-write`	Whether to write collected telemetry to files. If enabled, could potentially write several MB of data every second
`data-root`	Root folder for file writing. Ignored if `file-write=false`
`open-telemetry-receiver`	Enables export to specified open telemetry receiver
`remote-write-receiver`	Enables export to specified open telemetry receiver
`fluentbit-config-dir`	Enables export to fluent-bit receivers. By default the folder is the same fluent bit configs folder used by DTS. Users can specify different folder, as long as it is available in DTS container, according to container volumes
`prometheus-endpoint`	Enables in process HTTP server with data cache in prometheus, json and csv format. This is not recommended for HFT data collection due to high update rate of the data cache
`busy-wait-sampling`	Set busy wait between iterations (instead of sleep, which is less accurate).
`provider.diagnostic-data-num-samples`	Number of samples to collect in one iteration. Affects the buffer used by the firmware for diagnostic data.
`provider.diagnostic-data-sample-period-nsec`	Sample period between samples (in nanoseconds). This option specifies the sample interval per iteration, as the provider collects N samples during each iteration
`provider.diagnostic-data-yml-file`	The diagnostic data counters YAML file to use

Option	Description
`provider.diagnostic-data-timestamp-collection-type`	`no_counters`, `start_and_end`, or `per_counter`. Same option as in low frequency provider
`provider.diagnostic-data-timestamp-source`	`RTC` or `FRC`. Same option as in low frequency provider

4.3.3.6.2. HFT Configuration File Example

```
## DTS configuration file for ad-hoc high frequency collection
## When modified, the file is parsed and applied.
## Note that the folders path is the container path, not the host path.

## Each section defines a collection. A file may have several sections, each one defines a high frequency
collection.
## Section names must be unique and will be used as collection name by clx.
[collection-session]

### Time between samples in microseconds
sample-time-us=100000

### Start time of high frequency collection. Can be in the format HH:MM:SS or HH:MM or as epoch timestamp in
microseconds
### Note - in container, the time is in UTC
start-time=18:00:00

### End time of high frequency collection. Can be in the format HH:MM:SS or HH:MM or as epoch timestamp in
microseconds
### Note - in container, the time is in UTC
end-time=18:01:00
### Alternatively, you can set the number of iterations. This and start_time field will determine the end time
#num-iterations=300

### Data provider to use
provider=diagnostic_data_high_freq

### Write data to file system. Could potentially fill up the disk
file-write=false

### Root directory to store the data
# Ignored if file-write is set to false
data-root=/data

### Enable busy wait between iterations, for a more accurate sample time (default is false)
#busy-wait-sampling=true

### Set prometheus endpoint to enable http endpoint
#prometheus-endpoint=http://0.0.0.0:9112

### Set fluentbit config dir to enable fluentbit export
#fluentbit-config-dir=/config/fluent_bit_configs

### Set open telemetry receiver to enable open telemetry export
#open-telemetry-receiver=http://0.0.0.0:9502/v1/metrics

### Set remote write receiver to enable remote write export
#remote-write-receiver=http://0.0.0.0:9090/api/v1/write

### Provider specific parameters. Format is 'provider.$KEY=$VALUE'.
### The options below are specific to the diagnostic data high frequency provider

# Number of samples to collect on each iteration
provider.diagnostic-data-num-samples=1000

# The time period (in nanoseconds) between samples
provider.diagnostic-data-sample-period-nsec=100000

# The YAML file with the configuration for the diagnostic data provider
provider.diagnostic-data-yml-file=/config/diagnostic_data_configs/all-single-port.yml

# Diagnostic Data timestamp collection type. Options are ['no_counters', 'start_and_end', 'per_counter']. default:
'no_counters'
#provider.diagnostic-data-timestamp-collection-type=start_and_end

# Diagnostic Data timestamp source. Options are ['RTC', 'FRC']. default: 'RTC'
```

```
#provider.diagnostic-data-timestamp-source=FRC
```

8.13.4.3.3.7 4.3.3.7. Diagnostic Data YAML File

For compatibility with other related tools, the counter set is defined in YAML format.

There are 4 YAML files within a DTS container (one per permutation of BlueField-3 and ConnectX-7 with dual or single ports). The path to the YAMLs folder is **/opt/mellanox/doca/services/telemetry/config/diagnostic_data_configs** which is mounted to **/config/diagnostic_data_configs**.

By default, YAML files include a counter set that is not device-specific. This implies that the same counter set is utilized across all devices by default.

To target specific devices, set **device** (or its alias **devices**) to a comma-separated list of RDMA device names or full PCI BDFs. One YAML file can target multiple devices. If the field is omitted from a single YAML file, the counter list is applied to all available physical devices. Use **diagnostic-data-yml-dir** when different devices require different counter sets; each file in that directory must specify its target devices. This folder should be available to the container under **/opt/mellanox/doca/services/telemetry/config**.

The following list describes the expected entries in the YAML file:

- **counters** – sequence of counters to collect
  - **id** – counter data ID
  - **desc** – counter description (optional)
  - **unit** – name of unit to collect from (optional)
  - **name** – name of counter to use (optional). If not specified, the generated name is based on the counter description. Otherwise, it is based on the data ID.
- **device** or **devices** – comma-separated RDMA device names or full PCI BDFs, for example **mlx5_0,mlx5_2** or **0000:03:00.0,0000:17:00.0** (optional).

Unlike port counters, ASIC temperature counters are indexed by sensor. The counter name uses **s0** for sensor 0. The YAML file **asic-sensors.yml** can be used for ASIC temperature collection.

Do not name it with a trailing _<digit>, for example: mtmp_sensor_temperature_0 , as a trailing _<digit> is parsed as a port index by the Diagnostic Data provider. Use **_s0**, **_s1**, and so on for ASIC sensor names. To collect another ASIC temperature sensor, change both the sensor index in the counter ID and the name suffix:

```
  - id: 0x11A0000400000000
    name: mtmp_sensor_temperature_s0
  - id: 0x11A0000400000001
    name: mtmp_sensor_temperature_s1
```

4.3.3.7.1. YAML File Example

The following is the content of the **all-dual-port.yml** file provided by DTS:

```
counters:
  - id: 0x1020000100000000
```

```
    name: port_rx_bytes_0
  - id: 0x1020000100000001
    name: port_rx_bytes_1
  - id: 0x1020000300000000
    name: port_rx_packets_0
  - id: 0x1020000300000001
    name: port_rx_packets_1
  - id: 0x1140000100000000
    name: port_tx_bytes_0
  - id: 0x1140000100000001
    name: port_tx_bytes_1
  - id: 0x1140000300000000
    name: port_tx_packets_0
  - id: 0x1140000300000001
    name: port_tx_packets_1
  - id: 0x1100000100000000
    name: port_tx_transport_cnp_sent_packets_0
  - id: 0x1100000100000001
    name: port_tx_transport_cnp_sent_packets_1
  - id: 0x1080000500000000
    name: port_rx_transport_cnp_handled_packets_0
  - id: 0x1080000500000001
    name: port_rx_transport_cnp_handled_packets_1
  - id: 0x1080000400000000
    name: port_rx_transport_ecn_packets_0
  - id: 0x1080000400000001
    name: port_rx_transport_ecn_packets_1
  - id: 0x1160000b00000000
    name: pcie_link_latency_total_read_ns
  - id: 0x1160000c00000000
    name: pcie_link_latency_total_read_packets
  - id: 0x1160000d00000000
    name: pcie_link_latency_max_read_ns
  - id: 0x1160000e00000000
    name: pcie_link_latency_min_read_ns
  - id: 0x1180000100000000
    name: global_icmc_request
  - id: 0x1180000200000000
    name: global_icmc_hit
  - id: 0x1180000300000000
    name: global_icmc_miss
```

The ‘Diagnostic Data’ section of the NVIDIA Adapters PRM defines both the diagnostic counter set and the rules for their data IDs

8.13.4.3.4 4.3.4. RDMA Notifications

The RDMA Notifications events provider collects notifications from the RDMA firmware and converts them into DTS events for telemetry purposes.

To enable the RDMA notifications provider, add the following line to the DTS configuration file (**/opt/mellanox/doca/services/telemetry/config/dts_config.ini**):

```
enable-provider=rdma_notifications
```

8.13.4.3.4.1 4.3.4.1. Prerequisites

- RDMA device(s) configured to operate in Ethernet protocol
- OFED installed
- ConnectX-7 or BlueField-3 and later
- Firmware version x.44.0820 or newer

8.13.4.3.4.2 4.3.4.2. Options

Name	Description	Type	Default	Notes
rdma-notifications-hca	HCA used for receive QP, comma separated for multiple HCAs	String	**mlx5_0**	

Name	Description	Type	Default	Notes
`rdma-notifications-num-event-indexes`	Number of event indexes	Integer	`1`	This option is relevant for Prometheus Export, in case user wants to store last N events of each type. For proper indexing of Prometheus export, in case the autodetection of fset indexes is disabled, add the configuration option `prometheus-fset-indexes=device_name,idx,syndrome`.
`rdma-notifications-num-buffers`	Number of buffers used by QP	Integer	`1000`	
`rdma-notifications-buffer-size`	Size of every buffer used by QP	Integer	`256`	
`rdma-notifications-syndrome-counter-event-interval-sec`	Interval in seconds to export syndrome counter events	Integer	`30`	See "counter event" in event types section.
`rdma-notifications-include-syndrome-zero-counters`	Export syndrome counter events if no events are found	Boolean	`false`	There are potentially 2^7 event types. If enabled, an event per syndrome will be sent every `rdma-notifications-syndrome-counter-event-interval-sec`.
`rdma-notifications-rt-all-vports`	Set the corresponding field in RDMA telemetry object	Boolean	`true`	
`rdma-notifications-rt-num-vport-ranges`	Number of vport ranges	Integer	`0`	May conflict with `rdma-notifications-rt-all-vports` option.
`rdma-notifications-rt-vport-range-[0-3]-max-id`	Vport range maximal ID (inclusive) in RDMA telemetry object	Integer	`0`	This row is for 4 different options. Use one index in every line. Example: `rdma-notifications-rt-vport-range-0-max-id`
`rdma-notifications-rt-vport-range-[0-3]-min-id`	Vport range maximal ID (inclusive) in RDMA telemetry object	Integer	`0`	This row is for 4 different options. Use one index in every line. Example: `rdma-notifications-rt-vport-range-0-min-id`

Name	Description	Type	Default	Notes
`rdma-notifications-qkey`	QKey used for firmware registration	Hexadecimal Integer	`0x100`	

8.13.4.3.4.3 4.3.4.3. Events List

There are two types of events provided by the RDMA notifications:

- Notification Event – A direct translation of an RDMA notification into one of four possible event types, based on the notification content
- Counter Event – A periodic aggregation event that reports the total number of events collected for each syndrome

For additional details, refer to the "RDMA Telemetry Notifications" section in the NVIDIA Adapters Programmer's Reference Manual (PRM).

Currently, the CQE with error is the only supported type of RDMA notification.

4.3.4.3.1. Notification Events

Additional fields are provided for convenience:

- **`guid`** – A globally unique identifier for the device
- **`idx`** – The event index, determined by the provider option **`rdma-notifications-num-event-indexes`**
- **`device_name`** – The name of the RDMA device, based on the devices listed in **`rdma-notifications-hca`**

These fields are not specified in the NVIDIA Adapters PRM but are included in the event data to enhance usability.

4.3.4.3.1.1. cqe_with_error_responder_ipv4

```
telemetry_counter
msn
syndrome
qp_type
vport_id
source_qpn
destination_qpn
psn
timestamp
dest_ipv4
node_guid
idx
device_name
```

4.3.4.3.1.2. cqe_with_error_responder_ipv6

```
telemetry_counter
msn
syndrome
qp_type
vport_id
```

```
source_qpn
destination_qpn
psn
timestamp
dest_ipv6
node_guid
idx
device_name
```

4.3.4.3.1.3. cqe_with_error_requestor_ipv4

```
telemetry_counter
msn
syndrome
opcode
qp_type
vport_id
source_qpn
destination_qpn
psn
timestamp
dest_ipv4
node_guid
idx
device_name
```

4.3.4.3.1.4. cqe_with_error_requestor_ipv6

```
telemetry_counter
msn
syndrome
opcode
qp_type
vport_id
source_qpn
destination_qpn
psn
timestamp
dest_ipv6
node_guid
idx
device_name
```

4.3.4.3.2. Counter Events

Describing the number of events per syndrome, for each device.

4.3.4.3.2.1. cqe_with_error_syndrome_counter

```
syndrome
device_name
node_guid
CQEwE_events
```

8.13.4.3.5 4.3.5. Adaptive Retx Provider

The **doca_telemetry_adp_retx** provider exposes statistics for the Adaptive Retransmission (ADP-RETX) algorithm configured on supported NVIDIA network devices (BlueField and ConnectX).

The provider collects this telemetry as a histogram, organizing retransmit counts into specific time-range bins. Data can be reported globally for all Queue Pairs (QPs) on a device, or scoped to a specific VHCA ID.

To enable the Adaptive Retx provider, add the following to your configuration:

```
enable-provider=apt_retx
```

8.13.4.3.6 4.3.6. Phy Provider

The PHY provider collects transceiver module temperature through the DOCA PHY API. Enable it in the DTS config:

```
enable-provider=phy
```

The provider exports one temperature counter per supported device. The value is reported in Celsius.

If a cable is disconnected or the temperature cannot be read, the provider reports 0 for that sample.

For full list of prerequisites - see Phy API documentation in DOCA Telemetry library.

8.13.4.3.7 4.3.7. PCI Provider

The PCI provider collects PCIe status and performance information. Enable it in the DTS configuration:

```
enable-provider=pci
```

This provider uses the DOCA PCI API and has the same prerequisites.

By default, the provider gathers telemetry for all HCAs. To specify the devices to sample, set **pci-devices** to a comma-separated list of RDMA device names. For example:

```
pci-devices=mlx5_0,mlx5_1
```

Some information provided by this provider may contain strings, which can negatively affect time-series exports. To add a time series with the counter name **timestamp** and all collected strings as labels, set:

```
pci-enable-management-info-strings=true
```

List of counters:

Name	Description
link_width_enabled	Maximum link width enabled (PCI lanes; -1 if unknown)
link_speed_enabled_g	Maximum link speed enabled (-1 if unknown)
link_width_active	Active link width (PCI lanes; -1 if unknown)
link_speed_active_g	Active link speed (-1 if unknown)
max_read_request_size	Max read request size in bytes (-1 if unknown)
max_payload_size	Max payload size in bytes (-1 if unknown)
link_peer_max_speed_g	Peer max link speed (-1 if unknown)

Name	Description
num_of_pfs	Number of Physical Functions (PFs)
num_of_vfs	Number of Virtual Functions (for all PFs)
bdf0	Bus Device Function — only for function0
pci_power	Power reported by the PCI device (Watts; 0 = unknown)
lane0_physical_position	Physical lane position of logical lane0
precode_sup	1 if precoding is supported for current speed
precode_active	1 if precoding is active for current speed
flit_sup	1 if flit is supported for current speed
flit_active	1 if flit is active for current speed
correctable_error_detected	1 if a correctable error was detected
non_fatal_error_detected	1 if a non-fatal error was detected
fatal_error_detected	1 if a fatal error was detected
unsupported_request_detected	1 if an unsupported request was detected
aux_power_detected	1 if aux power was detected
transaction_pending	1 if a transaction is pending
tx_overflow_buffer_pkt	Packets dropped due to lack of PCIe buffers or receive path from NIC toward host
tx_overflow_buffer_marked_pkt	Packets marked due to lack of PCIe buffers or receive path from NIC toward host
rx_errors	Transitions to recovery due to framing and CRC (DLLP/TLP) errors
tx_errors	Transitions to recovery due to EIEOS and TS errors
crc_error_dllp	Transitions to recovery due to CRC DLLP errors
crc_error_tlp	Transitions to recovery due to CRC TLP errors
outbound_stalled_reads	% of last second NIC had outbound non-posted reads but lacked credits
outbound_stalled_writes	% of last second NIC had outbound posted writes but lacked credits
outbound_stalled_reads_events	Events where outbound_stalled_reads was above threshold
outbound_stalled_writes_events	Events where outbound_stalled_writes was above threshold
fec_correctable_error_counter	FEC correctable error counter
fec_uncorrectable_error_counter	FEC uncorrectable error counter

Name	Description
`l0_to_recovery`	Total L0→recovery (sum of EIEOS, TS, framing, retrain causes)
`effective_ber_magnitude`	Effective BER = `effective_ber_coef` × 10^(- `effective_ber_magnitude`)
`effective_ber_coef`	Effective BER coefficient
`fber_magnitude`	FBER = `fber_coef` × 10^(- `fber_magnitude`)
`fber_coef`	FBER coefficient
`pci_latency_bucket{le="..."}`	Cumulative count of reads with latency ≤ threshold (nanoseconds)
`pci_latency_bucket{le="+Inf"}`	Total read count
`pci_latency_sum`	Estimated total latency (nanoseconds); see caveat below
`pci_latency_count`	Total number of PCI reads counted

If `pci-enable-management-info-strings=true` , the Prometheus output below demonstrates the expected time series which contains all the strings as labels:

```
timestamp{data_type="pci_management_info_strings_event",  source="lego02", link_bdf="0000:01:00.0",
device_name="mlx5_0", pwr_status="N/A", port_type="upstream", lane_reversal="straight"}  1782109632836
1782109632836
```

8.13.4.4 4.4. Aggregation Providers

Unlike direct telemetry collectors, aggregation providers function as intermediaries. They receive, pool, and consolidate telemetry data from external applications or remote endpoints before routing it to the system's configured data exporters.

8.13.4.4.1 4.4.1. Fluent Aggregator

`fluent_aggr` listens on a port for Fluent Bit Forward protocol input connections. Received data can be streamed via a DOCA Telemetry Service Guide#Fluent Bit exporter.

The default port is 42442. This can be changed by updating the following option:

```
fluent-aggr-port=42442
```

8.13.4.4.2 4.4.2. Prometheus Aggregator

`prometheus_aggr` polls data from a list of Prometheus endpoints.

Each endpoint is listed in the following format:

```
prometheus_aggr_endpoint.{N}={host_name},{host_port_url},{poll_inteval_msec}
```

Where N starts from 0.

Aggregated data can be exported via a Prometheus Aggr Exporter endpoint.

8.13.5 5. Data Exporters

DTS can send the collected data to the following outputs:

- Data writer (saves binary data to disk)
- DOCA Telemetry Service Guide#Prometheus endpoint (keeps the most recent data to be pulled)
- DOCA Telemetry Service Guide#Fluent Bit (push-model streaming)
- Open Telemetry exporter (push-model streaming)
- Prometheus Remote write exporter (push-model streaming)

8.13.5.1 5.1. Data Writer

The data writer is disabled by default to save space on BlueField. Steps for activating data write during debug can be found under section DOCA Telemetry Service Guide#Enabling Data Output.

The schema folder contains JSON-formatted metadata files which allow reading the binary files containing the actual data. The binary files are written according to the naming convention shown in the following example (**apt install tree**):

```
tree /opt/mellanox/doca/services/telemetry/data/
/opt/mellanox/doca/services/telemetry/data/
    {year}
        {mmdd}
             {hash}
                  {source_id}
                      {source_tag}{timestamp}.bin
                  {another_source_id}
                       {another_source_tag}{timestamp}.bin
    schema
        schema_{MD5_digest}.json
```

New binary files appears when the service starts or when binary file age/size restriction is reached. If no schema or no data folders are present, refer to the Troubleshooting section.

source_id is usually set to the machine hostname. **source_tag** is a line describing the collected counters, and it is often set as the provider's name or name of user-counters.

Reading the binary data can be done from within the DTS container using the following command:

```
crictl exec -it <Container ID> /opt/mellanox/collectx/bin/clx_read -s /data/schema /data/path/to/datafile.bin
```

The path to the data file must be an absolute path.

Example output:

```
{
    "timestamp": 1634815738799728,
    "event_number": 0,
    "iter_num": 0,
    "string_number": 0,
    "example_string": "example_str_1"
}
{
    "timestamp": 1634815738799768,
    "event_number": 1,
    "iter_num": 0,
    "string_number": 1,
    "example_string": "example_str_2"
}
```

8.13.5.2 5.2. Prometheus

The standard Prometheus endpoint retains the most recent data for pull-based collection by a Prometheus server. This endpoint is enabled by default on port **9100**.

8.13.5.2.1 5.2.1. Data Retrieval & Formatting

To verify that data is available, run the following command on the BlueField device:

```
curl -s http://0.0.0.0:9100/metrics
```

This returns counters in the standard format: **counter_name {list of meta fields} counter_value timestamp**.

You can also request alternative output formats:

- JSON: **curl -s http://0.0.0.0:9100/json/metrics**
- CSV: **curl -s http://0.0.0.0:9100/csv/metrics**

8.13.5.2.2 5.2.2. Retrieving Schema Metadata

You can retrieve the schema metadata used by DTS by querying the management endpoint:

```
curl -s http://0.0.0.0:9100/management/schema
```

To fetch a specific schema, append the **schema_id** URI parameter using one of the IDs listed in the previous response:

```
curl -s http://0.0.0.0:9100/management/schema?schema_id=xxx
```

To view all available schemas, run:

```
curl -s http://0.0.0.0:9100/management/schema?schema_id=all
```

8.13.5.2.3 5.2.3. Configuration Details (dts_config.ini)

You can customize the endpoint (including changing the default **9100** port) using the following parameters in **dts_config.ini**:

- Disable Prometheus: Comment out the **prometheus=http://0.0.0.0:9100** line.
- Enable HTTPS: Change the prefix to **https** and ensure the server key and certificate files are located in the DTS config volume:

```
prometheus=https://0.0.0.0:9100
prometheus-cert=/config/certs/server.crt
prometheus-key=/config/certs/server.key
```

- Filter by Name/Tag: Provide a comma-separated list of counter names or data source tags to ignore. Because the Prometheus exporter cannot be used for streaming, users should add tags to ignore all streaming data. By default, **FI_metrics** are disabled:

```
prometheus-ignore-names=counter_name1,counter_name2
prometheus-ignore-tags=FI_metrics
```

- Custom Data Indexing: By default, DTS manages the Prometheus indexing mechanism automatically. However, users can override this to keep multiple data records with different index values (these index fields are added to Prometheus labels):

```
# Comma-separated counterset description for Prometheus indexing:
prometheus-indexes=idx1,idx2

# Comma-separated fieldset description for Prometheus indexing:
# Note: Starting in version 1.22, fset indexes are autodetected based on the provider's built-in schema.
Set this to override:
prometheus-fset-indexes=idx1,idx2
```

8.13.5.2.4 5.2.4. Prometheus Aggregator Exporter

Unlike the standard pull endpoint, the Prometheus Aggregator Exporter keeps the latest data aggregated from *multiple* polled endpoints using **prometheus_aggr**.

To enable this provider, set the host and port in **dts_config.ini**:

```
prometheus-aggr-exporter-host=0.0.0.0
prometheus-aggr-exporter-port=33333
```

8.13.5.2.4.1 5.2.4.1. Full Configuration Example

This example enables aggregation from two endpoints (sampled every 1000ms and 3000ms, respectively) and exports the aggregated data on port **33333**.

```
enable-provider=prometheus_aggr
prometheus_aggr_endpoint.0=AAA,http://0.0.0.0:9000,1000
prometheus_aggr_endpoint.1=BBB,http://0.0.0.0:9002/metrics,3000
prometheus-aggr-exporter-host=0.0.0.0
prometheus-aggr-exporter-port=33333
```

The endpoint name is automatically added as a label to the metrics (e.g., **ep_name=AAA** for the first endpoint).

8.13.5.3 5.3. Fluent Bit

Fluent Bit allows streaming to multiple destinations. Destinations are configured in **.exp** files that are documented in-place and can be found under:

```
/opt/mellanox/doca/services/telemetry/config/fluent_bit_configs
```

Fluent Bit allows exporting data via "Forward" protocol which connects to the Fluent Bit/FluentD instance on customer side.

Export can be enabled manually:

1. Uncomment the line with **fluent_bit_configs=...** in **dts_config.ini**.
2. Set **enable=1** in required **.exp** files for the desired plugins.
3. Additional configurations can be set according to instructions in the **.exp** file if needed.
4. Restart the DTS.
5. Set up receiving instance of Fluent Bit/FluentD if needed.
6. See the data on the receiving side.

Export file destinations are set by configuring **.exp** files or creating new ones. It is recommended to start by going over documented example files. Documented examples exist for the following supported plugins:

- forward
- file
- stdout
- kafka
- es (elastic search)
- influx

All **.exp** files are disabled by default if not configured by **initContainer** entry point through **.yaml** file.

To forward the data to several destinations, create several **forward_{num}.exp** files. Each of these files must have their own destination host and port.

8.13.5.3.1 5.3.1. Export File Configuration Details

Each export destination has the following fields:

- **name** – configuration name
- **plugin_name** – Fluent Bit plugin name
- **enable** – 1 or 0 values to enable/disable this destination
- **host** – the host for Fluent Bit plugin
- **port** – port for Fluent Bit plugin
- **msgpack_data_layout** – the msgpacked data format. Default is **flb_std**. The other option is custom. See section Msgpack Data Layout for details.
- **plugin_key=val** – key-value pairs of Fluent Bit plugin parameter (optional)
- **counterset** / **fieldset** – file paths (optional). See details in section Cset/Fset Filtering.
- **source_tag=source_tag1,source_tag2** – comma-separated list of data page source tags for filtering. The rest tags are filtered out during export. Event tags are event provider names. All counters can be enabled/disabled only simultaneously with a **counters** keyword.

Use **#** to comment a configuration line.

8.13.5.3.2 5.3.2. Msgpack Data Layout

Data layout can be configured using **.exp** files by setting **msgpack_data_layout=layout**. There are two available layouts: Standard and Custom.

The standard **flb_std** data layout is an array of 2 fields:

- timestamp double value
- a plain dictionary (key-value pairs)

The standard layout is appropriate for all Fluent Bit plugins. For example:

```
[timestamp_val, {"timestamp"->ts_val, type=>"counters/events", "source"=>"source_val", "key_1"=>val_1,
"key_2"=>val_2,...}]
```

The custom data layout is a dictionary of meta-fields and counter fields. Values are placed into a separate plain dictionary. Custom data format can be dumped with **stdout_raw** output plugin of Fluent-Bit installed or can be forwarded with **forward** output plugin.

Counters example:

```
{"timestamp"=>timestamp_val, "type"=>"counters", "source"=>"source_val", "values"=> {"key_1"=>val_1,
"key_2"=>val_2,...}}
```

Events example:

```
{"timestamp"=>timestamp_val, "type"=>"events", "type_name"=>"type_name_val", "source"=>" source_val",
"values"=>{"key_1"=>val_1, "key_2"=>val_2,...}}
```

8.13.5.3.3 5.3.3. Cset/Fset Filtering

Each export file can optionally use one **cset** and one **fset** file to filter DTS counters and events data.

- **cset** contains tokens per line to filter data with **"type"="counters"**.
- **fset** contains several blocks started with the header line **[event_type_name]** and tokens under that header. An Fset file is used to filter data with **"type"="events"**.

If several tokens must be matched simultaneously, use **<tok1>+<tok2>+<tok3>**. Exclusive tokens are available as well. For example, the line **<tok1>+<tok2>-<tok3>-<tok4>** filters names that match both tok1 and tok2 and do not match tok3 or tok4.

The following are the details of writing **cset** files:

```
# Put tokens on separate lines
# Tokens are the actual name 'fragments' to be matched
# port$ # match names ending with token "port"
# ^port # match names starting with token "port"
# ^port$ # include name that is exact token "port
# port+xmit # match names that contain both tokens "port" and "xmit"
# port-support # match names that contain the token "port" and do not match the "-" token "support"
#
# Tip: To disable counter export put a single token line that fits nothing
```

The following are the details of writing **fset** files:

```
# Put your events here
# Usage:
#
# [type_name_1]
# tokens
# [type_name_2]
# tokens
# [type_name_3]
# tokens
```

```
# ...
# Tokens are the actual name 'fragments' to be matched
# port$ # match names ending with token "port"
# ^port # match names starting with token "port"
# ^port$ # include name that is exact token "port
# port+xmit # match names that contain both tokens "port" and "xmit"
# port-support # match names that contain the token "port" and do not match the "-" token "support"

# The next example will export all the "tc" events and all events with type prefix "ethtool_" "ethtool" are
filtered with token "port":
# [tc]
#
# [ethtool_*]
# packet

# To know which event type names are available check export and find field "type_name"=>"ethtool_event_p0"
# ...
# Corner cases:
# 1. Empty fset file will export all events.
# 2. Tokens written above/without [event_type] will be ignored.
# 3. If cannot open fset file, warning will be printed, all event types will be exported.
```

8.13.5.4 5.4. NetFlow Exporter

NetFlow exporter must be used when data is collected as NetFlow packets from the telemetry client applications. In this case, DOCA Telemetry Exporter NetFlow API sends NetFlow data packages to DTS via IPC. DTS uses NetFlow exporter to send data to the NetFlow collector (3rd party service).

To enable NetFlow exporter, set **netflow-collector-ip** and **netflow-collector-port** in **dts_config.ini**. **netflow-collector-ip** could be set either to IP or an address.

For additional information, refer to the **dts_config.ini** file.

8.13.5.5 5.5. Open Telemetry Exporter

DTS is capable of streaming telemetry data to an Open Telemetry receiver using the HTTP-based OTLP metrics protocol. The exporter is disabled by default and is configured via **dts_config.ini**.

Reference documentation:

- Open Telemetry Metrics Service API
- OTLP Specification 1.3.0

8.13.5.5.1 5.5.1. Transport & Security Configurations

The exporter supports OTLP over HTTP (**/v1/metrics**) with the payload encoded as JSON by default, as well as OTLP over gRPC. Mutual TLS (using a client certificate, key, and CA bundle) is supported for both transports.

To enable telemetry streaming, set the destination of the Open Telemetry receiver using one of the following configurations:

8.13.5.5.1.1 5.5.1.1. Insecure OTLP over HTTP

```
open-telemetry-receiver=http://0.0.0.0:9502/v1/metrics
```

This example expects the running Open Telemetry receiver to be bound to port 9502 on the host running the DTS instance. Section "DOCA Telemetry Service Guide#Open Telemetry Metrics Receiver Sample Configuration" provides an example of the Open Telemetry receiver configuration.

8.13.5.5.1.2 5.5.1.2. OTLP over HTTP with TLS

```
open-telemetry-receiver=https://0.0.0.0:9504/v1/metrics
open-telemetry-cert-file=certs/client.pem
open-telemetry-key-file=certs/client.key
open-telemetry-ca-file=certs/ca.pem
```

8.13.5.5.1.3 5.5.1.3. Insecure OTLP over gRPC

```
open-telemetry-transport=grpc
open-telemetry-receiver=0.0.0.0:9501
```

8.13.5.5.1.4 5.5.1.4. OTLP over gRPC with TLS

```
open-telemetry-transport=grpc
open-telemetry-receiver=0.0.0.0:9503
open-telemetry-cert-file=certs/client.pem
open-telemetry-key-file=certs/client.key
open-telemetry-ca-file=certs/ca.pem
```

8.13.5.5.2 5.5.2. Advanced Data & Filtering Configurations

You can heavily customize the data stream using the following optional parameters in **dts_config.ini**:

8.13.5.5.2.1 5.5.2.1. Counterset and Fieldset

Apply specific filters to the data stream:

```
open-telemetry-counter-set=slow_counters
open-telemetry-field-set=slow_counters
```

Refer to section "Counterset and Fieldset" for more information.

8.13.5.5.2.2 5.5.2.2. Per-data-point Attributes

By default, attributes are assigned efficiently at the highest hierarchy. If your pipeline (e.g., Prometheus DB) requires attributes assigned directly per data point, enable this setting:

```
open-telemetry-with-data-point-attributes=true
```

8.13.5.5.2.3 5.5.2.3. Bulk Export Sizing

Data is sent in bursts of up to 100 data points by default. You can alter this bulk size:

```
open-telemetry-bulk-size=20
```

8.13.5.5.2.4 5.5.2.4. Start Timestamp

Include the DTS start timestamp in the export:

```
open-telemetry-set-start-timestamp=true
```

8.13.5.5.2.5 5.5.2.5. Description Field

Include the description field in the exported data:

```
open-telemetry-with-description=true
```

8.13.5.5.3 5.5.3. Serialization Overrides

The default serialization logic uses a customized JSON encoder. You can override this behavior:

- Use official protobuf serializer: Encode the payload as Protobuf instead of JSON:

  ```
  open-telemetry-fast-serializer=false
  open-telemetry-binary-protobuf=true
  ```

- Cumulative sum metrics: Change the default aggregation temporality to **Cumulative** for all Sum metrics (effectively marking them as unspecified):

  ```
  open-telemetry-sum-is-always-cumulative=off
  ```

8.13.5.5.4 5.5.4. Open Telemetry Metrics Receiver Sample Configuration

The following is a sample Open Telemetry server configuration file (**collector-config.yaml**) that aligns with the port assignments used in the examples above:

```
receivers:
  otlp/insecure:
    protocols:
      grpc:
        endpoint: 0.0.0.0:9501
      http:
        endpoint: 0.0.0.0:9502

  # Secure endpoints with TLS
  otlp/secure:
    protocols:
      grpc:
        endpoint: 0.0.0.0:9503
        tls:
          cert_file: /etc/otel/tls/server.crt
          key_file: /etc/otel/tls/server.key
      http:
        endpoint: 0.0.0.0:9504
        tls:
          cert_file: /etc/otel/tls/server.crt
          key_file: /etc/otel/tls/server.key

exporters:
  debug:
    verbosity: detailed

service:
  pipelines:
    traces:
      receivers: [otlp/insecure, otlp/secure]
      exporters: [debug]
    metrics:
      receivers: [otlp/insecure, otlp/secure]
      exporters: [debug]
    logs:
      receivers: [otlp/insecure, otlp/secure]
      exporters: [debug]
```

8.13.5.5.5 5.5.5. Running the Open Telemetry Instance

Use the following Docker command to launch the Open Telemetry collector using the configuration file and certificates mounted from your current directory:

```
docker run --rm --name "$(whoami)_open_telemetry" -p 9501:9501 -p 9502:9502 -p 9503:9503 -p 9504:9504 \
    -v $(pwd)/data:/data -v $(pwd)/collector-config.yaml:/etc/otelcol/config.yaml -v $(pwd)/certs:/etc/otel/tls
 otel/opentelemetry-collector
```

8.13.5.6 5.6. Prometheus Remote Write Exporter

DTS can stream the telemetry data towards an external Prometheus service with an enabled metrics receiver. If enabled, the DTS data exporter acts as a Prometheus remote write protocol "Sender", streaming telemetry data towards a Prometheus server using remote write protocol.

Refer to Prometheus Remote Write Specification for more information.

To enable telemetry data streaming:

```
remote-write-receiver=http://0.0.0.0:9090/api/v1/write
```

This example assumes the running Prometheus remote write receiver to be bound to port 9090 on the host running the DTS instance.

For example, a Prometheus remote write receiver could be run using the following command:

```
docker run -p 9090:9090 prom/prometheus --config.file=/etc/prometheus/prometheus.yml \
    --storage.tsdb.path=/prometheus --web.console.libraries=/usr/share/prometheus/console_libraries \
    --web.console.templates=/usr/share/prometheus/consoles --web.enable-remote-write-receiver
```

The Prometheus remote write exporter can be configured to apply counter-set or field-set to the data stream, similar to the Open Telemetry exporter:

```
remote-write-counter-set=slow_counters
remote-write-field-set=slow_counters
```

The Prometheus remote write exporter sends data in bulk, combining bursts when possible. By default, the size of the bulk is up to 100 data points. If necessary, the size of the bulk can be altered using the bulk-size configuration parameter. For example:

```
remote-write-bulk-size=20
```

By default the posted telemetry follows V1 (see remote write spec). To enable V2 export, set:

```
remote-write-v2=true
```

8.13.5.7 5.7. Load Balancer Exporter

The load balancer exporter is an exporter which distributes or copies data between multiple instances of another (secondary) exporter.

For example, the number of instances may be set as follows depending on the use case:

Variable	Description	Default	Example	Required	Notes
`loadbalancer-exporter-name`	Name of the secondary exporter to load: `open_telemetry`, `remote_write`	N/A	`open_telemetry`	Yes	
`loadbalancer-num-instances`	Number of instances to create of the secondary exporter	N/A	`2`	Yes	Inferred when using `loadbalancer-INDEX-KEY` parameters
`loadbalancer-mode`	Balancing mode. Possible values: `distribution`; `replication`.	`distribution`	`replication`	No	
`loadbalancer-INDEX-KEY`	General prefix for key-value parameters to specific export instance	N/A	`loadbalancer-0-open-telemetry-receiver=http://0.0.0.0:9502/v1/metrics`	No	Index is 0-based. KEY expected be an option supported of the used exporter. See exporter description for list of options.
`loadbalancer-exporter-config-var`	Variable name to substitute for instances of the secondary exporter	N/A	`open-telemetry-receiver`	No	Will be deprecated. Use `loabalancer-INDEX-KEY` instead.

If any of the required variables is missing, the load balancer will not start.

When `loadbalancer-INDEX-KEY` and `loadbalancer-exporter-config-var` are not used, all instances are created equal, and their configuration is done according to other available configuration. If none of the instance configuration variables exist, that instance is created with other available configuration.

For example, the following is a section of the `dts_config.ini` configuration file which provides a sample of enabling load balancer to run 2 OTLP exporter instances:

```
loadbalancer-exporter-name=open_telemetry
loadbalancer-mode=replication
loadbalancer-0-open-telemetry-receiver=http://10.141.160.210:9502/v1/metrics
loadbalancer-1-open-telemetry-receiver=http://10.141.160.211:9502/v1/metrics
```

The default behavior is to export the entire collected data. To restrict the export interval of different instances (multirate export), add the required interval in seconds:

```
loadbalancer-0-open-telemetry-export-interval=30
loadbalancer-1-open-telemetry-export-interval=60
```

8.13.5.8 5.8. Distributed Real Time Analysis Exporter

The Real Time Analysis (RTA) Exporter is a powerful tool designed to export real-time analytics data from DTS. It provides seamless integration with various data sources and destinations, allowing you to efficiently collect, process, and export analytics information in real-time. One of its

key features is the ability to run an HTTP server at a preconfigured port, allowing for remote management of Lua scripts used in data processing.

It allows to get relevant insights based on telemetry data without the overhead of aggregation and central analysis. Potential use cases can be running link health tests (by analyzing L1 data) before executing an AI job (saving cluster time by avoiding starting a job on a bad link), or any other custom logic that is able to run on-demand \ on-going and uses telemetry data as input.

To enable this exporter, the lua scripts folder path (in DTS container filesystem) should be specified in DTS config file

```
lua-script-dir=/config/rta/lua_scripts
```

The default DTS configuration folder contains several Lua scripts that can be used as an example. The section below describes the API

This provider can run an HTTP server for RESTful interface of Lua scripts management.

Below are the parameters that can be modified to change the default logic:

```
rta-enable-http-api=true
rta-http-port=1812
```

8.13.5.8.1 5.8.1. Additional Configuration Parameters

Option Name	Description	Example
rta-inqueue	Size of the inbound data page processing queue. Default is 10.	**5**
rta-outqueue	Size of the outgoing Prometheus write queue. Default is 1000.	**5000**
rta-threads	Number of threads to run the remote write exporter. Default is 1.	**2**
rta-counter-set	Name of the counter set to apply to the data stream.	**low_freq**
rta-field-set	Name of the field set to apply to the data stream.	**low_freq**
rta-metrics-capacity	Defines the default maximum size of the buffer that stores the most recent metric values.	**16**
rta-max-metrics-capacity	Maximum capacity for metrics storage	**10000**
rta-enable-http-api	Enables RTA HTTP API requests.	**true**
rta-http-port	RTA HTTP server port.	**1814**
rta-ufm-events-endpoint	(In case UFM is running) URL of the UFM events endpoint for posting external events.	https://127.0.0.1/ufmRestV3/app/events/external_events

Option Name	Description	Example
`rta-ufm-access-token`	(In case UFM is running) Access token used for authentication when posting events to UFM.	`fJiq6mvCjMiQn6ZFwmKJxWygd8O2pX` Corresponds to r-ufm114
`rta-skip-unchanged-data-points`	Skips unchanged data points during processing if set to `true`.	`true`
`rta-script-log-lines-size`	Maximum number of log lines to keep per script	`1000`
`rta-lua-disallow-list`	Comma-separated list of forbidden Lua functions	`io.open,io.popen,os.execute,dofile,loadfile,require`
`rta-lua-allow-functions-list`	Comma-separated list of allowed Lua functions for scripts	`print,ipairs,pairs,type,tostring,tonumber`
`rta-lua-allow-libraries-list`	Comma-separated list of allowed Lua libraries for scripts	`math,string,table`

8.13.5.8.2 5.8.2. Lua Script Requirements

- If the script implements the `isEnabled()` function, it must return `true` for the script to be enabled. If this function is not implemented, the script will be disabled by default.
- Scripts must provide a set of metric names to be filtered for processing by implementing the `getMetricNames()` function.
- If a script is disabled, doesn't implement `getMetricNames()`, or returns an empty set of metric names, it will not be loaded.

8.13.5.8.3 5.8.3. Lua Scripts Workflow

The RTA Exporter plugin processes data in two ways:

1.
    a. Periodic:
        - At the end of each data page, the plugin calls the `processDataPageCompleted()` Lua function.
        - This function allows the Lua script to process the collected data.
2. On demand via HTTP server (see the HTTP REST API section).

8.13.5.8.4 5.8.4. Lua Script Examples

Lua scripts for RTA Exporter are available in `/opt/mellanox/doca/services/telemetry/config/rta/lua_scripts`. The following are examples for analyzing `sysfs` provider counters:

- `sysfs_normalized_cbw.lua` – Calculates the percentage of theoretical maximum bandwidth lost due to congestion on InfiniBand ports

- **sysfs_check_link_state.lua** – Monitors six key InfiniBand link metrics to verify that links are properly configured and operational

8.13.5.8.5 5.8.5. Lua Scripts API

There are 2 directions of function calls:

- DTS calls Lua script functions
- Lua script calls DTS functions

8.13.5.8.5.1 5.8.5.1. Lua Script Functions

2 functions are required - **isEnabled** and **getMetricNames**

5.8.5.1.1. isEnabled()

Determines if the script should be loaded.

```
function isEnabled()
    return true -- or false
end
```

5.8.5.1.2. getMetricNames()

Provides a set of metric names to be filtered for processing.

```
function getMetricNames()
    return {"metric1", "metric2", "metric3"}
end
```

5.8.5.1.3. processDataPageCompleted()

Processes collected data at the end of each data page.

```
function processDataPageCompleted()
    -- Implementation for processing data
end
```

5.8.5.1.4. getMetricsCapacity()

Defines the maximum size of the buffer that stores the most recent metric values.

```
function getMetricsCapacity()
    return 20 -- Example: Set the buffer size to 20 entries
end
```

5.8.5.1.5. processChangedMetricsOnly()

This function instructs the host application (CLX) to provide only metrics that have changed since the last time the script accessed them.

```
function processChangedMetricsOnly()
    return true -- Receive only updated metrics data
end
```

- Purpose: Optimizes data processing by instructing the host application to provide only metrics with new values since the script last accessed them. This can significantly improve performance and reduce unnecessary data transfer.
- Return value:
    - **true** : The host application will provide only metrics that have changed since the last access by the script.
    - **false** : The host application will provide all metrics, regardless of whether they've changed.

5.8.5.1.6. processDataMinPeriodMs()

Returns the minimum delay in milliseconds between calls to **processDataPageCompleted()** . This function helps reduce the frequency of calls if data pages are completed too quickly.

```
function processDataMinPeriodMs()
    return 5000 -- Example: Ensure processDataPageCompleted() is called at least 5 seconds apart
end
```

- Purpose: This function allows you to control maximum frequency at which **processDataPageCompleted()** is called when multiple data pages are completed in rapid succession.
- Note: The frequency of data pages becoming ready for processing is controlled by the CLX configuration. If the **processDataMinPeriodMs()** value is lower than the interval at which data pages are completed (as set by CLX), **processDataPageCompleted()** will be called on each data page completion, as the delay condition will always be met.
- If data pages are completed more frequently than the specified delay, **processDataPageCompleted()** will be called less often to prevent excessive processing.

5.8.5.1.7. processData()

Processes data on demand when requested via the HTTP API.

```
function processData()
    -- Your logic here to process data
    -- For example, let's assume a simple response
    return "Data processed successfully","text/plain; charset=utf-8"
end
```

The **processData()** function is an optional function in Lua scripts that is only required when the script is executed on demand. It processes data and returns a response. When implemented, it must return two values:

1. **Response Content** (string): The actual data or message to be sent back
2. **Content Type** (string): Specifies how the response should be interpreted

Return Values

- **First Value** : The response content as a string
- **Second Value** : The content type as a string (e.g., **"text/plain; charset=utf-8"**)

Content Type Examples

- **"text/plain; charset=utf-8"** - For plain text responses

- `"text/html; charset=utf-8"` - For HTML responses
- `"application/json"` - For JSON responses

- Purpose: This function allows scripts to handle on-demand data processing requests initiated through the RTA HTTP API.
- Note: The `processData()` function is called independently of the regular data page processing cycle and is used for ad-hoc data processing tasks.

8.13.5.8.5.2 5.8.5.2. DTS functions that can be called from Lua script

The RTA Exporter plugin provides several functions implemented by DTS that allow Lua scripts to interact with metrics data. These functions are prefixed with `clx_` and are designed to be called from within the `processDataPageCompleted()` and `processData()` functions in your Lua scripts.

5.8.5.2.1. clx_get_updated_metric_names(metricNames)

This function returns a list of metric names that have been updated since the last access.

```
function clx_get_updated_metric_names(metricNames)
```

- Parameters:
  - `metricNames` : A table of metric names to check for updates.
- Returns:
  - A table containing the names of metrics that have been updated.

5.8.5.2.2. clx_create_label_sets_iterator(metric_name, skip_unchanged)

This function creates an iterator for label sets associated with a given metric.

```
for label_set_index, labels in clx_create_label_sets_iterator(metric_name, skip_unchanged) do
    -- Process data
end
```

- Parameters:
  - `metric_name` : The name of the metric to iterate over.
  - `skip_unchanged` : A boolean indicating whether to skip label sets that haven't changed.
- Returns:
  - An iterator that yields pairs of `label_set_index` and `labels` for the specified metric.

5.8.5.2.3. clx_get_last_data_points(metric_name, label_set_index)

Retrieves both timestamps and corresponding values for a specific metric and label set index.

```
local timestamps, values = clx_get_last_data_points(metric_name, label_set_index)
```

- Parameters:
  - `metric_name` : The name of the metric.
  - `label_set_index` : The index of the label set associated with the metric.
- Returns:

- `timestamps` : A table of timestamps corresponding to the data points.
- `values` : A table of values corresponding to the timestamps.

5.8.5.2.4. clx_get_last_metric_values_by_index(metric_name, label_set_index, n)

Retrieves the last `n` values for a specific metric and label set index.

```
local values = clx_get_last_metric_values_by_index(metric_name, label_set_index, n)
```

- Parameters:
  - `metric_name` : The name of the metric.
  - `label_set_index` : The index of the label set associated with the metric.
  - `n` : The number of last values to retrieve.
- Returns:
  - A table containing the last `n` values for the specified metric and label set index.

5.8.5.2.5. clx_get_labels_set(label_set_index)

Retrieves the labels set for a given label set index.

```
local labels = clx_get_labels_set(label_set_index)
```

- Parameters:
  - `label_set_index` : The index of the label set to retrieve.
- Returns:
  - A table containing the labels set for the specified index.

5.8.5.2.6. clx_get_last_metric_values(metric_name, labels, n)

This function retrieves the last `n` values for a specific metric and label set.

```
function clx_get_last_metric_values(metric_name, labels, n)
```

- Parameters:
  - `metric_name` : The name of the metric.
  - `labels` : The label set associated with the metric.
  - `n` : The number of last values to retrieve.
- Returns:
  - A table containing the last `n` values for the specified metric and label set.

5.8.5.2.7. clx_get_all_metrics_data_points(label_set_index, n)

This function retrieves historical data points for a set of metrics

```
local timestamps, values = clx_get_all_metrics_data_points(label_set_index, n)
```

- Parameters:
  - `label_set_index` : number - Index of the label set to get data for
  - `n` : number - Maximum number of data points to retrieve per metric

- Returns:
  - **timestamps_table** : table - Array of timestamps for each metric
  - **values_table** : table - Array of values for each metric
- Usage: Used to access historical metric data for analysis
- Example:

```
local timestamps, values = clx_get_all_metrics_data_points(label_set_index, 2)
if timestamps and values then
    for i, metric_name in ipairs(metricNames) do
        if values[i] and #values[i] > 0 then
            local last_value = values[i][#values[i]]
            -- Process the value
        end
    end
end
```

5.8.5.2.8. clx_post_ufm_event(event_dictionary)

Posts a UFM external event and returns the response. (This is relevant in case UFM is running).

```
local response = clx_post_ufm_event(event_dictionary)
```

- Parameters:
  - **event_dictionary** : A table containing the event details to be posted. (See: Trigger External Events)
- Returns:
  - The UFM external event response if successful, or an error string if the operation fails.

5.8.5.2.9. clx_log_info(...)

Logs an informational message.

```
clx_log_info("Informational message %s %d", "Additional context", 123)
```

- Parameters:
  - **...** : Variable number of arguments to log. These can be strings, numbers, or other types that can be converted to strings.

5.8.5.2.10. clx_log_warning(...)

Logs a warning message.

```
clx_log_warning("Warning message %s %d", "Additional context", 456)
```

- Parameters:
  - **...** : Variable number of arguments to log.

5.8.5.2.11. clx_log_error(...)

Logs an error message.

```
clx_log_error("Error message %s %d", "Additional context", 789)
```

- Parameters:

- `...` : Variable number of arguments to log.

5.8.5.2.12. clx_log_debug(...)

Logs a debug-level message, useful during development and troubleshooting.

```
clx_log_debug("Debugging details %s %d", "Additional context", 101112)
```

- Parameters:
  - `...` : Variable number of arguments to log.

5.8.5.2.13. Alternative Logging Functions

The following functions provide logging capabilities similar to Lua's **`print()`** function. They accept variadic arguments and automatically concatenate them into a single string for logging:

- **`clx_log_print_info(...)`** : Logs informational messages.
- **`clx_log_print_warning(...)`** : Logs warning messages.
- **`clx_log_print_error(...)`** : Logs error messages.
- **`clx_log_print_debug(...)`** : Logs debug-level messages.

Expected Arguments:

- These functions accept a variable number of arguments (`...`) without requiring modifiers for formatting.
- Arguments can be strings, numbers, or other types convertible to strings, and they are concatenated into a single log message.

5.8.5.2.14. Usage Example #1

Here's how you might use these logging functions within your Lua script:

```
function processDataPageCompleted()
  clx_log_info("Data page processing started with %s %s", #getMetricNames(), "metrics.")
end
```

5.8.5.2.15. Usage Example #2

Here's an updated example of how these functions should be used within the **`processDataPageCompleted()`** function:

```
function processDataPageCompleted()
  local metricNames = getMetricNames()
  local updatedMetrics = clx_get_updated_metrics_names(metricNames)
  for _, metric_name in ipairs(updatedMetrics) do
    for label_set_index, labels in clx_create_label_sets_iterator(metric_name, true) do
      local lastValues = clx_get_last_metric_values_by_index(metric_name, label_set_index, 5)
      -- Process the last 5 values and labels for this metric and label set
    end
  end
end
```

This example demonstrates the correct usage of the updated and new host application functions within **`processDataPageCompleted()`** . It shows how to:

1. Iterate over updated metrics and their label sets.

2. Retrieve the last 5 values for each metric and label set combination using the label set index.

5.8.5.2.16. Best Practices

Remember to call these functions only within **processDataPageCompleted()** to ensure access to the most recent and consistent metrics data.

1. It's recommended to return **true** from **processChangedMetricsOnly()** in most cases to receive only updated metrics data, improving efficiency.
2. Use **false** only when your script specifically needs to process all metrics on each data page completion, regardless of changes.
3. Ensure your script's logic can handle receiving and processing only a subset of metrics when this function returns **true**.
4. Be aware that returning **true** may result in receiving fewer metrics per processing cycle, but these metrics will always be the most recently updated ones.
5. Use **getMetricsCapacity()** to control memory usage and performance by limiting the number of stored metric values.
6. Use **processData()** to handle on-demand data processing tasks initiated via the HTTP API's *Run Script* command.
7. Ensure your script's logic can handle both regular data page processing and on-demand requests independently.
8. Be aware that **processData()** is called outside of regular processing cycles, so it should be designed for standalone execution.

8.13.5.8.6 5.8.6. HTTP REST API

8.13.5.8.6.1 5.8.6.1. List

- Purpose: Retrieves a list of available Lua scripts.
- Method: POST
- Request Body: A JSON object with the **scriptCommand** set to **"list"**.

```
{
  "scriptCommand": "list"
}
```

- Response: A JSON array containing the base file names (without path or extension) of the available Lua scripts.

8.13.5.8.6.2 5.8.6.2. Add

- Purpose: Submits a request to add a new Lua script to the system. The actual addition is performed at the earliest opportunity, typically at the beginning of the next incoming data page processing cycle.
- Method: POST
- Request Body: A JSON object with the **scriptCommand** set to **"add"**, **scriptName**, and **scriptContent**.

```
{
```

```
  "scriptCommand": "add",
  "scriptName": "example_script", // Optional
  "scriptContent": "Lua script content here"
}
```

- If **scriptName** is provided, the script will be saved with that name to the directory containing your Lua scripts.
- If **scriptName** is not provided (or empty), the script content will be saved to a temporary file. The lifecycle of such scripts will be tied to the RTA Exporter's lifecycle, meaning they will be automatically removed when the RTA Exporter terminates.

- Response: A success message indicating that the request has been queued for processing.

8.13.5.8.6.3 5.8.6.3. Remove

- Purpose: Submits a request to remove an existing and enabled Lua script from the system. The actual removal is performed at the earliest opportunity, typically at the beginning of the next incoming data page processing cycle.
- Method: POST
- Request Body: A JSON object with the **scriptCommand** set to **"remove"** and **scriptName**.

```
{
  "scriptCommand": "remove",
  "scriptName": "example_script"
}
```

  Important: Only enabled (active and running) Lua scripts that exist within the configured Lua scripts directory can be removed using this command. Attempts to remove scripts outside of this directory or scripts that are not enabled will be disregarded.
- Response: A success message indicating that the request has been queued for processing.

8.13.5.8.6.4 5.8.6.4. Run

- Purpose: Submits a request to execute a Lua script immediately. The script is executed directly.
- Method: POST
- Request Body: A JSON object with the **scriptCommand** set to **"run"**, optionally **scriptName**, and **scriptContent**.

```
{
  "scriptCommand": "run",
  "scriptName": "example_script", // Optional
  "scriptContent": "Lua script content here"  // Used if scriptName is not provided
}
```

  - If **scriptName** is provided, the existing script with that name will be executed.
  - If **scriptName** is not provided (or empty), the script content specified in **scriptContent** will be executed directly.
- Response: Returns the result of the **processData()** function execution if successful, or an error message if any error condition occurs during execution.
- Note: The Lua script must implement a function named **processData()** which is called by this **Run** request during execution. If this function is not present, execution will fail.

8.13.5.8.6.5 5.8.6.5. Reload Scripts

- Purpose: Submits a request to reload all Lua scripts from the configured scripts directory. This command does not perform reload immediately but queues it for execution at the earliest opportunity, typically at the beginning of the next incoming data page processing cycle.
- Method: POST
- Request Body: A JSON object with the `scriptCommand` set to `"reloadScripts"`.

```
{
  "scriptCommand": "reloadScripts"
}
```

- Response: A success message indicating that the reload request has been queued for processing, or an error message if any issue occurs during queuing.

8.13.5.8.6.6 5.8.6.6. Show Script Log

- Purpose: Retrieves the log entries for a specified Lua script.
- Method: POST
- Request Body: A JSON object with the `scriptCommand` set to `"log"` and `scriptName`.

```
{
  "scriptCommand": "log",
  "scriptName": "example_script"
}
```

-
    - Important: Only existing and enabled Lua scripts can be queried using this command. Attempts to query non-existent or disabled scripts will return an empty log.
- Response: A JSON object containing the log entries for the specified script, or an empty log if the script does not exist, is not enabled, or if there are no log entries.

8.13.5.8.6.7 5.8.6.7. Alternative Endpoints

In addition to the `/rta/lua/scripts/cmd` endpoint, the following endpoints provide a simpler way to perform common operations:

- Add script:

```
curl -X POST http://localhost:$CLX_RTA_HTTP_PORT/rta/lua/script/add/example_script \ -H "Content-Type:
text/plain" \ -d 'print("Hello, Lua!")'
```

- Remove script:

```
curl -X DELETE http://localhost:$CLX_RTA_HTTP_PORT/rta/lua/script/remove/example_script
```

- Show script log:

```
curl -X GET http://localhost:$CLX_RTA_HTTP_PORT/rta/lua/script/showlog/example_script
```

- Run script (by name):

```
curl -X GET http://localhost:$CLX_RTA_HTTP_PORT/rta/lua/script/run/example_script
```

- Run script (with content):

```
curl -X POST http://localhost:$CLX_RTA_HTTP_PORT/rta/lua/script/run \ -H "Content-Type: text/plain" \ -d
'function processData() print("Processing Data") end'
```

- List scripts:

```
curl -X GET http://localhost:$CLX_RTA_HTTP_PORT/rta/lua/scripts/list
```

- Reload scripts:

```
curl -X GET http://localhost:$CLX_RTA_HTTP_PORT/rta/lua/scripts/reload
```

8.13.5.8.6.8 5.8.6.8. Execution Timing

Commands like **add**, **remove**, and **reloadScripts** are not executed immediately. Instead, they are queued and performed at the earliest opportunity, typically at the beginning of the next incoming data page processing cycle. This ensures that script management operations do not interfere with ongoing data processing tasks. The **run** command executes scripts immediately.

8.13.5.8.6.9 5.8.6.9. Error Handling

- Invalid or Missing Command: If an unsupported **scriptCommand** is provided or if **scriptCommand** is not specified, the endpoint will return an error message indicating that the command is not recognized.
- Missing Arguments: If required arguments are missing for any command, the endpoint will return an error message specifying the missing fields.
- Script Existence: For the **add** command, if a script with the same name already exists, the endpoint will overwrite the existing script.
- Script Removal: For the **remove** command, if the specified script does not exist within the configured Lua scripts directory or is not enabled, the request will be disregarded without error.
- Script Execution: For the **run** command:
    - If execution fails due to missing or invalid code (e.g., missing **processData()** function), an error message will be returned.
- Script Reload Request:
    - If any issue occurs during queuing of reload requests (e.g., invalid configuration), an error message will be returned.

8.13.5.8.6.10 5.8.6.10. Security Considerations

- Authentication: Ensure that appropriate authentication mechanisms are in place to restrict access to this endpoint.
- Input Validation: Always validate the input data to prevent potential security vulnerabilities such as script injection.

8.13.5.8.6.11 5.8.6.11. Example Usage

- List scripts:

```
curl -X POST http://localhost:1814/rta/lua/scripts/cmd \
       -H "Content-Type: application/json" \
       -d '{"scriptCommand": "list"}'
```

- Add script:

```
curl -X POST http://localhost:1814/rta/lua/scripts/cmd \
       -H "Content-Type: application/json" \
       -d '{"scriptCommand": "add", "scriptName": "example", "scriptContent": "print(\"Hello, Lua!\")"}'
```

- Remove script:

```
curl -X POST http://localhost:1814/rta/lua/scripts/cmd \
       -H "Content-Type: application/json" \
       -d '{"scriptCommand": "remove", "scriptName": "example"}
```

- Run script:

```
curl -X POST http://localhost:$CLX_RTA_HTTP_PORT/rta/lua/scripts/cmd \
       -H "Content-Type: application/json" \
       -d '{"scriptCommand": "run", "scriptContent": "function processData() print(\"Processing Data\")
end processData()"}'
```

- Reload scripts request:

```
curl -X POST http://localhost:$CLX_RTA_HTTP_PORT/rta/lua/scripts/cmd \
      -H "Content-Type: application/json" \
      -d '{"scriptCommand": "reloadScripts"}'
```

- Show script log:

```
curl -X POST http://localhost:1814/rta/lua/scripts/cmd \
       -H "Content-Type: application/json" \
       -d '{"scriptCommand": "log", "scriptName": "example"}
```

8.13.6 6. Counterset and Fieldset Files

Counterset and fieldset files are configuration files used by DTS exporters to define which telemetry data fields to export and how to transform them. These files provide fine-grained control over data export, allowing you to select specific counters or fields and assign aliases for better readability.

They are primarily used by exporters to filter the telemetry data being exported.

The default folder for DTS counterset and fieldset files is: **`/opt/mellanox/doca/services/telemetry/config/prometheus_configs/cset/`** .

8.13.6.1 6.1. Difference Between .cset and .fset Files

Internally, DTS uses two types of schemas: counters and events. Each provider defines its own schema type.

- Most DTS providers use the events schema, which corresponds to **`.fset`** files.

- A few providers (e.g., **sysfs**, **diagnostic_data_low_freq**) use the counters schema, which corresponds to **.cset** files.

The main difference:

- A **.cset** file is a flat list of counter names or patterns.
- An **.fset** file groups fields under specific event types, which add an additional layer of hierarchy.

Schemas can be viewed in the data folder or retrieved via the HTTP endpoint: **http://<host>:9100/management/schema**.

8.13.6.2 6.2. Counterset (.cset) File Format

Counterset files follow a simple, comment-friendly key-value format. They support:

- Direct counter name selection
- Substring pattern matching
- Aliases for field renaming

```
# Lines starting with '#' are comments
# Select specific counters
port_xmit_bytes
_rcv_

# Field aliases
infiniband_link_downed_events=LinkDownedCounterExtended
infiniband_port_xmit_data_bytes=PortXmitDataExtended
normalized_cbw=normalized_CBW
```

8.13.6.3 6.3. Fieldset (.fset) File Format

Fieldset files define event-specific fields grouped under event type headers:

```
[ethtool_event_*]
rx_queue_0_xdp_packets
rx_queue_1_xdp_packets
tx_queue_0_xdp_packets

[dcgm_events_gpu]
DCGM_FI_DEV_GPU_TEMP
DCGM_FI_DEV_GPU_UTIL
DCGM_FI_DEV_MEM_CLOCK
```

8.13.6.4 6.4. Best Practices

File organization:

- Use descriptive filenames that reflect the file's purpose
- Group related counters or fields together
- Apply consistent naming conventions

Performance considerations:

- Keep files reasonably sized to avoid overhead
- Use patterns to minimize manual effort
- Cache frequently used configurations when possible

Maintainability:

- Add comments to clarify complex patterns or logic
- Use consistent formatting and indentation
- Document custom patterns or transformations

Testing:

- Start with small datasets
- Validate that patterns match expected fields
- Check the exported data format and structure

Security:

- Avoid exposing sensitive information in field names
- Use aliases to mask internal names if needed
- Set appropriate file and directory permissions

8.13.6.5 6.5. Common Issues

- Files not found: Check file paths and permissions.
- Fields not exported: Ensure the fields exist in the data schema and are referenced in the configuration.
- Performance issues: Review file size and pattern complexity.

8.13.7 7. DOCA Privileged Executor

When running on a BlueField, the DOCA Privileged Executer (DPE) is a daemon that enables specific DOCA services, including DTS, to access BlueField information that would otherwise be inaccessible from a container due to technology limitations or restricted permissions. By granting access to privileged system information, DPE enriches the data collected by DTS, providing additional insights and metrics. However, DTS can still operate independently without DPE, as the daemon is disabled by default.

DPE is available on Debian based operation systems only.

8.13.7.1 7.1. DPE Usage

DPE is controlled by systemd, and can be used as follows:

- To check DPE status:

```
sudo systemctl status dpe
```

- To start DPE:

```
sudo systemctl start dpe
```

- To stop DPE:

```
sudo systemctl stop dpe
```

DPE logs can be found in **/var/log/doca/telemetry/dpe.log**.

8.13.7.2 7.2. DPE Configuration File

DPE can be configured by the user. This section covers the syntax and implications of its configuration file.

The DPU telemetry collected by DTS does not require for this configuration file to be used.

The DPE configuration file allows users to define the set of commands that DPE should support. This may be done by passing the **-f** option in the following line of **/etc/systemd/system/dpe.service**:

```
ExecStart=/opt/mellanox/doca/services/telemetry/dpe/bin/dpeserver -vvv
```

To use the configuration file:

```
ExecStart=/opt/mellanox/doca/services/telemetry/dpe/bin/dpeserver -vvv -f /path/to/dpe_config.ini
```

The configuration file supports the following sections:

- **[server]** - list of key=value lines for general server configuration. Allowed keys: **socket**.
- **[commands]** - list of bash command lines that are not using custom RegEx
- **[commands_regex]** - list of bash command lines that are using custom RegEx
- **[regex_macros]** - custom RegEx definitions used in the **commands_regex** section

Consider the following example configuration file:

```
[server]
socket=/tmp/dpe.sock

[commands]
hostname
cat /etc/os-release

[commands_regex]
crictl inspect $HEXA       # resolved as "crictl inspect [a-f0-9]+"
lspci $BDF                 # resolved as "lspci ([0-9a-f]{4}\:|)[0-9a-f]{2}\:[0-9a-f]{2}\.[0-9a-f]"

[regex_macros]
HEXA=[a-f0-9]+
BDF=([0-9a-f]{4}\:|)[0-9a-f]{2}\:[0-9a-f]{2}\.[0-9a-f]
```

DPE is shipped with a preconfigured file that matches the commands used by the standalone DTS version included in the same DOCA installation. The file is located in **/opt/mellanox/doca/services/telemetry/dpe/etc/dpe_config.ini**.

Using a DPE configuration file allows for a fine-grained control over the interface exposed by it to the rest of the DOCA services. However, even when using the pre-supplied configuration file mentioned above, one should remember that it has been configured to match a fixed DTS version. That is, replacing the standalone DTS version with a new one downloaded from NGC means that the used configuration file might not cover additional features added in the new DTS version.

8.13.8 8. Troubleshooting

On top of the Troubleshooting section in the NVIDIA DOCA Container Deployment Guide, here are additional troubleshooting tips for DTS:

- For general troubleshooting, refer to the DOCA Troubleshooting.
- If the pod's state fails to be marked as "Ready", refer to **/var/log/syslog**.
- Check if the service is configured to write data to the disk as this may cause the system to run out of disk space.
- If a PIC bus error occurs, configure the following files inside the container:

```
crictl exec -it <container-id> /bin/bash
# Add to /config/clx.env the following line:
"
export UCX_TLS=tcp
"
```

- If a wrong hostname is set in the **source** / **source_id** label of DTS exports, this may indicate that the container is not running on the host network. In such cases, usually the hostname is based on the container name **doca-telemetry**. To fix this, update the docker command/ YAML file accordingly.

8.13.9 9. Release Notes

8.13.9.1 9.1. New Features

- Introduce new **phy** provider for transceiver module temperature collection.
- Support ASIC temperature sensor collection in **diagnostic_data** providers.
- Introduce new **pci** provider for PCIe status and performance information.
- Added options to **sysfs** provider - **sysfs-allowed-ports** , **sysfs-excluded-ports** , **sysfs-invalid-counter-value** .
- **remote_write** exporter - support V2 spec.
- Common labeling options for (PCI, NUMA node) for better time series indexing.

8.13.9.2 9.2. Bug Fixes

- **open telemetry** exporter memory and performance optimizations.
- **amber** provider
  - Fixed an issue where values in scientific notation were treated as strings instead of floating-point numbers.
  - Added the **amber_stop_collecting_on_command_error** option, allowing users to choose whether to stop or continue collecting data from a device after an error.
  - Added the **amber_service_stagger_msec** option, to reduce latency and load spikes.

8.13.9.3 9.3. Known issues

- DTS deployments with many sampled devices, multiple providers, or an unusually large number of platform files may exceed the container runtime's default RLIMIT_NOFILE limit. Configure an adequate limit for the container runtime.
- **amber** provider
  - An MST version mismatch between the DTS image and the host may prevent the Amber provider from running. We are working on an MST-agnostic implementation as the long-term solution. For this release, affected users should temporarily add the

required MFT volumes to the **docker run** command so the container uses the host's MFT installation

```
-v /etc/mft:/etc/mft
-v /usr/share/mft:/usr/share/mft
-v /usr/bin/mlxlink_ext:/usr/bin/mlxlink_ext
```

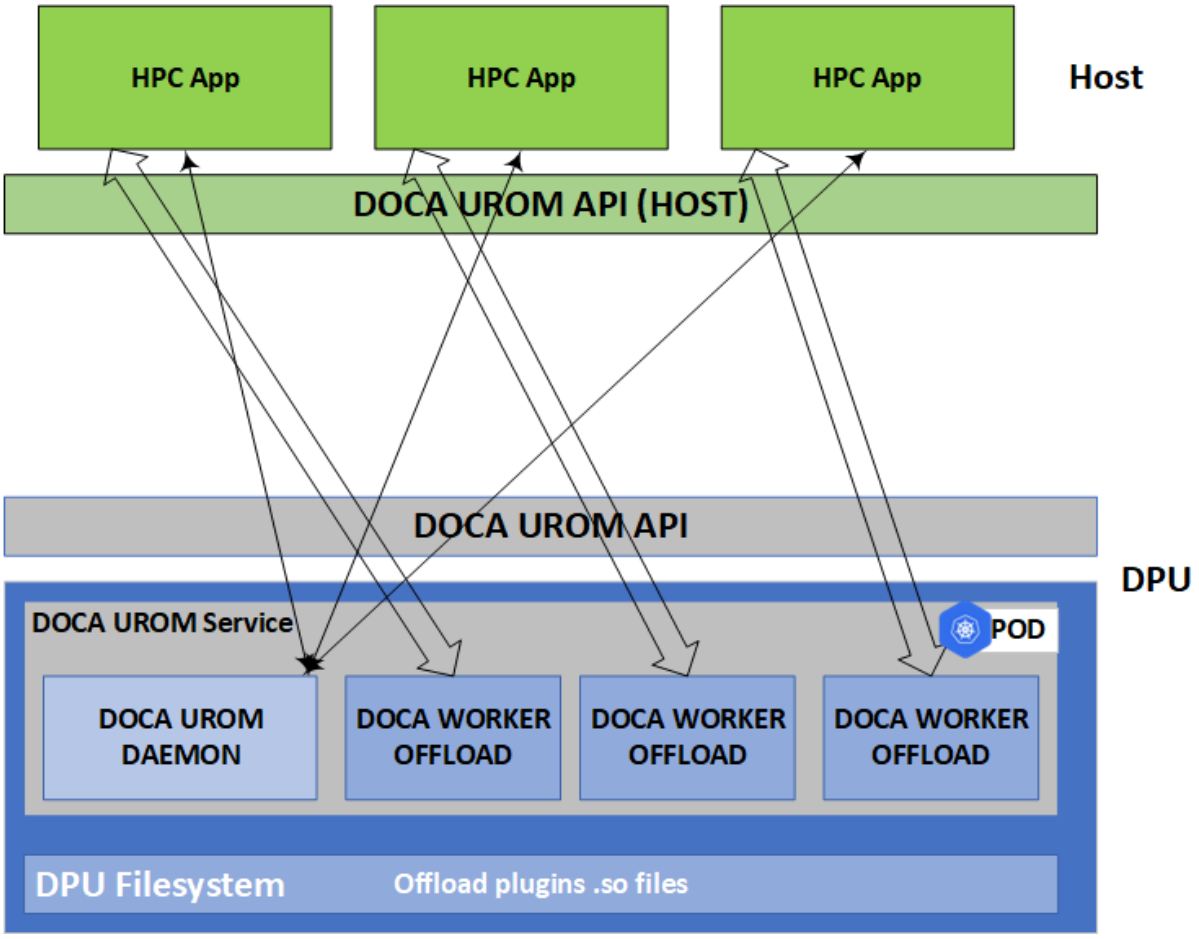
8.14 DOCA UROM Service Guide

This guide provides instructions on how to use the DOCA UROM Service on top of the NVIDIA® BlueField® networking platform.

8.14.1 Introduction

The DOCA UROM service provides a framework for offloading significant portions of HPC software stack directly from the host and to the BlueField device.

Using a daemon, the service handles the discovery of resources, the coordination between the host and BlueField, and the spawning, management, and teardown of the BlueField workers themselves.



The first step in initiating an offload request involves the UROM host application establishing a connection with the UROM service. Upon receiving the plugin discovery command, the UROM service responds by providing the application with a list of plugins available on the BlueField. The application then attaches the plugin IDs that correspond to the desired workers to their network identifiers. Finally, the service triggers UROM worker plugin instances on the BlueField to execute the parallel computing tasks. Within the service's Kubernetes pod, workers are spawned by the daemon in response to these offload requests. Each computation can utilize either a single library or multiple computational libraries.

8.14.2 Requirements

Before deploying the UROM service container, ensure that the following prerequisites are satisfied:

- Allocate huge pages as needed by DOCA:

```
$ echo '2048' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
$ sudo mkdir /mnt/huge
$ sudo mount -t hugetlbfs -o pagesize=2M nodev /mnt/huge
```

8.14.3 Service Deployment

For information about the deployment of DOCA containers on top of the BlueField, refer to the NVIDIA BlueField Container Deployment Guide.

Service-specific configuration steps and deployment instructions can be found under the service's container page.

8.14.4 Description

8.14.4.1 Plugin Discovery and Reporting

When the application initiates a connection request to the DOCA UROM Service, the daemon reads the **UROM_PLUGIN_PATH** environment variable. This variable stores directory paths to **.so** files for the plugins with multiple paths separated by semicolons. The daemon scans these paths sequentially and tries loading each **.so** file. Once the daemon finishes the scan, it reports the available BlueField plugins to the host application.

The host application gets the list of available plugins as a list of **doca_urom_service_plugin_info** structures:

```
struct doca_urom_service_plugin_info {
    uint64_t id;        // Unique ID to send commands to the plugin
    uint64_t version;   // Plugin version
    char plugin_name[DOCA_UROM_PLUGIN_NAME_MAX_LEN]; // .so filename
};
```

The UROM daemon is responsible for generating unique identifiers for the plugins, which are necessary to enable the worker to distinguish between different plugin tasks.

8.14.4.2 Loading Plugin in Worker

During the spawning of UROM workers by the UROM daemon, the daemon attaches a list of desired plugins in the worker command line. Each plugin is passed in a format of **so_path:id**.

As part of worker bootstrapping, the flow iterates all **.so** files and tries to load them by using **dlopen** system call and look for **urom_plugin_get_iface()** symbol to get the plugin operations interface.

8.14.4.3 Yaml File

The **.yaml** file downloaded from NGC can be easily edited according to users' needs:

```
 env:
      # Service-Specific command line arguments
      - name: SERVICE_ARGS
        value: "-l 60 -m 4096"
      - name: UROM_PLUGIN_PATH
        value: "/opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox/;/opt/mellanox/doca/samples/doca_urom/
plugins/worker_graph/"
```

- The **SERVICE_ARGS** are the runtime arguments received by the service:
  - **-l**, **--log-level <value>** – sets the (numeric) log level for the program **<10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>**
  - **--sdk-log-level** – sets the SDK (numeric) log level for the program **<10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>**
  - **-m**, **--max-msg-size** – specify UROM communication channel maximum message size
- The **UROM_PLUGIN_PATH** is an env variable that stores directory paths to **.so** files for the plugins

For each plugin on the BlueField, it is necessary to add a volume mount inside the service container. For example:

```
volumes:
  - name: urom-sandbox-plugin
    hostPath:
      path: /opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox
      type: DirectoryOrCreate
...
volumeMounts:
    - mountPath: /opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox
      name: urom-sandbox-plugin
```

8.14.5 Troubleshooting

When troubleshooting a container deployment issues, it is highly recommended to follow the deployment steps and tips found in the "Review Container Deployment" section of the DOCA Container Deployment Guide.

One could also check the **/var/log/doca/urom** log files for more details about the running cycles of service components (daemon and workers).

The log file name for workers is **urom_worker_<pid>_dev.log** and for the daemon it is **urom_daemon_dev.log**.

8.14.5.1 Pod is Marked as "Ready" and No Container is Listed

8.14.5.1.1 Error

When deploying the container, the pod's STATE is marked as **Ready** and an image is listed, however, no container can be seen running:

```
$ sudo crictl pods
POD ID              CREATED             STATE               NAME
NAMESPACE           ATTEMPT             RUNTIME
3162b71e67677       4 seconds ago       Ready               doca-urom-my-dpu                                  default
0                  (default)

$ sudo crictl images
IMAGE                                 TAG                 IMAGE ID            SIZE
k8s.gcr.io/pause                      3.2                 2a060e2e7101d       487kB
nvcr.io/nvidia/doca/doca_urom         1.0.0-doca2.7.0     2af1e539eb7ab       86.8MB

$ sudo crictl ps
CONTAINER           IMAGE               CREATED             STATE               NAME                ATTEMPT
POD ID              POD
```

8.14.5.1.2 Solution

In most cases, the container did start but immediately exited. This could be checked using the following command:

```
$ sudo crictl ps -a
CONTAINER           IMAGE               CREATED             STATE               NAME                ATTEMPT
POD ID              POD
556bb78281e1d       2af1e539eb7ab       6 seconds ago       Exited              doca-urom           1
          3162b71e67677       doca-urom-my-dpu
```

Should the container fail (i.e., reporting a state of **Exited**), it is recommended to examine the UROM's main log at **/var/log/doca/urom/urom_daemon_dev.log**.

In addition, for a short period of time after termination, the container logs could also be viewed using the container's ID:

```
$ sudo crictl logs 556bb78281e1d
...
```

8.14.5.2 Pod is Not Listed

8.14.5.2.1 Error

When placing the container's YAML file in the Kubelet's input folder, the service pod is not listed in the list of pods:

```
$ sudo crictl pods
POD ID              CREATED             STATE               NAME
NAMESPACE           ATTEMPT             RUNTIME
```

8.14.5.2.2 Solution

In most cases, the pod has not started because of the absence of the requested hugepages. This can be verified using the following command:

```
$ sudo journalctl -u kubelet -e. . .
Oct 04 12:12:19 <my-dpu> kubelet[2442376]: I1004 12:12:19.905064 2442376 predicate.go:103] "Failed to admit pod,
unexpected error while attempting to recover from admission failure" pod="default/doca-urom-service-<my-dpu>"
 err="preemption: error finding a set of pods to preempt: no set of running pods found to reclaim resources: [(res:
hugepages-2Mi, q: 104563999874), ]"
```

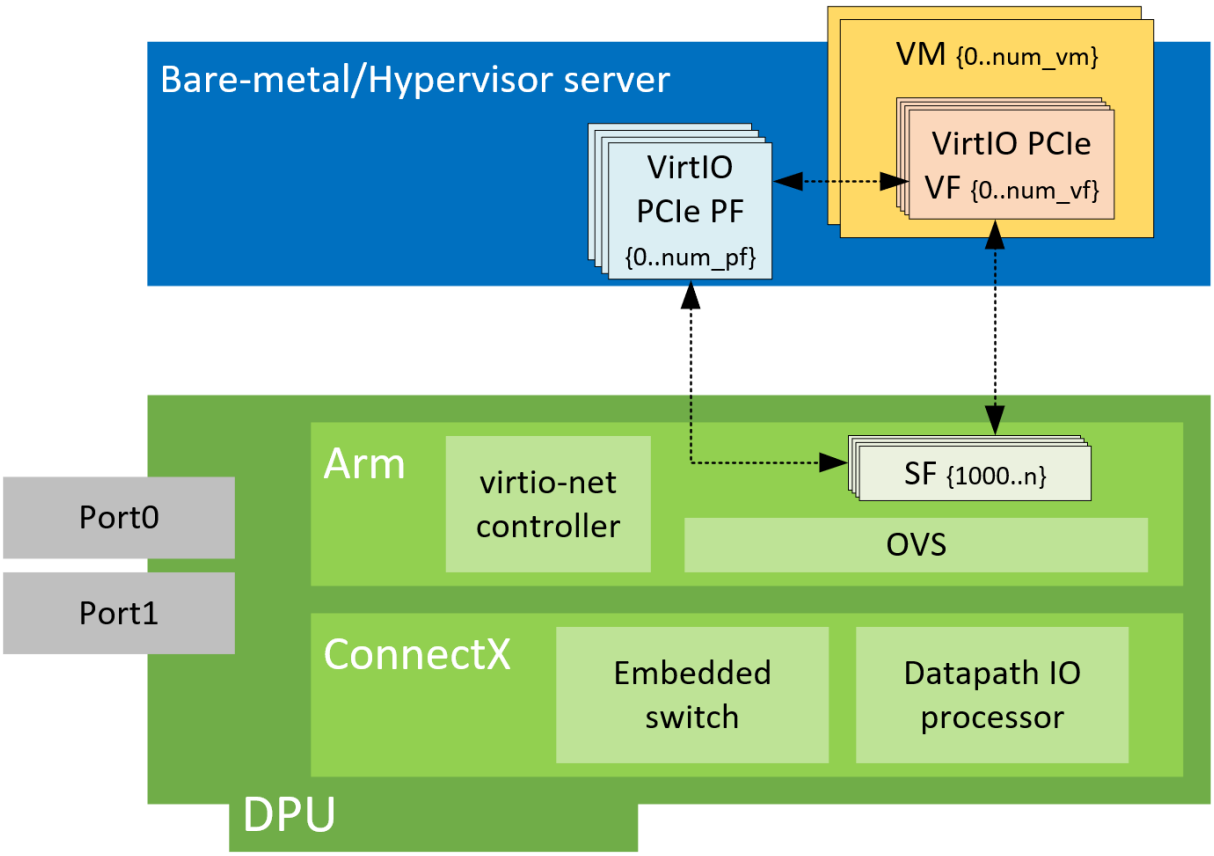
8.15 DOCA Virtio-net Service Guide

This guide provides instructions on how to use the DOCA virtio-net service container on top of NVIDIA® BlueField®-3 networking platform.

8.15.1 Introduction

NVIDIA® BlueField® virtio-net enables users to create virtio-net PCIe devices in the system where the BlueField is connected. In a traditional virtualization environment, virtio-net devices can be emulated by QEMU from the hypervisor, or offloading part of the work (e.g., dataplane) to the NIC (e.g., vDPA). Compared to those solutions, virtio-net PCIe devices offload both data and control plane to the BlueField networking device. The PCIe virtio-net devices exposed to the hypervisor do not depend on QEMU or other software emulators/vendor drivers from the guest OS.

The solution is based on BlueField family technology on top of virtual switch and OVS, so that virtio-net devices can benefit from the full SDN and hardware offload methodologies.

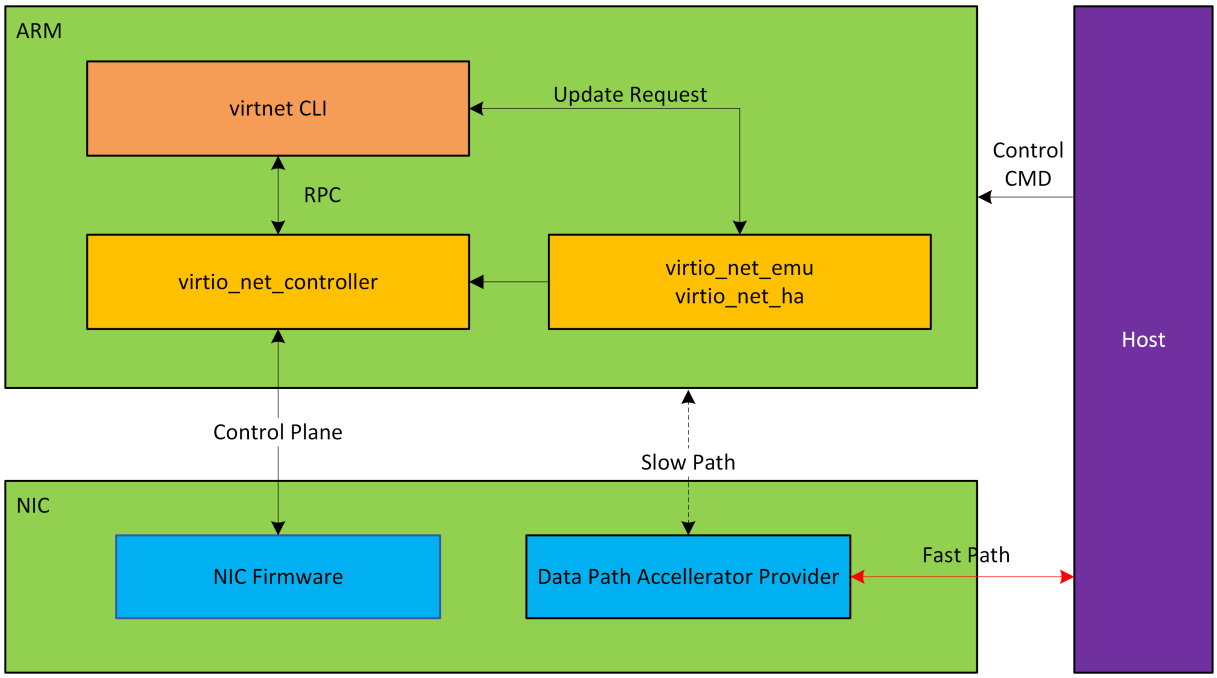


8.15.2 Virtio-net Controller SystemD Service

Virtio-net-controller is a **systemd** service which runs the BlueField with a command-line interface (CLI) frontend to communicate with the service running in the background. The controller **systemd** service is enabled by default and runs automatically after certain firmware configurations are deployed.

> Refer to "DOCA Virtio-net Service Guide#Virtio-net Deployment" for more information.

The processes **virtio_net_emu** and **virtio_net_ha** are created to manage live update and high availability.



8.15.3 Virtio-net Deployment

8.15.3.1 Updating OS Image on BlueField

To install the BFB bundle on the NVIDIA® BlueField®, run the following command from the Linux hypervisor:

```
[host]# sudo bfb-install --rshim <rshimN> --bfb <image_path.bfb>
```

For more information, refer to section "Deploying BlueField Software Using BFB from Host" in the *NVIDIA BlueField DPU BSP* documentation.

8.15.3.2 Updating NIC Firmware

From the BlueField networking platform, run:

```
[dpu]# sudo /opt/mellanox/mlnx-fw-updater/mlnx_fw_updater.pl --force-fw-update
```

For more information, refer to section "Upgrading Firmware" in the *NVIDIA DOCA Installation Guide for Linux*.

8.15.3.3 Configuring NIC Firmware

As default, DPU should be configured in DPU mode. A simple way to confirm DPU is running at DPU mode is to log into the BlueField Arm system and check if **`p0`** and **`pf0hpf`** both exists by running command below.

```
[dpu]# ip link show
```

Virtio-net full emulation only works in DPU mode. For more information about DPU mode configuration, please refer to *BlueField Modes of Operation*.

Before enabling the virtio-net service, configure firmware via **mlxconfig** tool is required. There are examples on typical configurations, the table listed relevant **mlxconfig** entry descriptions.

> ⚠ For **mlxconfig** configuration changes to take effect, perform a BlueField system-level reset.

Mlxconfig Entries	Description
VIRTIO_NET_EMULATION_ENABLE	Must be set to **TRUE**, for virtio-net to be enabled
VIRTIO_NET_EMULATION_NUM_PF	Total number of PCIe functions (PFs) exposed by the device for virtio-net emulation. Those functions are persistent along with host/BlueField power cycle.
VIRTIO_NET_EMULATION_NUM_VF	The max number of virtual functions (VFs) that can be supported for each virtio-net PF
VIRTIO_NET_EMULATION_NUM_MSIX	Number of MSI-X vectors assigned for each PF of the virtio-net emulation device, minimal is **4**.
VIRTIO_NET_EMULATION_NUM_VF_MSIX	Number of MSI-X vectors assigned for each VF of the virtio-net emulation device, minimal is **4**. Relevant for BlueField-3 devices only.
PCI_SWITCH_EMULATION_ENABLE	When **TRUE**, the device exposes a PCIe switch. All PF configurations are applied on the switch downstream ports. In such case, each PF gets a different PCIe device on the emulated switch. This configuration allows exposing extra network PFs toward the host which can be enabled for virtio-net hot-plug devices.
PCI_SWITCH_EMULATION_NUM_PORT	The maximum number of emulated switch ports. Each port can hold a single PCIe device (emulated or not). This determines the supported maximum number of hot-plug virtio-net devices. The maximum number depends on hypervisor PCIe resource and cannot exceed 31. ⚠ Check system PCIe resource. Changing this entry to a big number may result in the host not booting up, which would necessitate disabling the BlueField device and clearing the host NVRAM.
PER_PF_NUM_SF	When **TRUE**, the SFs configuration is defined by **TOTAL_SF** and **SF_BAR_SIZE** for each PF individually. If they are not defined for a PF, device defaults are used.

Mlxconfig Entries	Description
PF_TOTAL_SF	The total number of scalable function (SF) partitions that can be supported for the current PF. Valid only when PER_PF_NUM_SF is set to TRUE. This number should be greater than the total number of virtio-net PFs (both static and hotplug) and VFs. ⚠ This entry differs between the BlueField and host side mlxconfig. It is also a system wide value, which is shared by virtio-net and other users. The DPU normally creates 1 SF as default per port. Consider this default SF into account when reserving the PF_TOTAL_SF.
PF_SF_BAR_SIZE	Log (base 2) of the BAR size of a single SF, given in KB. Valid only when PF_TOTAL_SF is non-zero and PER_PF_NUM_SF is set to TRUE.
PF_BAR2_ENABLE	When TRUE, BAR2 is exposed on all external host PFs (but not on the embedded Arm PFs/ECPFs). The BAR2 size is defined by the log_pf_bar2_size.
SRIOV_EN	Enable single-root I/O virtualization (SR-IOV) for virtio-net and native PFs
EXP_ROM_VIRTIO_NET_PXE_ENABLE	Enable expansion ROM option for PXE for virtio-net functions ⚠ All virtio EXP_ROM options should be configured from host side other than the BlueField platform's side, only static PF is supported.
EXP_ROM_VIRTIO_NET_UEFI_ARM_ENABLE	Enable expansion ROM option for UEFI for Arm based host for virtio-net functions
EXP_ROM_VIRTIO_NET_UEFI_x86_ENABLE	Enable expansion ROM option for UEFI for x86 based host for virtio-net functions

The maximum number of supported devices is listed below. It does not apply when there are hot-plug and VF created at the same time.

Static PF	Hot-plug PF	VF
31	31	1008

⚠ The maximum supported number of hotplug PFs depends on the host PCI resource, it may support less or none on specific systems. Refer to host BIOS specification.

8.15.3.3.1 Static PF

Static PF is defined as virtio-net PFs which are persistent even after DPU or host power cycle. It also supports creating SR-IOV VFs.

The following is an example for enabling the system with 4 static PFs (VIRTIO_NET_EMULATION_NUM_PF) only:

10 SFs (**PF_TOTAL_SF**) are reserved to take into account other application using the SFs.

```
[dpu]# mlxconfig -d 03:00.0 s \
VIRTIO_NET_EMULATION_ENABLE=1 \
VIRTIO_NET_EMULATION_NUM_PF=4 \
VIRTIO_NET_EMULATION_NUM_VF=0 \
VIRTIO_NET_EMULATION_NUM_MSIX=64 \
PCI_SWITCH_EMULATION_ENABLE=0 \
PCI_SWITCH_EMULATION_NUM_PORT=0 \
PER_PF_NUM_SF=1 \
PF_TOTAL_SF=64 \
PF_BAR2_ENABLE=0 \
PF_SF_BAR_SIZE=8 \
SRIOV_EN=0
```

8.15.3.3.2 Hotplug PF

Hotplug PF is defined as virtio-net PFs which can be hotplugged or unplugged dynamically after the system comes up.

Hotplug PF does not support creating SR-IOV VFs.

The following is an example for enabling 16 hotplug PFs (**PCI_SWITCH_EMULATION_NUM_PORT**):

```
[dpu]# mlxconfig -d 03:00.0 s \
VIRTIO_NET_EMULATION_ENABLE=1 \
VIRTIO_NET_EMULATION_NUM_PF=0 \
VIRTIO_NET_EMULATION_NUM_VF=0 \
VIRTIO_NET_EMULATION_NUM_MSIX=64 \
PCI_SWITCH_EMULATION_ENABLE=1 \
PCI_SWITCH_EMULATION_NUM_PORT=16 \
PER_PF_NUM_SF=1 \
PF_TOTAL_SF=64 \
PF_BAR2_ENABLE=0 \
PF_SF_BAR_SIZE=8 \
SRIOV_EN=0
```

8.15.3.3.3 SR-IOV VF

SR-IOV VF is defined as virtio-net VFs created on top of PFs. Each VF gets an individual virtio-net PCIe devices.

VFs cannot be dynamically created or destroyed, they can only change from X to 0, or from 0 to X.

VFs will be destroyed when reboot host or unbind PF from virtio-net kernel driver.

The following is an example for enabling 126 VFs per static PF—504 (4 PF x 126) VFs in total:

```
[dpu]# mlxconfig -d 03:00.0 s \
VIRTIO_NET_EMULATION_ENABLE=1 \
VIRTIO_NET_EMULATION_NUM_PF=4 \
VIRTIO_NET_EMULATION_NUM_VF=126 \
VIRTIO_NET_EMULATION_NUM_MSIX=64 \
VIRTIO_NET_EMULATION_NUM_VF_MSIX=6 \
PCI_SWITCH_EMULATION_ENABLE=0 \
PCI_SWITCH_EMULATION_NUM_PORT=0 \
PER_PF_NUM_SF=1 \
PF_TOTAL_SF=512 \
PF_BAR2_ENABLE=0 \
PF_SF_BAR_SIZE=8 \
NUM_VF_MSIX=0 \
```

```
SRIOV_EN=1
```

8.15.3.3.4 PF/VF Combinations

Creating static/hotplug PFs and VFs at the same time is supported.

The total sum of PCIe functions to the external host must not exceed 1008. For example:

- If there are 2 PFs with no VFs (**NUM_OF_VFS=0**) and there is 1 RShim, then the remaining static functions is 1005 (1008-3).
- If 1 virtio-net PF is configured (**VIRTIO_NET_EMULATION_NUM_PF=1)** , then up to 1004 virtio-net VFs can be configured (**VIRTIO_NET_EMULATION_NUM_VF=1004**)
- If 2 virtio-net PF (**VIRTIO_NET_EMULATION_NUM_PF=2**), then up to 502 virtio-net VFs can be configured (**VIRTIO_NET_EMULATION_NUM_VF=502**)

The following is an example for enabling 15 hotplug PFs, 2 static PFs, and 200 VFs (2 PFs x 100):

```
[dpu]# mlxconfig -d 03:00.0 s \
VIRTIO_NET_EMULATION_ENABLE=1 \
VIRTIO_NET_EMULATION_NUM_PF=2 \
VIRTIO_NET_EMULATION_NUM_VF=100 \
VIRTIO_NET_EMULATION_NUM_MSIX=10 \
VIRTIO_NET_EMULATION_NUM_VF_MSIX=6 \
PCI_SWITCH_EMULATION_ENABLE=1 \
PCI_SWITCH_EMULATION_NUM_PORT=15 \
PER_PF_NUM_SF=1 \
PF_TOTAL_SF=256 \
PF_BAR2_ENABLE=0 \
PF_SF_BAR_SIZE=8 \
NUM_VF_MSIX=0 \
SRIOV_EN=1
```

The maximum value of **PCI_SWITCH_EMULATION_NUM_PORT** is 32.
One port is reserved for the emulated PCIe switch upstream connection, so at most 31 hotplug virtio-net devices can be created.
This limit applies with or without virtio-net SR-IOV VFs. Use **virtnet list** and check **max_hotplug_devices** for the live limit.In hotplug virtio-net PFs and virtio-net SR-IOV VFs setups, only up to 32 hotplug devices are supported.

8.15.3.4 System Configuration

8.15.3.4.1 Host System Configuration

For hotplug device configuration, it is recommended to modify the hypervisor OS kernel boot parameters and add the options below:

```
pci=realloc
```

For SR-IOV configuration, first enable SR-IOV from the host.

Refer to MLNX_OFED documentation under Features Overview and Configuration > Virtualization > Single Root IO Virtualization (SR-IOV) > Setting Up SR-IOV for instructions on how to do that.

Make sure to add the following options to Linux boot parameter.

```
intel_iommu=on iommu=pt
```

Add **pci=assign-busses** to the boot command line when creating more than 127 VFs. Without this option, the following errors may trigger from the host and the virtio driver would not probe those devices.

```
pci 0000:84:00.0: [1af4:1041] type 7f class 0xffffff
pci 0000:84:00.0: unknown header type 7f, ignoring device
```

Because the controller from the BlueField side provides hardware resources and acknowledges (ACKs) the request from the host's virtio-net driver, it is mandatory to reboot the host OS (or unload the virtio-net driver) first and the BlueField afterwards. This also applies to reconfiguring a controller from the BlueField platform (e.g., reconfiguring LAG). Unloading the virtio-net driver from host OS side is recommended.

8.15.3.4.2 BlueField System Configuration

Virtio-net full emulation is based on ASAP^2. For each virtio-net device created from host side, there is an SF representor created to represent the device from the BlueField side. It is necessary to have the SF representor in the same OVS bridge of the uplink representor.

The SF representor name is designed in a fixed pattern to map different type of devices.

	Static PF	Hotplug PF	SR-IOV VF
SF Range	1000-1999	2000-2999	3000 and above

For example, the first static PF gets the SF representor of **en3f0pf0sf1000** and the second hotplug PF gets the SF representor of **en3f0pf0sf2001**. It is recommended to verify the name of the SF representor from the **sf_rep_net_device** field in the output of **virtnet list**.

```
[dpu]# virtnet list
{
  ...
  "devices": [
    {
      "pf_id": 0,
      "function_type": "static PF",
      "transitional": 0,
      "vuid": "MT2151X03152VNETS0D0F2",
      "pci_bdf": "14:00.2",
      "pci_vhca_id": "0x2",
      "pci_max_vfs": "0",
      "enabled_vfs": "0",
      "msix_num_pool_size": 0,
      "min_msix_num": 0,
      "max_msix_num": 32,
      "min_num_of_qp": 0,
      "max_num_of_qp": 15,
      "qp_pool_size": 0,
      "num_msix": "64",
      "num_queues": "8",
      "enabled_queues": "7",
      "max_queue_size": "256",
      "msix_config_vector": "0x0",
      "mac": "D6:67:E7:09:47:D5",
      "link_status": "1",
      "max_queue_pairs": "3",
      "mtu": "1500",
      "speed": "25000",
      "rss_max_key_size": "0",
```

```
      "supported_hash_types": "0x0",
      "ctrl_mac": "D6:67:E7:09:47:D5",
      "ctrl_mq": "3",
      "sf_num": 1000,
      "sf_parent_device": "mlx5_0",
      "sf_parent_device_pci_addr": "0000:03:00.0",
      "sf_rep_net_device": "en3f0pf0sf1000",
      "sf_rep_net_ifindex": 15,
      "sf_rdma_device": "mlx5_4",
      "sf_cross_mkey": "0x18A42",
      "sf_vhca_id": "0x8C",
      "sf_rqt_num": "0x0",
      "aarfs": "disabled",
      "dim": "disabled"
    }
  ]
 }
```

Once SF representor name is located, add it to the same OVS bridge of the corresponding uplink representor and make sure the SF representor is up:

```
[dpu]# ovs-vsctl show
f2c431e5-f8df-4f37-95ce-aa0c7da738e0
    Bridge ovsbr1
        Port ovsbr1
            Interface ovsbr1
                type: internal
        Port en3f0pf0sf0
            Interface en3f0pf0sf0
        Port p0
            Interface p0
[dpu]# ovs-vsctl add-port ovsbr1 en3f0pf0sf1000
[dpu]# ovs-vsctl show
f2c431e5-f8df-4f37-95ce-aa0c7da738e0
    Bridge ovsbr1
        Port ovsbr1
            Interface ovsbr1
                type: internal
        Port en3f0pf0sf0
            Interface en3f0pf0sf0
        Port en3f0pf0sf1000
            Interface en3f0pf0sf1000
        Port p0
            Interface p0
[dpu]# ip link set dev en3f0pf0sf1000 up
```

8.15.3.5 Usage

After firmware/system configuration and after system power cycle, the virtio-net devices should be ready to deploy.

First, make sure that **mlxconfig** options take effect correctly by issuing the following command:

> The output has a list with 3 columns: **default** configuration, **current** configuration, and **next-boot** configuration. Verify that the values under the 2nd column match the expected configuration.

```
[dpu]# mlxconfig -d 03:00.0 -e q | grep -i \*
*         PER_PF_NUM_SF                           False(0)         True(1)          True(1)
*         NUM_OF_VFS                              16               0                0
*         PF_BAR2_ENABLE                          True(1)          False(0)         False(0)
*         PCI_SWITCH_EMULATION_NUM_PORT           0                8                8
*         PCI_SWITCH_EMULATION_ENABLE             False(0)         True(1)          True(1)
*         VIRTIO_NET_EMULATION_ENABLE             False(0)         True(1)          True(1)
*         VIRTIO_NET_EMULATION_NUM_VF             0                126              126
*         VIRTIO_NET_EMULATION_NUM_PF             0                1                1
*         VIRTIO_NET_EMULATION_NUM_MSIX           2                64               64
*         VIRTIO_NET_EMULATION_NUM_VF_MSIX        0                64               64
*         PF_TOTAL_SF                             0                508              508
*         PF_SF_BAR_SIZE                          0                8                8
```

If the system is configured correctly, virtio-net-controller service should be up and running. If the service does not appear as active, double check the firmware/system configurations above.

```
[dpu]# systemctl status virtio-net-controller.service
● virtio-net-controller.service - Nvidia VirtIO Net Controller Daemon
```

```
   Loaded: loaded (/etc/systemd/system/virtio-net-controller.service; enabled; vendor preset: disabled)
   Active: active (running)
     Docs: file:/opt/mellanox/mlnx_virtnet/README.md
 Main PID: 30715 (virtio_net_cont)
    Tasks: 55
   Memory: 11.7M
   CGroup: /system.slice/virtio-net-controller.service
           30715 /usr/sbin/virtio_net_controller
           30859 virtio_net_emu
           30860 virtio_net_ha
```

To reload or restart the service, run:

```
[dpu]# systemctl restart virtio-net-controller.service
```

When using "force kill" (i.e., **kill -9** or **kill -SIGKILL**) for the virtio-net-controller service, users should use **kill -9 -<pid of virtio_net_controller process, i.e. 30715 in previous example>** (note the dash "**-**" before the **pid**).

8.15.3.5.1 Hotplug PF Devices

8.15.3.5.1.1 Creating PF Devices

1. To create a hotplug virtio-net device, run:

   ```
   [dpu]# virtnet hotplug -i mlx5_0 -f 0x0 -m 0C:C4:7A:FF:22:93 -t 1500 -n 3 -s 1024
   ```

   Refer to "Virtnet CLI Commands" for full usage.

   This command creates one hotplug virtio-net device with MAC address **0C:C4:7A:FF:22:93**, MTU 1500, and 3 virtio queues with a depth of 1024 entries. The device is created on the physical port of **mlx5_0**. The device is uniquely identified by its index. This index is used to query and update device attributes. If the device is created successfully, an output similar to the following appears:

   ```
   {
     "bdf": "15:00.0",
     "vuid": "MT2151X03152VNETS1D0F0",
     "id": 0,
     "transitional": 0,
     "sf_rep_net_device": "en3f0pf0sf2000",
     "mac": "0C:C4:7A:FF:22:93",
     "errno": 0,
     "errstr": "Success"
   }
   ```

2. Add the representor port of the device to the OVS bridge and bring it up. Run:

   ```
   [dpu]# ovs-vsctl add-port <bridge> en3f0pf0sf2000
   [dpu]# ip link set dev en3f0pf0sf2000 up
   ```

   Once steps 1-2 are completed, the virtio-net PCIe device should be available from hypervisor OS with the same PCIe BDF.

   ```
   [host]# lspci | grep -i virtio
   15:00.0 Ethernet controller: Red Hat, Inc. Virtio network device (rev 01)
   ```

3. Probe virtio-net driver (e.g., kernel driver):

```
[host]# modprobe -v virtio-pci && modprobe -v virtio-net
```

4. The virtio-net device should be created. There are two ways to locate the net device:
   - Check the dmesg from the host side for the corresponding PCIe BDF:

     ```
     [host]# dmesg | tail -20 | grep 15:00.0 -A 10 | grep virtio_net
     [3908051.494493] virtio_net virtio2 ens2f0: renamed from eth0
     ```

   - Check all net devices and find the corresponding MAC address:

     ```
     [host]# ip link show | grep -i "0c:c4:7a:ff:22:93" -B 1
     31: ens2f0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group
     default qlen 1000
         link/ether 0c:c4:7a:ff:22:93 brd ff:ff:ff:ff:ff:ff
     ```

5. Check that the probed driver and its BDF match the output of the hotplug device:

```
[host]# ethtool -i ens2f0
driver: virtio_net
version: 1.0.0
firmware-version:
expansion-rom-version:
bus-info: 0000:15:00.0
supports-statistics: yes
supports-test: no
supports-eeprom-access: no
supports-register-dump: no
supports-priv-flags: no
```

Now the hotplug virtio-net device is ready to use as a common network device.

8.15.3.5.1.2 Destroying PF Devices

To hot-unplug a virtio-net device, run:

```
[dpu]# virtnet unplug -p 0
{'id': '0x1'}
{
  "errno": 0,
  "errstr": "Success"
}
```

The hotplug device and its representor are destroyed.

8.15.3.5.1.3 Host-Aware Attention Button Mode (AB Mode)

The Host-Aware Attention Button (AB) mode provides a coordinated mechanism for device removal utilizing the PCIe Attention Button. This mode ensures the host operating system has the opportunity to gracefully shut down the device driver prior to physical or logical removal.

When utilizing AB mode (**`-w 3`**), the hotplug and unplug lifecycle operates as follows:

1. Hotplug initiation: The device is created with AB mode awareness, a state that persists for the lifetime of the device.
2. Unplug notification: Upon an unplug request, the controller sends an Attention Button notification to the host, signaling a pending device removal.
3. Host acknowledgment: The host OS receives the notification and gracefully shuts down the associated driver.
4. Clean removal: Once the host acknowledges the shutdown, the device is cleanly removed from the system.

Usage Commands

Creating a device with AB mode:

```
[dpu]# virtnet hotplug -i mlx5_0 -f 0x0 -m 0C:C4:7A:FF:22:93 -t 1500 -n 3 -s 1024 -w 3
```

Removing a device with AB mode:

```
[dpu]# virtnet unplug -p 0 -w 3
```

> **Mode Matching**
>
> Devices hotplugged using AB mode (**-w 3**) strictly require the same AB mode flag (**-w 3**) during the unplug operation. Attempting to unplug an AB-mode device without specifying **-w 3** will fail and return a mode mismatch error.

Verification and Querying

Checking AB Mode Support

Run the **virtnet list** command to verify if the underlying controller supports AB mode. Look for the **hp_host_aware_ab_supported** key in the JSON output.

```
{
    "controller": {
        "hp_host_awareness_supported": "1",
        "hp_host_aware_ab_supported": "1"
    }
}
```

Querying Device HP Mode

To verify which hotplug mode a specific device was created with, use the **virtnet query** command. An awareness mode of **3** indicates AB mode.

```
{
    "hp_host_awareness_mode": 3
}
```

Default AB Mode Configuration

To automate AB mode for all hotplug and unplug operations without manually specifying the **-w 3** flag on the CLI, configure the **virtnet.conf** file:

```
force_ab_hotplug_default=1
```

Behavior when enabled:

- Hotplug operations automatically default to AB mode (**-w 3**).
- Unplug operations automatically default to AB mode (**-w 3**).
- Explicit **-w** CLI options will always override this configuration.

- If the underlying Host Channel Adapter (HCA) does not support AB mode, the controller will safely fall back to mode `0`.

8.15.3.5.2 SR-IOV VF Devices

8.15.3.5.2.1 Creating SR-IOV VF Devices

After configuring the firmware and BlueField/host system with correct configuration, users can create SR-IOV VFs.

The following procedure provides an example of creating one VF on top of one static PF:

1. Locate the virtio-net PFs exposed to the host side:

```
[host]# lspci | grep -i virtio
14:00.2 Network controller: Red Hat, Inc. Virtio network device
```

2. Verify that the PCIe BDF matches the backend device from the BlueField side:

```
[dpu]# virtnet list
{
  ...
   "devices": [
    {
      "pf_id": 0,
      "function_type": "static PF",
      "transitional": 0,
      "vuid": "MT2151X03152VNETS0D0F2",
      "pci_bdf": "14:00.2",
      "pci_vhca_id": "0x2",
      "pci_max_vfs": "0",
      "enabled_vfs": "0",
      "msix_num_pool_size": 0,
      "min_msix_num": 0,
      "max_msix_num": 32,
      "min_num_of_qp": 0,
      "max_num_of_qp": 15,
      "qp_pool_size": 0,
      "num_msix": "64",
      "num_queues": "8",
      "enabled_queues": "7",
      "max_queue_size": "256",
      "msix_config_vector": "0x0",
      "mac": "D6:67:E7:09:47:D5",
      "link_status": "1",
      "max_queue_pairs": "3",
      "mtu": "1500",
      "speed": "25000",
      "rss_max_key_size": "0",
      "supported_hash_types": "0x0",
      "ctrl_mac": "D6:67:E7:09:47:D5",
      "ctrl_mq": "3",
      "sf_num": 1000,
      "sf_parent_device": "mlx5_0",
      "sf_parent_device_pci_addr": "0000:03:00.0",
      "sf_rep_net_device": "en3f0pf0sf1000",
      "sf_rep_net_ifindex": 15,
      "sf_rdma_device": "mlx5_4",
      "sf_cross_mkey": "0x18A42",
      "sf_vhca_id": "0x8C",
      "sf_rqt_num": "0x0",
      "aarfs": "disabled",
      "dim": "disabled"
    }
   ]
 }
```

3. Probe `virtio_pci` and `virtio_net` modules from the host:

```
[host]# modprobe -v virtio-pci && modprobe -v virtio-net
```

The PF net device should be created.

```
[host]# ip link show | grep -i "4A:82:E3:2E:96:AB" -B 1
21: ens2f2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group default qlen
1000
    link/ether 4a:82:e3:2e:96:ab brd ff:ff:ff:ff:ff:ff
```

The MAC address and PCIe BDF should match between the BlueField side (**virtnet list**) and host side (**ethtool**).

```
[host]# ethtool -i ens2f2
driver: virtio_net
version: 1.0.0
firmware-version:
expansion-rom-version:
bus-info: 0000:14:00.2
supports-statistics: yes
supports-test: no
supports-eeprom-access: no
supports-register-dump: no
supports-priv-flags: no
```

4. To create SR-IOV VF devices on the host, run the following command with the PF PCIe BDF (**0000:14:00.2** in this example):

```
[host]# echo 1 > /sys/bus/pci/drivers/virtio-pci/0000\:14\:00.2/sriov_numvfs
```

1 extra virtio-net device is created from the host:

```
[host]# lspci | grep -i virtio
14:00.2 Ethernet controller: Red Hat, Inc. Virtio network device (rev 01)
14:00.4 Ethernet controller: Red Hat, Inc. Virtio network device (rev 01)
```

The BlueField side shows the VF information from **virtnet list** as well:

```
[dpu]# virtnet list
    ...
    {
      "vf_id": 0,
      "parent_pf_id": 0,
      "function_type": "VF",
      "transitional": 0,
      "vuid": "MT2151X03152VNETS0D0F2VF1",
      "pci_bdf": "14:00.4",
      "pci_vhca_id": "0xD",
      "pci_max_vfs": "0",
      "enabled_vfs": "0",
      "num_msix": "12",
      "num_queues": "8",
      "enabled_queues": "7",
      "max_queue_size": "256",
      "msix_config_vector": "0x0",
      "mac": "16:FF:A2:6E:6D:A9",
      "link_status": "1",
      "max_queue_pairs": "3",
      "mtu": "1500",
      "speed": "25000",
      "rss_max_key_size": "0",
      "supported_hash_types": "0x0",
      "ctrl_mac": "16:FF:A2:6E:6D:A9",
      "ctrl_mq": "3",
      "sf_num": 3000,
      "sf_parent_device": "mlx5_0",
      "sf_parent_device_pci_addr": "0000:03:00.0",
      "sf_rep_net_device": "en3f0pf0sf3000",
      "sf_rep_net_ifindex": 18,
      "sf_rdma_device": "mlx5_5",
      "sf_cross_mkey": "0x58A42",
      "sf_vhca_id": "0x8D",
      "sf_rqt_num": "0x0",
      "aarfs": "disabled",
      "dim": "disabled"
     }
```

5. Add the corresponding SF representor to the OVS bridge as the virtio-net PF and bring it up. Run:

```
[dpu]# ovs-vsctl add-port <bridge> en3f0pf0sf3000
[dpu]# ip link set dev en3f0pf0sf3000 up
```

Now the VF is functional.

SR-IOV enablement from the host side takes a few minutes. For example, it may take 5 minutes to create 504 VFs.

It is recommended to disable VF autoprobe before creating VFs.

```
[host]# echo 0 > /sys/bus/pci/drivers/virtio-pci/<virtio_pf_bdf>/sriov_drivers_autoprobe
[host]# echo <num_vfs> > /sys/bus/pci/drivers/virtio-pci/<virtio_pf_bdf>/sriov_numvfs
```

Users can pass through the VFs directly to the VM after finishing. If using the VFs inside the hypervisor OS is required, bind the VF PCIe BDF:

```
[host]# echo <virtio_vf_bdf> > /sys/bus/pci/drivers/virtio-pci/bind
```

Keep in mind to reenable the autoprobe for other use cases:

```
[host]# echo 1 > /sys/bus/pci/drivers/virtio-pci/<virtio_pf_bdf>/sriov_drivers_autoprobe
```

MAC addresses are randomly generated for the new virtual functions (VFs).

Creating VFs for the same PF on different threads may cause the hypervisor OS to hang.

8.15.3.5.2.2 Destroying SR-IOV VF Devices

To destroy SR-IOV VF devices on the host, run:

```
[host]# echo 0 > /sys/bus/pci/drivers/virtio-pci/<virtio_pf_bdf>/sriov_numvfs
```

When the echo command returns from the host OS, it does not necessarily mean the BlueField side has finished its operations. To verify that the BlueField is done, and it is safe to recreate the VFs, either:

- Check controller log from the BlueField and make sure you see a log entry similar to the following:

  ```
  [dpu]# journalctl -u virtio-net-controller.service -n 3 -f
  virtio-net-controller[5602]: [INFO] virtnet.c:675:virtnet_device_vfs_unload: static PF[0], Unload
  (1) VFs finished
  ```

- Query the last VF from the BlueField side:

  ```
  [dpu]# virtnet query -p 0 -v 0 -b
  {'all': '0x0', 'vf': '0x0', 'pf': '0x0', 'dbg_stats': '0x0', 'brief': '0x1', 'latency_stats':
  '0x0', 'stats_clear': '0x0'}
  {
    "Error": "Device doesn't exist"
  }
  ```

⚠ Once VFs are destroyed, SFs created for virtio-net from the BlueField side are not destroyed but are saved into the SF pool for reuse later.

⚠ Restarting virtio-net-controller service while performing device create/destroy for either hotplug or VF is unsupported.

8.15.3.6 Assigning Virtio-net Device to VM

All virtio-net devices (static/hotplug PF and VF) support PCIe passthrough to a VM. PCIe passthrough allows the device to get better performance in the VM.

Assigning a virtio-net device to a VM can be done via **virt-manager** or **virsh command**.

8.15.3.6.1 Locating Virtio-net Devices

All virtio-net devices can be scanned by the PCIe subsystem in hypervisor OS and displayed as a standard PCIe device. Run the following command to locate the virtio-net devices devices with its PCIe BDF.

```
[host]# lspci | grep 'Virtio network'
00:09.1 Ethernet controller: Red Hat, Inc. Virtio network device (rev 01)
```

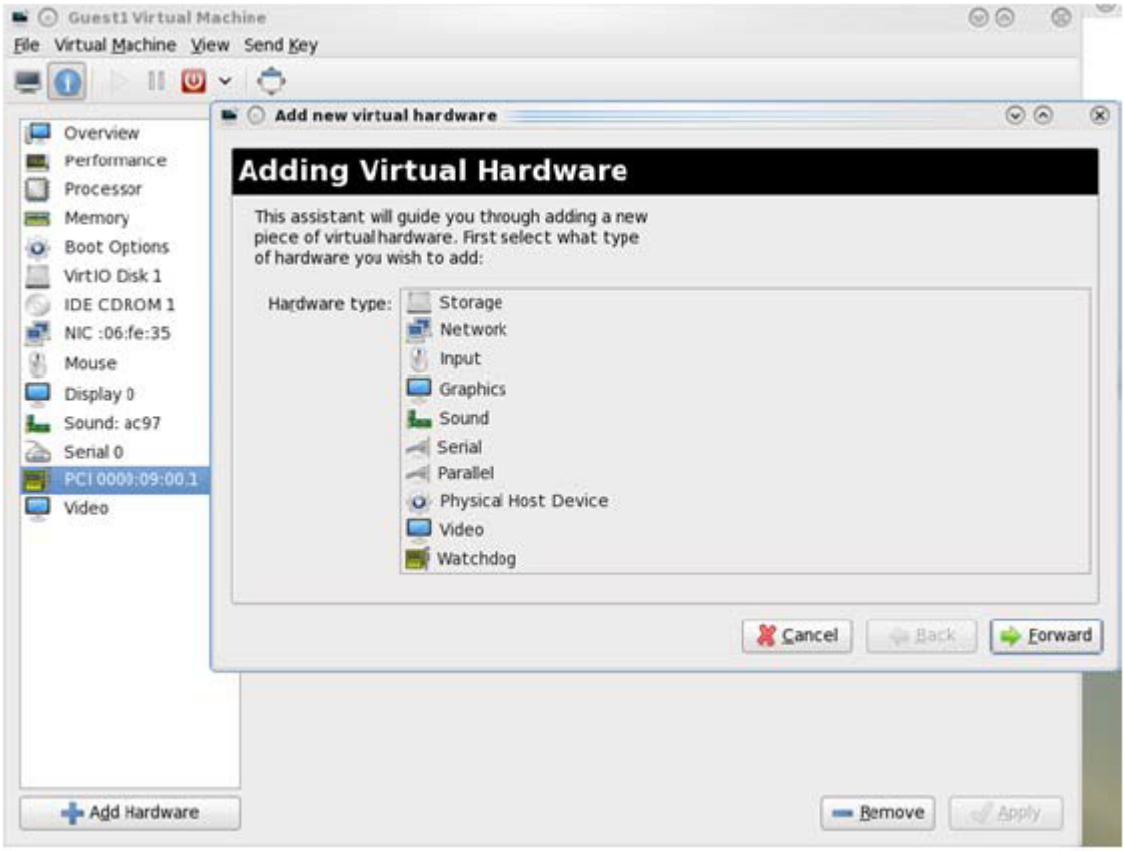
8.15.3.6.2 Using virt-manager

Start **virt-manager**, run the following command:

```
[host]# virt-manager
```

⚠ Make sure your system has xterm enabled to show the virt-manager GUI.

Double-click the virtual machine and open its Properties. Navigate to Details → Add hardware → PCIe host device.



Choose a virtio-net device virtual function according to its PCIe device (e.g., 00:09.1), reboot or start the VM.

8.15.3.6.3 Using virsh Command

1. Run the following command to get the VM list and select the target VM by **Name** field:

```
[host]# virsh list --all
 Id    Name                           State
----------------------------------------------------
 1     host-101-CentOS-8.5            running
```

2. Edit the VMs XML file, run:

```
[host]# virsh edit <VM_NAME>
```

3. Assign the target virtio-net device PCIe BDF to the VM, using **vfio** as driver, replace **BUS/SLOT/FUNCTION/BUS_IN_VM/SLOT_IN_VM/FUNCTION_IN_VM** with corresponding settings.

```
<hostdev mode='subsystem' type='pci' managed='no'>
  <driver name='vfio'/>
    <source>
      <address domain='0x0000' bus='<#BUS>' slot='<#SLOT>' function='<#FUNCTION>'/>
    </source>
  <address type='pci' domain='0x0000' bus='<#BUS_IN_VM>' slot='<#SLOT_IN_VM>' function='<#FUNCTION_IN_VM>'/
>
</hostdev>
```

For example, assign target device **00.09.1** to the VM and its PCIe BDF within the VM is **01:00.0**:

```
<hostdev mode='subsystem' type='pci' managed='no'>
  <driver name='vfio'/>
    <source>
      <address domain='0x0000' bus='0x00' slot='0x09' function='0x1'/>
    </source>
  <address type='pci' domain='0x0000' bus='0x01' slot='0x00' function='0x0'/>
</hostdev>
```

4. Destroy the VM if it is already started:

```
[host]# virsh destory <VM_NAME>
```

5. Start the VM with new XML configuration:

```
[host]# virsh start <VM_NAME>
```

8.15.4 Configuration File

8.15.4.1 Configuration File Options

The controller service has an optional JSON format configuration file which allows users to customize several parameters. The configuration file should be defined on the DPU at **/opt/mellanox/mlnx_virtnet/virtnet.conf**. This file is read every time the controller starts.

Controller **systemd** service should be restarted when there is configuration file change. Dynamic change of **virtnet.conf** is not supported.

In DOCA 3.3.0, the **mrg_rxbuf** and **packed_vq** parameters will no longer be supported in the **virtnet.conf** configuration file. Configuration of the **mrg_rxbuf** and **packed_vq** features must now be performed using the CLI modify device option.

Parameter	DefaultValue	Type	Description
ib_dev_p0	**mlx5_0**	String	RDMA device (e.g., **mlx5_0**) used to create SF on port 0. This port is the EMU manager when **is_lag** is 0.
ib_dev_p1	**mlx5_1**	String	RDMA device (e.g., **mlx5_1**) used to create SF on port 1

<table>
<tr><th>Parameter</th><th>DefaultValue</th><th>Type</th><th>Description</th></tr>
<tr><td>ib_dev_for_static_pf</td><td>mlx5_0</td><td>String</td><td>The RDMA device (e.g., mlx5_0) which the static virtio PF is created on</td></tr>
<tr><td>ib_dev_lag</td><td>Null</td><td>String</td><td>RDMA LAG device (e.g., mlx5_bond_0) used to create SF on LAG. Default value is mlx5_bond_0. This port is EMU manager when is_lag is 1. ib_dev_lag and ib_dev_p0 / ib_dev_p1 cannot be configured simultaneously.</td></tr>
<tr><td>static_pf</td><td>N/A</td><td>List</td><td>The following sub-parameters can be used to configure the static PF:
<table>
<tr><th>Sub-parameter</th><th>Default Value</th><th>Type</th><th>Description</th></tr>
<tr><td>mac_base</td><td>Null</td><td>String</td><td>Base MAC address for static PFs. MACs are automatically assigned with the following pattern: mac_base → pf_0, mac_base + 1 → pf_1, etc.
⚠ Controller does not validate the MAC address (other than its length). The user must ensure the MAC is valid and unique.
⚠ The virtio driver on the host OS must be unloaded when restarting the controller if the mac_base setting is enabled for the first time or is modified.</td></tr>
<tr><td>features</td><td>Auto</td><td>Number</td><td>Virtio spec-defined feature bits for static PFs.
⚠ If unsure, leave features out of the JSON file and a default value is automatically assigned. The default value is determined dynamically when controller starts. Refer to section "DOCA Virtio-net Service Guide#Feature Bits" for more information.</td></tr>
<tr><td>mtu</td><td>1500</td><td>Number</td><td>The maximum transmission unit for the PF, can be ≤ 9216</td></tr>
</table></td></tr>
</table>

Parameter	DefaultValue	Type	Description
`is_lag`	`0`	Number	Specifies whether LAG is used ⚠ If LAG is used, make sure to use the correct IB dev for static PF
`single_port`	`0`	Number	Specifies whether the DPU is a single port device. It is mutually exclusive with `is_lag`.
`recovery`	`1`	Number	Specifies whether recovery is enabled. If unspecified, recovery is enabled by default. To disable it, set `recovery` to 0. Refer to section "DOCA Virtio-net Service Guide#Recovery" for the items which are recovered and more information.
`sf_pool_percent`	`0`	Number	Determines the initial SF pool size as the percentage of `PF_TOTAL_SF` of `mlxconfig`. Valid range: [0, 100]. For instance, if the value is 5, an SF pool with 5% of `PF_TOTAL_SF` is created. 0 indicates that no SF pool is reserved beforehand (default). ⚠ `PF_TOTAL_SF` is shared by all applications. The user must ensure that the percent request is guaranteed, or else the controller would not be able to reserve the requested SFs resulting in failure.
`sf_pool_force_destroy`	`0`	Number	Specifies whether to destroy the SF pool. When set to 1, the controller destroys the SF pool when stopped/restarted (and the SF pool is recreated if `sf_pool_percent` is not 0 when starting). Otherwise, it does not. Default value is 0.
`dpa_core_start`	`0`	Number	Specifies the start DPA core for virtnet application. Valid only for NVIDIA® BlueField®-3 and up. Value must be greater than 0 and less than 11. Together with `dpa_core_end`, `dpa_core_start` defines how many DPA cores are used for the virtio-net data plane. ⚠ This is advanced options when there are multiple DPA applications running at the same time. Regular user should keep this option as default. ⚠ The number of cores/EUs impacts the maximum number of VQs that can be created.
`dpa_core_end`	`10`	Number	Specifies the end DPA core for virtnet application. Valid only for BlueField-3 and up. Value must be greater than `dpa_core_start` and less than 11.

<table>
<tr><th>Parameter</th><th>DefaultValue</th><th>Type</th><th>Description</th></tr>
<tr><td>vf</td><td>N/A</td><td>List</td><td>The following sub-parameters can be used to configure the VF:
<table>
<tr><th>Sub-parameter</th><th>Default Value</th><th>Type</th><th>Description</th></tr>
<tr><td>mac_base</td><td>Null</td><td>String</td><td>Base MAC address for VFs. MACs are automatically assigned with the following pattern: mac_base → vf_0, mac_base + 1 → vf_1, etc.
⚠ Controller does not validate the MAC address (other than its length). The user must ensure the MAC is valid and unique.
⚠ The virtio driver on the guest OS must be unloaded when restarting the controller if the mac_base setting is enabled for the first time or is modified.</td></tr>
<tr><td>features</td><td>Auto</td><td>Number</td><td>Virtio spec-defined feature bits for VFs.
⚠ If unsure, leave features out of the JSON file and a default value is automatically assigned. The default value is determined dynamically when controller starts. Refer to section "DOCA Virtio-net Service Guide#Feature Bits" for more information.</td></tr>
<tr><td>vfs_per_pf</td><td>0</td><td>Number</td><td>The number of VFs to create on each PF. For example: if vfs_per_pf is 100, then vf_0 on pf_1 will use mac_base + 100 as its MAC.
⚠ vfs_per_pf ≤ VIRTIO_NET_EMULATION_NUM_VF in mlxconfig.
⚠ User is responsible for ensuring, on each static PF, that the created VFs ≤ vfs_per_pf.
⚠ This parameter is mandatory if mac_base is specified.</td></tr>
</table></td></tr>
</table>

<table>
<tr><th>Parameter</th><th>DefaultValue</th><th>Type</th><th>Description</th></tr>
<tr><td></td><td></td><td></td><td>
<table>
<tr><th>Sub-parameter</th><th>DefaultValue</th><th>Type</th><th>Description</th></tr>
<tr><td>max_queue_pairs</td><td>Auto</td><td>Number</td><td>Number of queue pairs to use. If not specified, default queue pair number is inherited from the parent PF.</td></tr>
<tr><td>max_queue_size</td><td>Auto</td><td>Number</td><td>Virtqueue size (i.e., vq depth) to use. If not specified, default vq size is inherited from the parent PF.</td></tr>
<tr><td>mtu</td><td>1500</td><td>Number</td><td>Maximum transmission unit for the VF, can be ≤ 9216.</td></tr>
</table>
</td></tr>
<tr><td>virtio_spec_admin_legacy</td><td>0</td><td>Number</td><td>Enable (1) or disable (0) virtio spec legacy interface commands.</td></tr>
<tr><td>virtio_spec_admin_lm</td><td>0</td><td>Number</td><td>Enable (1) or disable (0) virtio spec live migration commands.</td></tr>
</table>

Parameter	DefaultValue	Type	Description

`dpa_partition`	`N/A`	String	DPA partition configuration file full path. Refer to section "DOCA Virtio-net Service Guide#DPA Configuration (SPRD)" for more information.
			The DPA partition conf file is generated by the DOCA dpa-resource-mgmt tool. Refer to DOCA DPA Tools
			Configuration requirements:
			• App name is `virtio-net`
			• The minimum number of EUs is 32
			• EU groups are not supported
			Example of `config.yaml` input to `dpa-resource-mgmt`:
			`---`
			`version: 25.04`
			`---`
			`DPA_APPS:`
			`  virtio-net:`
			`  - partition: ROOT`
			`  affinity_core:`
			`  - core: 1`
			`  num_EUs: 16`
			`  - core: 3`
			`  num_EUs: 16`
			`  - core: 5`
			`  num_EUs: 16`
			⚠ **Operational Constraint: ECPF Partitioning**
			For `virtio-net` deployments on the DPU, the DPA application resides on the ECPF device. Consequently, ECPF interactions are restricted to the default ROOT partition. You must utilize `dpa-resource-mgmt` to initialize the ROOT partition on the active device (e.g., `mlx5_0`, `ib_dev_p0`, or `ib_dev_lag`).
			⚠ **Parameter Precedence**
			Specifying a `dpa_partition` overrides individual core allocations, causing the `dpa_core_start` and `dpa_core_end` parameters to be ignored.
			⚠ The number of cores/EUs impacts the maximum number of VQs that can be created.

`eth_vq_workers`	`0`	Number	Specifies the number of worker threads allocated for Ethernet Virtual Queue (VQ) lifecycle operations, such as queue creation and destruction. • Performance tuning: Because these operations are strictly I/O-bound and subject to firmware latency, configuring more threads than available CPU cores yields better performance. The underlying firmware supports 8-way parallelism, making an allocation of 8 to 24 workers optimal. • Auto-calculation: If left undefined (or set to 0), the system automatically provisions workers using the formula `num_cpu*4`, bounded by a minimum of `8` and a maximum of `24`.

<table>
<tr><th>Parameter</th><th>DefaultValue</th><th>Type</th><th>Description</th></tr>
<tr><td>admin_cmd_workers</td><td>0</td><td>Number</td><td>Specifies the number of worker threads allocated for administrative command operations.
Performance tuning: Similar to ETH VQ operations, administrative commands are I/O-bound. Provisioning more threads than available CPU cores enhances responsiveness. This parameter is designed to provide a conservative baseline for administrative command parallelism.
Auto-calculation: If left undefined (or set to 0), the system automatically provisions workers using the formula num_cpu*2, bounded by a minimum of 4 and a maximum of 16.</td></tr>
<tr><td>force_ab_hotplug_default</td><td>0</td><td>Number</td><td>To automate AB mode for all hotplug and unplug operations without manually specifying the -w 3 flag on the CLI.</td></tr>
<tr><td>event_publisher</td><td>N/A</td><td>JSON object</td><td>Configuration for virtio-net event notifications.
<table>
<tr><th>Sub-parameter</th><th>Default Value</th><th>Type</th><th>Description</th></tr>
<tr><td>enabled</td><td>false</td><td>Boolean</td><td>Master enable/disable switch</td></tr>
<tr><td>broker_url</td><td>""</td><td>String</td><td>Must be nats://127.0.0.1:4222</td></tr>
<tr><td>subject_prefix</td><td>"virtio.vf"</td><td>String</td><td>Prefix for NATS subjects</td></tr>
<tr><td>connect_timeout_ms</td><td>2000</td><td>Integer</td><td>Broker connect attempt timeout in milliseconds. Range: 100–30000. 0 uses the default.</td></tr>
<tr><td>reconnect_backoff_ms</td><td>1000</td><td>Integer</td><td>Backoff between reconnect attempts in milliseconds. Range: 100–60000. 0 uses the default.</td></tr>
<tr><td>max_queue_depth</td><td>4096</td><td>Integer</td><td>Bounded publish queue depth. Range: 16–65536. 0 uses the default.</td></tr>
</table>
If the event_publisher section is entirely missing from virtnet.conf, event publishing is disabled by default, which means the controller starts normally with zero overhead.</td></tr>
</table>

8.15.4.2 Configuration File Examples

> ⚠ Validate the JSON format of the configuration file before restarting the controller, especially the syntax and symbols. Otherwise, the controller may fail to start.

8.15.4.2.1 Configuring LAG on Dual Port BlueField

Refer to "Link Aggregation" documentation for information on configuring BlueField in LAG mode.

Refer to the "Link Aggregation" page for information on configuring virtio-net in LAG mode.

8.15.4.2.2 Configuring Static PF on Dual Port BlueField

The following configures all static PFs to use **mlx5_0** (port 0) as the data path device in a non-LAG configuration, and the default MAC and features for the PF:

```
{
  "ib_dev_p0": "mlx5_0",
  "ib_dev_p1": "mlx5_1",
  "ib_dev_for_static_pf": "mlx5_0",
  "is_lag": 0,
  "static_pf": {
    "mac_base": "08:11:22:33:44:55",
    "features": "0x230047082b"
  }
}
```

8.15.4.2.3 Configuring VF Specific Options

The following configures VFs with default parameters. With this configuration, each PF assigns the MAC based on **mac_base** up to 126 VFs. Each VF creates 4 queue pairs, with each queue having a depth of 256.

> ⚠ If **vfs_per_pf** is less than the **VIRTIO_NET_EMULATION_NUM_VF** in mlxconfig, and more VFs are created, duplicated MACs would be assigned to different VFs.

```
{
  "vf": {
    "mac_base": "06:11:22:33:44:55",
    "features": "0x230047082b",
    "vfs_per_pf": 126,
    "max_queue_pairs": 4,
    "max_queue_size": 256
  }
}
```

8.15.4.2.4 Virtio Live Migration Settings

The following table provides an example of configurations for the new options introduced for VirtIO Live Migration:

virtio_spec_admin_legacy	virtio_spec_admin_lm	Expected Result
`1`	`1`	Enables both legacy interface and VFIO kernel Live Migration commands
`1`	`0`	Enables legacy interface commands only
`0`	`1`	Enables VFIO kernel Live Migration commands only
`0`	`0`	Disable both legacy interface and VFIO kernel live migration commands
`-`	`-`	Supports VDPA Live Migration solutions

8.15.4.2.5 DPA Configuration (SPRD)

The Single Point of Resource Distribution (SPRD) is a centralized orchestration system that manages Data Path Accelerator (DPA) Execution Unit (EU) allocation across applications and Virtual HCAs (VHCAs). By centralizing this process, SPRD prevents ad-hoc per-application sizing and eliminates resource over-commitment.

Administrators define EU assignments to partitions and applications within a single YAML configuration file. The SPRD system validates this configuration, programs the hardware partitions and EU groups accordingly, and generates per-partition configuration files for the applications to consume.

8.15.4.2.5.1 Control and Integration

- CLI utility: The system is managed using the **`dpa-resource-mgmt`** command-line tool.
- Virtio-net integration: The **`virtio-net`** controller consumes the SPRD-generated output file via the **`dpa_partition`** configuration option to inherit its managed EU affinity.

8.15.4.2.5.2 SPRD Configuration Workflow for Virtio-net

Inspect Available Resources

Query the DPU to verify the **`ROOT`** partition and determine the set of free EUs before assigning them to **`virtio-net`**.

```
dpa-resource-mgmt query -t resources -d mlx5_0
```

Create an Input SPRD YAML

Create an input file (e.g., **`input_vnet.yaml`**) that explicitly maps the **`virtio-net`** application to the **`ROOT`** partition and defines the required EUs per core.

Example for a virtio-net "solo app" deployment:

```
version: 26.01
DPA_APPS:
   "virtio-net":
       - partition: ROOT
         affinity_core:
             - core: 0
```

```
      num_EUs: 16
    - core: 1
      num_EUs: 16
    - core: 2
      num_EUs: 16
    - core: 3
      num_EUs: 16
    - core: 4
      num_EUs: 16
    - core: 5
      num_EUs: 16
    - core: 6
      num_EUs: 16
    - core: 7
      num_EUs: 16
    - core: 8
      num_EUs: 16
    - core: 9
      num_EUs: 16
```

Generate the SPRD Output YAML

Run the configuration command to generate the output file.

```
dpa-resource-mgmt config -d mlx5_0 -f input_vnet.yaml -v
```

> **Naming Conventions**
>
> When utilizing the **ROOT** partition on a DPU, the generated output file automatically inherits the partition name (e.g., **ROOT.yaml**). This is the exact file you must reference in the **virtio-net** controller using the **dpa_partition** configuration option.

8.15.4.2.5.3 Example SPRD Output & Validation

The generation command produces a parsed YAML file utilized by the application. The **affinity_EUs** list contains the absolute EU IDs reserved for **virtio-net**. During execution, SPRD converts these absolute IDs into partition-relative indices.

Example **ROOT.yaml** Output:

```
version: 26.01
DPA_APPS:
-   name: virtio-net
    number_of_affinity_EUs: 160
    affinity_EUs: [0, 1, 2, 4, 5, 6, 8, 9, 10, 11, 12, 13, 15, 16, 17, 19, 21, 22,
        23, 24, 26, 27, 28, 29, 30, 31, 33, 34, 36, 37, 39, 40, 41, 42, 43, 44, 45,
        48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 60, 61, 62, 63, 65, 66, 67, 69,
        70, 71, 72, 73, 74, 75, 76, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 89, 90,
        92, 93, 94, 95, 96, 97, 98, 99, 101, 102, 103, 104, 105, 106, 107, 108, 109,
        110, 111, 112, 113, 114, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125,
        126, 127, 129, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 143,
        144, 145, 147, 148, 149, 150, 151, 152, 154, 155, 156, 157, 158, 159, 160,
        161, 162, 163, 165, 166, 167, 168, 169, 172, 173, 174, 175, 177, 178, 179,
        181, 182, 183, 184, 185, 187, 188, 189]
```

Performance Validation (BlueField-3 DPU)

This specific SPRD configuration was validated in a "solo app" scenario (where **virtio-net** was the exclusive DPA application assigned EUs) to ensure all data-plane threads ran optimally across the 160 managed EUs.

Hardware and test profile:

- Devices: 32 full emulation virtio-net VFs (16 TX, 16 RX)
- Queue Configuration: Virtqueue (VQ) depth of 1024; 31 Queue Pairs (QPs) per device.
- CQE Moderation: Count = 32, Period = 32

- Traffic Profile: `testpmd` UDP, 16 streams, 64-byte message size.

Measured aggregate performance:

- RX-Only: 72.6 Mpps
- TX-Only: 81.05 Mpps
- Bidirectional (RX + TX): 100.36 Mpps total

8.15.5 Virtnet CLI Commands

8.15.5.1 User Front End CLI

To communicate with the virtio-net-controller backend service, a user frontend program, virtnet, is installed on the BlueField which is based on remote procedure call (RPC) protocol with JSON format output.

8.15.5.2 Hotplug

This command hotplugs a virtio-net PCIe PF device exposed to the host side.

8.15.5.2.1 Syntax

```
virtnet hotplug -i IB_DEVICE -m MAC -t MTU -n MAX_QUEUES -s MAX_QUEUE_SIZE [-h] [-u SF_NUM] [-f FEATURES] [-l] [-w HP_HOST_AWARENESS]
```

Option	Abbr	Argument Type	Required	Description
`--help`	`-h`	N/A	No	Show the help message and exit
`--ib_device`	`-i`	String	Yes	RDMA device (e.g., `mlx5_0`) of the physical port on top of which the hotplug device is created. Options: • `mlx5_0` – port 1 • `mlx5_1` – port 2 • `mlx5_bond_0` – LAG
`--features`	`-f`	Hex Number	No	Feature bits to be enabled in hex format. Refer to the "Virtio-net Feature Bits" page. ⚠ Note that some features are enabled by default. Query the device to show the supported bits.

Option	Abbr	Argument Type	Required	Description
`--mac`	`-m`	Number	Yes	MAC address of the virtio-net device. ⚠ Controller does not validate the MAC address (other than its length). The user must ensure MAC is valid and unique.
`--mtu`	`-t`	Number	Yes	Maximum transmission unit (MTU) size of the virtio-net device. It must be less than the uplink rep MTU size.
`--num_queues`	`-n`	Number	Yes	Mutually exclusive with `max_queue_pairs`. Max number of virt queues could be created for the virtio-net device. TX, RX, ctrl queues are counted separately (e.g., 3 has 1 TX VQ, 1 RX VQ, 1 Ctrl VQ). ⚠ This option will be depreciated in the future.
`--max_queue_pairs`	`-qp`	Number	Yes	Mutually exclusive with `num_queues`. Number of data VQ pairs. One VQ pair has one TX queue and one RX queue. It does not count control or admin VQ. From the host side, it appears as `Pre-set maximums->Combined` in `ethtool -l <virtio-dev>`.
`--max_queue_size`	`-s`	Number	Yes	Maximum number of buffers in the virt queue, between 0x4 and 0x8000. Must be power of 2.
`--sf_num`	`-u`	Number	No	SF number to be used for this hotplug device, must between 2000 and 2999.
`--legacy`	`-l`	N/A	No	Create legacy (transitional) hotplug device ⚠ Relevant for BlueField-2 only.

<table>
<tr><th>Option</th><th>Abbr</th><th>Argument Type</th><th>Required</th><th>Description</th></tr>
<tr><td><code>--hp_host_awareness</code></td><td><code>-w</code></td><td>Number</td><td>No</td><td>This setting determines how the device interacts with the host during hot plug operations. The following modes are available:
<table>
<tr><th>Value</th><th>Mode</th><th>Description</th></tr>
<tr><td><code>0</code></td><td>Device default (Default)</td><td>The device operates in its default mode. For virtio-net, the default is host-aware.</td></tr>
<tr><td><code>1</code></td><td>Host-aware</td><td>The device monitors the host’s state and proceeds only after the host has completed initialization (e.g., after the PCIe bus has been fully scanned) and is ready for new PCIe connections</td></tr>
<tr><td><code>2</code></td><td>Host-unaware</td><td>The hot plug operation can proceed regardless of the host’s state. This means the operation may complete even if the host has not finished initialization or cannot respond to events.
Using this mode may result in undefined behavior on systems with older BIOS or OS versions. It is not recommended for such environments.</td></tr>
<tr><td><code>3</code></td><td>Host-aware attention button</td><td>The device will be attached /de-attached in a graceful mode, using the PCIe attention button sequence.</td></tr>
</table>
For optimal stability and compatibility, it is recommended to use either the device default mode (<code>0</code>) or host-aware mode (<code>1</code>).
Always ensure that your system's BIOS and OS are up to date to avoid compatibility issues with hot plug features.
The <code>virtnet list</code> CLI command indicates whether the controller supports hot plug host awareness:
<pre>
#virtnet list
{
        "controller":   {
                …
                "emulation_manager":   "mlx5_0",
                "max_hotplug_devices": "15",
                "hp_host_awareness_supported": "1",   // <===== new indication
                "max_virt_net_devices": "15",
                "max_virt_queues":   "256",
                ...
</pre></td></tr>
</table>

8.15.5.2.2 Output

Entry	Type	Description
bdf	String	The PCIe BDF (bus:device:function) number enumerated by host. The user should see this PCIe device from host side.
vuid	String	Unique device SN. It can be used as an index to query/modify/unplug this device.
id	Num	Unique device ID. It can be used as an index to query/modify/unplug this device.
transitional	Num	Is the current device a transitional hotplug device. • 0 – modern device • 1 – transitional device
sf_rep_net_device	String	The SF representor name represents the virtio-net device. It should be added into the OVS bridge.
mac	String	The hotplug virtio-net device MAC address
errno	Num	Error number if hotplug failed. • 0 – success • non-0 – failed
errstr	String	Explanation of the error number

8.15.5.2.3 Example

The following example of hot plugging one device with MAC address `0C:C4:7A:FF:22:93`, MTU 1500, and 1 pair of virtual queue (QP) pair with a depth of 1024 entries. The device is created on the physical port of `mlx5_0`.

```
# virtnet hotplug -i mlx5_0 -m 0C:C4:7A:FF:22:93 -t 1500 -qp 1 -s 1024
{
  "bdf": "15:00.0",
  "vuid": "MT2151X03152VNETS1D0F0",
  "id": 0,
  "transitional": 0,
  "sf_rep_net_device": "en3f0pf0sf2000",
  "mac": "0C:C4:7A:FF:22:93",
  "errno": 0,
  "errstr": "Success"
}
```

8.15.5.3 Unplug

This command unplugs a virtio-net PCIe PF device.

8.15.5.3.1 Syntax

```
virtnet unplug [-h] [-p PF | -u VUID] [-w HP_HOST_AWARENESS] [-T HOTPLUG_POWER_OFF_TIMEOUT]
```

Only one of `--pf` and `--vuid` is needed to unplug the device.

Option	Abbr	ArgumentType	Required	Description
`--help`	`-h`	N/A	No	Show the help message and exit
`--pf`	`-p`	Number	Yes	Unique device ID returned when doing hotplug. Can be retrieved by using `virtnet list`.
`--vuid`	`-u`	String	Yes	Unique device SN returned when doing hotplug. Can be retrieved by using `virtnet list`.

<table>
<tr><th>Option</th><th>Abbr</th><th>ArgumentType</th><th>Required</th><th>Description</th></tr>
<tr><td><code>--hp_host_awareness</code></td><td><code>-w</code></td><td>Number</td><td>No</td><td>This setting determines how the device interacts with the host during hot unplug operations. The following modes are available:
<table>
<tr><th>Value</th><th>Mode</th><th>Description</th></tr>
<tr><td><code>0</code></td><td>Device default (Default)</td><td>The device operates in its default mode. For virtio-net, the default is host-aware.</td></tr>
<tr><td><code>1</code></td><td>Host-aware</td><td>The device monitors the host's state and proceeds only after the host has completed initialization (e.g., after the PCIe bus has been fully scanned) and is ready for new PCIe connections</td></tr>
<tr><td><code>2</code></td><td>Host-unaware</td><td>The hot unplug operation can proceed regardless of the host's state. This means the operation may complete even if the host has not finished initialization or cannot respond to events.
Using this mode may result in undefined behavior on systems with older BIOS or OS versions. It is not recommended for such environments.</td></tr>
</table>
For optimal stability and compatibility, it is recommended to use either the device default mode (<code>0</code>) or host-aware mode (<code>1</code>).
Always ensure that your system's BIOS and OS are up to date to avoid compatibility issues with hot unplug features.
The <code>virtnet list</code> CLI command indicates whether the controller supports hot unplug host awareness:
<pre>
#virtnet list

{
        "controller":   {
                …
                "emulation_manager":   "mlx5_0",
                "max_hotplug_devices": "15",
                "hp_host_awareness_supported": "1",   // <===== new
indication
                "max_virt_net_devices": "15",
                "max_virt_queues":   "256",
                ...
</pre></td></tr>
</table>

Option	Abbr	ArgumentType	Required	Description
`--hotplug_poweroff_timeout`	`-T`	Number	No	Specifies the duration (in seconds) the controller waits for the host OS to power off during an unplug operation before executing a forced unplug. • Valid range: `1` to `900` seconds. • Default value: `10` seconds. • Important constraint: This timeout parameter is not applicable and will be ignored if the device is operating in Host-Aware Attention Button (AB) mode.

8.15.5.3.2 Output

Entry	Type	Description
`errno`	Num	Error number if operation failed • 0 – success • non-0 – failed
`errstr`	String	Explanation of the error number

8.15.5.3.3 Example

Unplug-hotplug device using the PF ID:

```
# virtnet unplug -p 0
{'id': '0x1'}
{
  "errno": 0,
  "errstr": "Success"
}
```

8.15.5.4 List

This command lists all existing virtio-net devices, with global information and individual information for each device.

8.15.5.4.1 Syntax

```
virtnet list [-h] [-c | -d [-p | -v [-p ID]] | -f]
```

The options `--controller`, `--devices` and `--config-file` are mutually exclusive, `--pf` and `--vf` are valid only with `--devices`.

Option	Abbr	Argument Type	Required	Description
`--help`	`-h`	N/A	No	Show the help message and exit
`--controller`	`-c`	N/A	No	Prints only the global information for the controller
`--devices`	`-d`	N/A	No	Prints device information only
`--pf`	`-p`	Number	No	List PF devices only: `virtnet list -d -p`
`--vf`	`-v`	N/A	No	List VF devices only: `virtnet list -d -v` List VF devices of a parent PF: `virtnet list -d -v -p 0`
`--config-file`	`-f`	N/A	No	Prints the effective parsed configuration loaded from `virtnet.conf`

8.15.5.4.2 Output

The output has two main sections. The first section wrapped by the `controller` are global configurations and capabilities.

Entry	Type	Description
`controller`	String	Entries under this section is global information for the controller
`emulation_manager`	String	The RDMA device manager used to manage internal resources. Should be default `mlx5_0`.
`max_hotplug_devices`	String	Maximum number of devices that can be hotpluged
`max_virt_net_devices`	String	Total number of emulated devices managed by the device emulation manager
`max_virt_queues`	String	Maximum number of virt queues supported per device
`max_tunnel_descriptors`	String	Maximum number of descriptors the device can send in a single tunnel request
`supported_features`	String	Total list of features supported by device
`supported_virt_queue_types`	String	Currently supported virt queue types: Packed and Split
`supported_event_modes`	String	Currently supported event modes: `no_msix_mode`, `qp_mode`, `msix_mode`
`hp_host_awareness_supported`	String	Indicates whether hot plug host awareness is supported

Each device has its own section under `devices`.

Entry	Type	Description
devices	String	Entries under this section is per device information
pf_id	Number	Physical function ID
function_type	String	Function type: Static PF, hotplug PF, VF
transitional	Number	The current device a transitional hotplug device: • 0 – modern device • 1 – transitional device
vuid	String	Unique device SN, it can be used as an index to query/modify/unplug a device
pci_bdf	String	Bus:device:function to describe the virtio-net PCIe device
pci_vhca_id	Number	Virtual HCA identifier for the general virtio-net device. For debug purposes only.
pci_max_vfs	Number	Maximum number of virtio-net VFs that can be created for this PF. Valid only for PFs.
enabled_vfs	Number	Currently enabled number of virtio-net VFs for this PF
msix_num_pool_size	Number	Number of free dynamic MSIX available for the VFs on this PF
min_msix_num	Number	The minimum number of dynamic MSI-Xs that can be set for an virtio-net VF
max_msix_num	Number	The maximum number of dynamic MSI-Xs that can be set for an virtio-net VF
min_num_of_qp	Number	The minimum number of dynamic data VQ pairs (i.e., each pair has one TX and 1 RX queue) that can be set for an virtio-net VF

Entry	Type	Description
`max_num_of_qp`	Number	The minimum number of dynamic data VQ pairs (i.e., each pair has one TX and 1 RX queue) that can be set for an virtio-net VF
`qp_pool_size`	Number	Number of free dynamic data VQ pairs (i.e., each pair has one TX and 1 RX queue) available for the VFs on this PF
`num_msix`	Number	Maximum number of MSI-X available for this device
`num_queues`	Number	Maximum virtual queues can be created for this device, driver can choose to create less
`enabled_queues`	Number	Currently enabled number of virtual queues by the driver
`max_queues_size`	Number	Maximum virtual queue depth in byte can be created for each VQ, driver can use less
`msix_config_vector`	String	MSIX vector number used by the driver for the virtio config space. 0xFFFF means that no vector is requested.
`mac`	String	The virtio-net device permanent MAC address, can be only changed from controller side via modify command
`link_status`	Number	Link status of the virtio-net device on the driver side • 0 – down • 1 – up
`max_queue_pairs`	Number	Number of data VQ pairs. One VQ pair has one TX queue and one RX queue. Control or admin VQ are not counted. From the host side, it appears as `Pre-set maximums->Combined` in `ethtool -l <virtio-dev>`.
`mtu`	Number	The virtio-net device MTU. Default is 1500.
`speed`	Number	The virtio-net device link speed in Mb/s

Entry	Type	Description
`rss_max_key_size`	Number	The maximum supported length of the RSS key. Only applicable when `VIRTIO_NET_F_RSS` or `VIRTIO_NET_F_HASH_REPORT` is enabled.
`supported_hash_types`	Number	Supported hash types for this device in hex. Only applicable when `VIRTIO_NET_F_HASH_REPORT` is enabled: • `VIRTIO_NET_HASH_TYPE_IPv4 (bit 0)` • `VIRTIO_NET_HASH_TYPE_TCPv4 (bit 1)` • `VIRTIO_NET_HASH_TYPE_UDPv4 (bit 2)` • `VIRTIO_NET_HASH_TYPE_IPv6 (bit 3)` • `VIRTIO_NET_HASH_TYPE_TCPv6 (bit 4)` • `VIRTIO_NET_HASH_TYPE_UDPv6 (bit 5)`
`ctrl_mac`	String	Admin MAC address configured by driver. Not persistent with driver reload or host reboot.
`ctrl_mq`	Number	Number of queue pairs/channels configured by the driver. From the host side, it appears as `Current hardware settings->Combined` in `ethtool -l <virtio-dev>`.
`sf_num`	Number	Scalable function number used for this virtio-net device
`sf_parent_device`	String	The RDMA device to use to create the SF
`sf_parent_device_pci_addr`	String	The PCIe device address (bus:device:function) to use to create the SF
`sf_rep_net_device`	String	Represents the virtio-net device
`sf_rep_net_ifindex`	Number	The SF representor network interface index
`sf_rdma_device`	String	The SF RDMA device interface name
`sf_cross_mkey`	Number	The cross-device MKEY created for the SF. For debug purposes only.

Entry	Type	Description
sf_vhca_id	Number	Virtual HCA identifier for the SF. For debug purposes only.
rqt_num	Number	The RQ table ID used for this virtio-net device. For debug purposes only.
aarfs	String	Whether Accelerated Receive Flow Steering configuration is enabled or disabled
dim	String	Whether dynamic interrupt moderation (DIM) is enabled or disabled

8.15.5.4.3 Example

The following is an example of a list with 1 static PF created:

```
# virtnet list
{
  "controller": {
    "emulation_manager": "mlx5_0",
    "max_hotplug_devices": "0",
    "max_virt_net_devices": "1",
    "max_virt_queues": "256",
    "max_tunnel_descriptors": "6",
    "supported_features": {
      "value": "0x8b00037700ef982f",
      "    0": "VIRTIO_NET_F_CSUM",
      "    1": "VIRTIO_NET_F_GUEST_CSUM",
      "    2": "VIRTIO_NET_F_CTRL_GUEST_OFFLOADS",
      "    3": "VIRTIO_NET_F_MTU",
      "    5": "VIRTIO_NET_F_MAC",
      "   11": "VIRTIO_NET_F_HOST_TSO4",
      "   12": "VIRTIO_NET_F_HOST_TSO6",
      "   15": "VIRTIO_NET_F_MRG_RXBUF",
      "   16": "VIRTIO_NET_F_STATUS",
      "   17": "VIRTIO_NET_F_CTRL_VQ",
      "   18": "VIRTIO_NET_F_CTRL_RX",
      "   19": "VIRTIO_NET_F_CTRL_VLAN",
      "   21": "VIRTIO_NET_F_GUEST_ANNOUNCE",
      "   22": "VIRTIO_NET_F_MQ",
      "   23": "VIRTIO_NET_F_CTRL_MAC_ADDR",
      "   32": "VIRTIO_F_VERSION_1",
      "   33": "VIRTIO_F_IOMMU_PLATFORM",
      "   34": "VIRTIO_F_RING_PACKED",
      "   36": "VIRTIO_F_ORDER_PLATFORM",
      "   37": "VIRTIO_F_SR_IOV",
      "   38": "VIRTIO_F_NOTIFICATION_DATA",
      "   40": "VIRTIO_F_RING_RESET",
      "   41": "VIRTIO_F_ADMIN_VQ",
      "   56": "VIRTIO_NET_F_HOST_USO",
      "   57": "VIRTIO_NET_F_HASH_REPORT",
      "   59": "VIRTIO_NET_F_GUEST_HDRLEN",
      "   63": "VIRTIO_NET_F_SPEED_DUPLEX"
    },
    "supported_virt_queue_types": {
      "value": "0x1",
      "    0": "SPLIT"
    },
    "supported_event_modes": {
      "value": "0x5",
      "    0": "NO_MSIX_MODE",
      "    2": "MSIX_MODE"
    }
  },
  "devices": [
    {
      "pf_id": 0,
      "function_type": "static PF",
      "transitional": 0,
      "vuid": "MT2306XZ00BNVNETS0D0F2",
      "pci_bdf": "e2:00.2",
```

```
      "pci_vhca_id": "0x2",
      "pci_max_vfs": "0",
      "enabled_vfs": "0",
      "msix_num_pool_size": 0,
      "min_msix_num": 0,
      "max_msix_num": 256,
      "min_num_of_qp": 0,
      "max_num_of_qp": 127,
      "qp_pool_size": 0,
      "num_msix": "256",
      "num_queues": "255",
      "enabled_queues": "0",
      "max_queue_size": "256",
      "msix_config_vector": "0xFFFF",
      "mac": "16:B0:E0:41:B8:0D",
      "link_status": "1",
      "max_queue_pairs": "127",
      "mtu": "1500",
      "speed": "100000",
      "rss_max_key_size": "0",
      "supported_hash_types": "0x0",
      "ctrl_mac": "00:00:00:00:00:00",
      "ctrl_mq": "0",
      "sf_num": 1000,
      "sf_parent_device": "mlx5_0",
      "sf_parent_device_pci_addr": "0000:03:00.0",
      "sf_rep_net_device": "en3f0pf0sf1000",
      "sf_rep_net_ifindex": 10,
      "sf_rdma_device": "mlx5_3",
      "sf_cross_mkey": "0x12642",
      "sf_vhca_id": "0x124",
      "sf_rqt_num": "0x0",
      "aarfs": "disabled",
      "dim": "disabled"
    }
  ]
}
```


The following is an example of a loaded configuration file:


```
# virtnet list -f
{
  "config_file": {
    "ib_dev_for_static_pf": "mlx5_0",
    "ib_dev_p0": "mlx5_0",
    "ib_dev_p1": "mlx5_1",
    "is_lag": 0,
      "static_pf": {
      "mac_base": "02:11:22:33:44:50"
    },
    "sf_provider": "mlxdevm"
  }
}
```


8.15.5.5 Query

This command queries detailed information for a given device, including all VQ information if created.

8.15.5.5.1 Syntax

```
virtnet query [-h] {[-a] | [-p PF] [-v VF] | [-u VUID]} [--dbg_stats] [-b] [--latency_stats] [-q QUEUE_ID] [--
stats_clear] [rx_drops [status|--drops-only]]
```

The options **--pf**, **--vf**, **--vuid**, and **--all** are mutually exclusive (except **--pf** and **--vf** which can be used together), but one of them must be applied.

Option	Abbr	Argument Type	Required	Description
`--help`	`-h`	N/A	No	Show the help message and exit
`--all`	`-a`	N/A	No	Query all the detailed information for all available devices. It can be time consuming if a large number of devices is available.
`--pf`	`-p`	Number	No	Unique device ID for the PF. Can be retrieved by using `virtnet list`.
`--vf`	`-v`	Number	No	Unique device ID for the VF. Can be retrieved by using `virtnet list`.
`--vuid`	`-u`	String	No	Unique device SN for the device (PF/VF). Can be retrieved by using `virtnet list`.
`--queue_id`	`-q`	Number	No	Queue index of the device VQs
`--brief`	`-b`	N/A	No	Query brief information of the device (does not print VQ information)
`--dbg_stats`	N/A	N/A	No	Print debug counters and information ⚠ This option will be depreciated in the future.
`--stats_clear`	N/A	N/A	No	Clear all the debug counter stats ⚠ This option will be depreciated in the future.
`rx_drops`	N/A	N/A	No	Query RX drop counters. Shows total drops across all devices with a list of devices that have non-zero drops. Works in both sync and async modes. ⚠ When used with `-p` and/or `-v`, `rx_drops` shows the total drop count for that specific device only.
`rx_drops status`	N/A	N/A	No	Show current async polling mode configuration.
`rx_drops --drops-only`	N/A	N/A	No	Show devices with drops, including per-RQ breakdown for each device.

8.15.5.5.2 Output

Output has two main sections.

- The first section, wrapped by **devices**, are configuration and capabilities on the device level, the majority of which are the same as the **list** command. This section only covers the differences between the two.

Entry	Type	Description
devices	String	Entries under this section is per-device information
pci_dev_id	String	Virtio-net PCIe device ID. Default: 0x1041. ⚠ This option will be depreciated in the future.
pci_vendor_id	String	Virtio-net PCIe vendor ID. Default: 0x1af4. ⚠ This option will be depreciated in the future.
pci_class_code	String	Virtio-net PCIe device class code. Default: 0x20000. ⚠ This option will be depreciated in the future.
pci_subsys_id	String	Virtio-net PCIe vendor ID. Default: 0x1041. ⚠ This option will be depreciated in the future.
pci_subsys_vendor_id	String	Virtio-net PCIe subsystem vendor ID. Default: 0x1af4. ⚠ This option will be depreciated in the future.
pci_revision_id	String	Virtio-net PCIe revision ID. Default: 1. ⚠ This option will be depreciated in the future.
device_features	String	Enabled device feature bits according to the virtio spec. Refer to section "Feature Bits".
driver_features	String	Enabled driver feature bits according to the virtio spec. Valid only when the driver probes the device. Refer to "Feature Bits".
status	String	Device status field bit masks according to the virtio spec: • **ACKNOWLEDGE (bit 0)** • **DRIVER (bit 1)** • **DRIVER_OK (bit 2)** • **FEATURES_OK (bit 3)** • **DEVICE_NEEDS_RESET (bit 6)** • **FAILED (bit 7)**
reset	Number	Shows if the current virtio-net device undergoing reset: • 0 – not undergoing reset • 1 – undergoing reset
enabled	Number	Shows if the current virtio-net device is enabled: • 0 – disabled, likely FLR has occurred • 1 – enabled

- The second section, wrapped by `enabled-queues-info`, provides per-VQ information:

Entry	Type	Description
`index`	Number	VQ index starting from 0 to `enabled_queues`
`size`	Number	Driver VQ depth in bytes. It is bound by device `max_queues_size`.
`msix_vector`	Number	The MSI-X vector number used for this VQ
`enable`	Number	If current VQ is enabled or not • 0 – disabled • 1 – enabled
`notify_offset`	Number	Driver reads this to calculate the offset from start of notification structure at which this virtqueue is located
`descriptor_address`	Number	The physical address of the descriptor area
`driver_address`	Number	The physical address of the driver area
`device_address`	Number	The physical address of the device area
`received_desc`	Number	Total number of received descriptors by the device on this VQ ⚠ This option will be depreciated in the future.
`completed_desc`	Number	Total number of completed descriptors by the device on this VQ ⚠ This option will be depreciated in the future.
`bad_desc_errors`	Number	Total number of bad descriptors received on this VQ ⚠ This option will be depreciated in the future.
`error_cqes`	Number	Total number of error CQ entries on this VQ ⚠ This option will be depreciated in the future.
`exceed_max_chain`	Number	Total number of chained descriptors received that exceed the maximum allowed chain by device ⚠ This option will be depreciated in the future.
`invalid_buffer`	Number	Total number of times the device tried to read or write buffer that is not registered to the device ⚠ This option will be depreciated in the future.

Entry	Type	Description
`batch_number`	Number	The number of RX descriptors for the last received packet. Relevant for BlueField-3 only. ⚠ This option will be depreciated in the future.
`dma_q_used_number`	Number	The DMA q index used for this VQ. Relevant for BlueField-3 only. ⚠ This option will be depreciated in the future.
`handler_schd_number`	Number	Scheduler number for this VQ. Relevant for BlueField-3 only. ⚠ This option will be depreciated in the future.
`aux_handler_schd_number`	Number	Aux scheduler number for this VQ. Relevant for BlueField-3 only. ⚠ This option will be depreciated in the future.
`max_post_desc_number`	Number	Maximum number of posted descriptors on this VQ. Relevant for DPA. ⚠ This option will be depreciated in the future.
`total_bytes`	Number	Total number of bytes handled by this VQ. Relevant for BlueField-3 only ⚠ This option will be depreciated in the future.
`rq_cq_max_count`	Number	Event generation moderation counter of the queue. Relevant for RQ. ⚠ This option will be depreciated in the future.
`rq_cq_period`	Number	Event generation moderation timer for the queue in 1µsec granularity. Relevant for RQ. ⚠ This option will be depreciated in the future.
`rq_cq_period_mode`	Number	Current period mode for RQ • 0x0 – `default_mode` – use device best defaults • 0x1 – `upon_event` – `queue_period` timer restarts upon event generation • 0x2 – `upon_cqe` – `queue_period` timer restarts upon completion generation ⚠ This option will be depreciated in the future.

8.15.5.5.3 Example

The following is an example of querying the information of the first PF:

```
# virtnet query -p 0
{
  "devices": [
    {
      "pf_id": 0,
      "function_type": "static PF",
      "transitional": 0,
      "vuid": "MT2349X00018VNETS0D0F1",
      "pci_bdf": "23:00.1",
      "pci_vhca_id": "0x1",
      "pci_max_vfs": "0",
      "enabled_vfs": "0",
      "pci_dev_id": "0x1041",
      "pci_vendor_id": "0x1af4",
      "pci_class_code": "0x20000",
      "pci_subsys_id": "0x1041",
      "pci_subsys_vendor_id": "0x1af4",
      "pci_revision_id": "1",
       "device_feature": {
        "value": "0x8930032300ef182f",
        "     0": "VIRTIO_NET_F_CSUM",
        "     1": "VIRTIO_NET_F_GUEST_CSUM",
        "     2": "VIRTIO_NET_F_CTRL_GUEST_OFFLOADS",
        "     3": "VIRTIO_NET_F_MTU",
        "     5": "VIRTIO_NET_F_MAC",
        "    11": "VIRTIO_NET_F_HOST_TSO4",
        "    12": "VIRTIO_NET_F_HOST_TSO6",
        "    16": "VIRTIO_NET_F_STATUS",
        "    17": "VIRTIO_NET_F_CTRL_VQ",
        "    18": "VIRTIO_NET_F_CTRL_RX",
        "    19": "VIRTIO_NET_F_CTRL_VLAN",
        "    21": "VIRTIO_NET_F_GUEST_ANNOUNCE",
        "    22": "VIRTIO_NET_F_MQ",
        "    23": "VIRTIO_NET_F_CTRL_MAC_ADDR",
        "    32": "VIRTIO_F_VERSION_1",
        "    33": "VIRTIO_F_IOMMU_PLATFORM",
        "    37": "VIRTIO_F_SR_IOV",
        "    40": "VIRTIO_F_RING_RESET",
        "    41": "VIRTIO_F_ADMIN_VQ",
        "    52": "VIRTIO_NET_F_VQ_NOTF_COAL",
        "    53": "VIRTIO_NET_F_NOTF_COAL",
        "    56": "VIRTIO_NET_F_HOST_USO",
        "    59": "VIRTIO_NET_F_GUEST_HDRLEN",
        "    63": "VIRTIO_NET_F_SPEED_DUPLEX"
      },
      "driver_feature": {
        "value": "0x8000002300ef182f",
        "     0": "VIRTIO_NET_F_CSUM",
        "     1": "VIRTIO_NET_F_GUEST_CSUM",
        "     2": "VIRTIO_NET_F_CTRL_GUEST_OFFLOADS",
        "     3": "VIRTIO_NET_F_MTU",
        "     5": "VIRTIO_NET_F_MAC",
        "    11": "VIRTIO_NET_F_HOST_TSO4",
        "    12": "VIRTIO_NET_F_HOST_TSO6",
        "    16": "VIRTIO_NET_F_STATUS",
        "    17": "VIRTIO_NET_F_CTRL_VQ",
        "    18": "VIRTIO_NET_F_CTRL_RX",
        "    19": "VIRTIO_NET_F_CTRL_VLAN",
        "    21": "VIRTIO_NET_F_GUEST_ANNOUNCE",
        "    22": "VIRTIO_NET_F_MQ",
        "    23": "VIRTIO_NET_F_CTRL_MAC_ADDR",
        "    32": "VIRTIO_F_VERSION_1",
        "    33": "VIRTIO_F_IOMMU_PLATFORM",
        "    37": "VIRTIO_F_SR_IOV",
        "    63": "VIRTIO_NET_F_SPEED_DUPLEX"
      },
      "status": {
        "value": "0xf",
        "     0": "ACK",
        "     1": "DRIVER",
        "     2": "DRIVER_OK",
        "     3": "FEATURES_OK"
      },
      "reset": "0",
      "enabled": "1",
      "num_msix": "64",
      "num_queues": "63",
      "enabled_queues": "63",
      "max_queue_size": "256",
      "msix_config_vector": "0x0",
      "mac": "4E:6A:E1:41:D8:BE",
      "link_status": "1",
      "max_queue_pairs": "31",
      "mtu": "1500",
      "speed": "200000",
      "rss_max_key_size": "0",
      "supported_hash_types": "0x0",
      "ctrl_mac": "4E:6A:E1:41:D8:BE",
      "ctrl_mq": "31",
      "sf_num": 1000,
      "sf_parent_device": "mlx5_0",
      "sf_parent_device_pci_addr": "0000:03:00.0",
      "sf_rep_net_device": "en3f0pf0sf1000",
      "sf_rep_net_ifindex": 12,
```

```
      "sf_rdma_device": "mlx5_2",
      "sf_cross_mkey": "0xC042",
      "sf_vhca_id": "0x7E8",
      "sf_rqt_num": "0x0",
      "aarfs": "disabled",
      "dim": "disabled",
      "enabled-queues-info": [
        {
          "index": "0",
          "size": "256",
          "msix_vector": "0x1",
          "enable": "1",
          "notify_offset": "0",
          "descriptor_address": "0x10cece000",
          "driver_address": "0x10cecf000",
          "device_address": "0x10cecf240",
          "received_desc": "256",
          "completed_desc": "0",
          "bad_desc_errors": "0",
          "error_cqes": "0",
          "exceed_max_chain": "0",
          "invalid_buffer": "0",
          "batch_number": "64",
          "dma_q_used_number": "6",
          "handler_schd_number": "4",
          "aux_handler_schd_number": "3",
          "max_post_desc_number": "0",
          "total_bytes": "0",
          "rq_cq_max_count": "0",
          "rq_cq_period": "0",
          "rq_cq_period_mode": "1"
        },
        ......
        }
      ]
    }
  ]
}
```


8.15.5.6 Stats

This command is recommended for obtaining all packet counter information. The existing packet counter information available using the **virtnet list** and **virtnet query** commands, but will be deprecated in the future.

This command retrieves the packet counters for a specified device, including detailed information for all Rx and Tx virtqueues (VQs).

To enable/disable byte wise packet counters for each Rx queue, use the following command:

```
virtnet modify {[-p PF] [-v VF]} device -pkt_cnt {enable,disable}
```

- When enabled, byte-wise packet counters are initialized to zero.
- When disabled, the previous values are retained for debugging purposes. The command will still return these old, disabled counter values.

Packet counters are attached to an RQ. Thus, RQ must be created first. This means that the virtio-net device should be probed by the driver on the host OS before running the commands above.

8.15.5.6.1 Syntax

```
virtnet stats [-h] {[-p PF] [-v VF] | [-u VUID]} [-q QUEUE_ID]
```

The options `--pf`, `--vf`, and `--vuid` are mutually exclusive (except `--pf` and `--vf` which can be used together), but one of them must be applied.

Option	Abbr	Argument Type	Required	Description
`--help`	`-h`	N/A	No	Show the help message and exit
`--pf`	`-p`	Number	No	Unique device ID for the PF. Can be retrieved by using `virtnet list`.
`--vf`	`-v`	Number	No	Unique device ID for the VF. Can be retrieved by using `virtnet list`.
`--vuid`	`-u`	String	No	Unique device SN for the device (PF/VF). Can be retrieved by using `virtnet list`.
`--queue_id`	`-q`	Number	No	Queue index of the device RQs or SQs

8.15.5.6.2 Output

The output has two sections.

- The first section wrapped by `device` are device details along with the packet counter statics enable state.

Entry	Type	Description
`device`	String	Entries under this section is per-device information
`pf_id`	String	Physical function ID
`packet_counters`	String	Indicates whether the packet counters feature is enabled or disabled

- The second section wrapped by queues-stats are information for each receive VQ.

Entry	Type	Description
`VQ Index`	Number	The VQ index starts at 0 (the first RQ) and continues up to the last SQ
`rx_64_or_less_octet_packets`	Number	The number of packets received with a size of 0 to 64 bytes. Relevant for BlueField-3 RQ.
`rx_65_to_127_octet_packets`	Number	The number of packets received with a size of 65 to 127 bytes. Relevant for BlueField-3 RQ.
`rx_128_to_255_octet_packets`	Number	The number of packets received with a size of 128 to 255 bytes. Relevant for BlueField-3 RQ.

Entry	Type	Description
rx_256_to_511_octet_packets	Number	The number of packets received with a size of 256 to 511 bytes. Relevant for BlueField-3 RQ.
rx_512_to_1023_octet_packets	Number	The number of packets received with a size of 512 to 1023 bytes. Relevant for BlueField-3 RQ.
rx_1024_to_1522_octet_packets	Number	The number of packets received with a size of 1024 to 1522 bytes. Relevant for BlueField-3 RQ.
rx_1523_to_2047_octet_packets	Number	The number of packets received with a size of 1523 to 2047 bytes. Relevant for BlueField-3 RQ.
rx_2048_to_4095_octet_packets	Number	The number of packets received with a size of 2048 to 4095 bytes. Relevant for BlueField-3 RQ.
rx_4096_to_8191_octet_packets	Number	The number of packets received with a size of 4096 to 8191 bytes. Relevant for BlueField-3 RQ.
rx_8192_to_9022_octet_packets	Number	The number of packets received with a size of 8192 to 9022 bytes. Relevant for BlueField-3 RQ.
received_desc	Number	Total number of received descriptors by the device on this VQ
completed_desc	Number	Total number of completed descriptors by the device on this VQ
bad_desc_errors	Number	Total number of bad descriptors received on this VQ
error_cqes	Number	Total number of error CQ entries on this VQ
exceed_max_chain	Number	Total number of chained descriptors received that exceed the max allowed chain by device
invalid_buffer	Number	Total number of times the device tried to read or write a buffer which is not registered to the device
batch_number	Number	The number of RX descriptors for the last received packet. Relevant for BlueField-3.
dma_q_used_number	Number	The DMA q index used for this VQ. Relevant for BlueField-3.
handler_schd_number	Number	Scheduler number for this VQ. Relevant for BlueField-3.
aux_handler_schd_number	Number	Aux scheduler number for this VQ. Relevant for BlueField-3.
max_post_desc_number	Number	Maximum number of posted descriptors on this VQ. Relevant for DPA.
total_bytes	Number	Total number of bytes handled by this VQ. Relevant for BlueField-3.

Entry	Type	Description
rq_cq_max_count	Number	Event generation moderation counter of the queue. Relevant for RQ.
rq_cq_period	Number	Event generation moderation timer for the queue in 1µsec granularity. Relevant for RQ.
rq_cq_period_mode	Number	Current period mode for RQ • 0x0 – **default_mode** – use device best defaults • 0x1 – **upon_event** – **queue_period** timer restarts upon event generation • 0x2 – **upon_cqe** – **queue_period** timer restarts upon completion generation

8.15.5.6.3 Example

The following is an example of querying the packet statistics information of PF 0 and VQ 0 (i.e., RQ):

```
# virtnet stats -p 0 -q 0
{'pf': '0x0', 'queue_id': '0x0'}
{
  "device": {
    "pf_id": 0,
    "packet_counters": "Enabled",
    "queues-stats": [
      {
        "VQ Index": 0,
        "rx_64_or_less_octet_packets": 0,
        "rx_65_to_127_octet_packets": 259,
        "rx_128_to_255_octet_packets": 0,
        "rx_256_to_511_octet_packets": 0,
        "rx_512_to_1023_octet_packets": 0,
        "rx_1024_to_1522_octet_packets": 0,
        "rx_1523_to_2047_octet_packets": 0,
        "rx_2048_to_4095_octet_packets": 199,
        "rx_4096_to_8191_octet_packets": 0,
        "rx_8192_to_9022_octet_packets": 0,
        "received_desc": "4096",
        "completed_desc": "0",
        "bad_desc_errors": "0",
        "error_cqes": "0",
        "exceed_max_chain": "0",
        "invalid_buffer": "0",
        "batch_number": "64",
        "dma_q_used_number": "0",
        "handler_schd_number": "44",
        "aux_handler_schd_number": "43",
        "max_post_desc_number": "0",
        "total_bytes": "0",
        "err_handler_schd_num": "0",
        "rq_cq_max_count": "0",
        "rq_cq_period": "0",
        "rq_cq_period_mode": "1"
      }
    ]
  }
}
```

8.15.5.7 Modify Device

This command modifies the attributes of a given device.

> When dynamic MSI-X mode is enabled, the user should provision the VF from the DPU side before attaching a VF to the VM.

When dynamic MSI-X mode is disabled, the default number of MSI-X vectors is according to **VIRTIO_NET_EMULATION_NUM_VF_MSIX** value.

8.15.5.7.1 Syntax

The **modify** command supports three subcommands: **device**, **queue**, and **global**.

```
virtnet modify [-h] [-p PF] [-v VF] [-u VUID] [-a] {device,queue,global} ...
```

The options **--pf**, **--vf**, **--vuid**, and **--all** are mutually exclusive (except **--pf** and **--vf** which can be used together), but one of them must be applied.

Option	Abbr	Argument Type	Required	Description
--help	**-h**	N/A	No	Show the help message and exit
--all	**-a**	N/A	No	Modify all available device attributes depending on the selection of **device** or **queue**
--pf	**-p**	Number	No	Unique device ID for the PF. May be retrieved using **virtnet list**.
--vf	**-v**	Number	No	Unique device ID for the VF. May be retrieved using **virtnet list**.
--vuid	**-u**	String	No	Unique device SN for the device (PF/VF). May be retrieved by using **virtnet list**.
device	N/A	Number	No	Modify device specific options
queue	N/A	N/A	No	Modify queue specific options
global	N/A	N/A	No	Modify global controller settings

8.15.5.7.1.1 Device Options

```
virtnet modify device [-h] [-m MAC] [-t MTU] [-e SPEED] [-l LINK]
                      [-s STATE] [-f FEATURES]
                      [-o SUPPORTED_HASH_TYPES] [-k RSS_MAX_KEY_SIZE]
                      [-r RX_MODE] [-n MSIX_NUM] [-q MAX_QUEUE_SIZE]
                      [-b RX_DMA_Q_NUM] [-dc {enable,disable}]
                      [-pkt_cnt {enable,disable}] [-aarfs {enable,disable}]
                      [-qp MAX_QUEUE_PAIRS] [-dim {enable,disable}]
```

Option	Abbr	Argument Type	Required	Description
`--help`	`-h`	String	No	Show the help message and exit
`--mac`	`-m`	Number	No	The virtio-net device MAC address
`--mtu`	`-t`	Number	No	The virtio-net device MTU
`--speed`	`-e`	Number	No	The virtio-net device link speed in Mb/s
`--link`	`-l`	Number	No	The virtio-net device link status • 0 – down • 1 – up
`--state`	`-s`	Number	No	The virtio-net device status field bit masks according to the virtio spec: • `ACKNOWLEDGE (bit 0)` • `DRIVER (bit 1)` • `DRIVER_OK (bit 2)` • `FEATURES_OK (bit 3)` • `DEVICE_NEEDS_RESET (bit 6)` • `FAILED (bit 7)`

`--features`	`-f`	Hex Number / Feature Name / Pattern	No	Configures the `virtio-net` device feature bits in accordance with the virtio specification. Administrators can explicitly set the base feature mask, or dynamically enable and disable specific features using hexadecimal bitmasks or predefined feature names.
				Supported syntax and patterns:
				• Enable features (`+`): Append a plus sign before a bitmask or feature name to enable it.
				  • Examples: `+0x8000` , `+VIRTIO_NET_F_MRG_RXBUF` , or `+MRG_RXBUF`
				• Disable features (`-`): Append a minus sign before a bitmask or feature name to disable it.
				  • Examples: `-0x400000000` , `-VIRTIO_NET_F_MRG_RXBUF` , or `-MRG_RXBUF`
				• Set exact features: Provide a raw bitmask to explicitly overwrite and set the feature bits.
				  • Example: `0x8100000300e7182f`
				• Combined operations: Set a base mask and sequentially append `+` or `-` operations to modify specific bits in a single, complex command.
				  • Format: `<bitmask>?([+-]<bitmask|name>)*`
				  • Example: `0x8100000300e7182f+0x400000000-MRG_RXBUF`

Option	Abbr	Argument Type	Required	Description
`--supported_hash_types`	`-o`	Hex Number	No	Supported hash types for this device in hex. Only applicable when `VIRTIO_NET_F_HASH_REPORT` is enabled. • `VIRTIO_NET_HASH_TYPE_IPv4 (bit 0)` • `VIRTIO_NET_HASH_TYPE_TCPv4 (bit 1)` • `VIRTIO_NET_HASH_TYPE_UDPv4 (bit 2)` • `VIRTIO_NET_HASH_TYPE_IPv6 (bit 3)` • `VIRTIO_NET_HASH_TYPE_TCPv6 (bit 4)` • `VIRTIO_NET_HASH_TYPE_UDPv6 (bit 5)`
`--rss_max_key_size`	`-k`	Number	No	The maximum supported length of RSS key. Only applicable when `VIRTIO_NET_F_RSS` or `VIRTIO_NET_F_HASH_REPORT` is enabled.
`--rx_mode`	`-r`	Hex Number	No	The RX mode exposed to the driver: • 0 – promisc • 1 – all-multi • 2 – all-uni • 3 – no-multi • 4 – no-uni • 5 – no-broadcast
`--msix_num`	`-n`	Number	No	Maximum number of VQs (both data and ctrl/admin VQ). It is bound by the cap of `max_virt_queues` at the controller level (`virtnet list`).
`--max_queue_size`	`-q`	Number	No	Maximum number of buffers in the VQ. The queue size value is always a power of 2. The maximum queue size value is 32768.
`--max_queue_pairs`	`-qp`	Number	No	Number of data VQ pairs. One VQ pair has one TX queue and one RX queue. Control or admin VQs are not counted. From the host side, it appears as `Pre-set maximums->Combined` in `ethtool -l <virtio-dev>` .
`--rx_dma_q_num`	`-b`	Number	No	Modify max RX DMA queue number
`--drop_counter`	`-dc`	String	No	Enable/disable virtio-net drop counter
`--packet_counter`	`-pkt_cnt`	String	No	Enable/disable virtio-net device packet counter stats
`--aarfs_config`	`-aarfs`	String	No	Enable/disable auto-AARFS. Only applicable for PF devices (static PF and hotplug PF).

Option	Abbr	Argument Type	Required	Description
`--dim_config`	`-dim`	String	No	Enable/disable dynamic interrupt moderation (DIM)

The following `modify` options require unbinding the virtio device from virtio-net driver in the guest OS:

- `mac`
- `mtu`
- `features`
- `msix_num`
- `max_queue_size`
- `max_queue_pairs`

For example:

1. On the guest OS:

```
[host]# echo "bdf of virtio-dev" > /sys/bus/pci/drivers/virtio-pci/unbind
```

2. On the DPU side:
   a. Modify the max queue size of device:

```
[dpu]# virtnet modify -p 0 -v 0 device -q 2048
```

   b. Modify the MSI-X number of VF device:

```
[dpu]# virtnet modify -p 0 -v 0 device -n 8
```

   c. Modify the MAC address of virtio physical device ID 0 (or with its "VUID string", which can be obtained through virtnet list/query):

```
[dpu]# virtnet modify -p 0 device -m 0C:C4:7A:FF:22:93
```

   d. Modify the maximum number of queue pairs of VF device:

```
[dpu]# virtnet modify -p 0 -v 0 device -qp 2
```

3. On the guest OS:

```
[host]# echo "bdf of virtio-dev" > /sys/bus/pci/drivers/virtio-pci/bind
```

Enabling and Disabling Virtio-net Features

Configuring virtio-net features involves verifying what the DPU supports, modifying the device configuration, and confirming that the host driver successfully negotiated the new features.

Verify Supported Features

Check the full list of features supported by the underlying controller using the **virtnet list** command.

```
virtnet list
```

Example JSON output:

```
"supported_features": {
    "value":         "0x8b00037700ef982f",
    "    0":         "VIRTIO_NET_F_CSUM",
    "    1":         "VIRTIO_NET_F_GUEST_CSUM",
    "    2":         "VIRTIO_NET_F_CTRL_GUEST_OFFLOADS",
    "    3":         "VIRTIO_NET_F_MTU",
    "    5":         "VIRTIO_NET_F_MAC",
    "   11":         "VIRTIO_NET_F_HOST_TSO4",
    "   12":         "VIRTIO_NET_F_HOST_TSO6",
    "   15":         "VIRTIO_NET_F_MRG_RXBUF",
    "   34":         "VIRTIO_F_RING_PACKED",
    "   63":         "VIRTIO_NET_F_SPEED_DUPLEX"
}
```

Important Constraints

- Feature names and bit numbers strictly follow the virtio-net specification.
- The specific list of supported features will vary depending on the device type, driver capabilities, and current application version.

Check Currently Enabled Device Features

Verify which features are currently enabled on your target device (e.g., Physical Function **0**) by inspecting the **device_feature** section of the query output.

```
virtnet query -p 0 -b
```

Example JSON output:

example, features enabled for PF 0

```
"pci_bdf": "0f:00.2",
"device_feature": {
    "value":         "0x8100032300ef982f",
    "    0":         "VIRTIO_NET_F_CSUM",
    "   32":         "VIRTIO_F_VERSION_1",
    "   33":         "VIRTIO_F_IOMMU_PLATFORM",
    "   63":         "VIRTIO_NET_F_SPEED_DUPLEX"
}
```

Modify Device Features

Driver Unbonding Required

You cannot modify device features while the device driver is actively bound. You must unbind the device driver before executing any **virtnet modify device** commands. Refer to the "Modify Device" section for exact unbinding steps.

Once unbound, use the **-f** flag to enable, disable, or explicitly set the feature bitmask.

- To enable a feature (**+**):

```
# Syntax: virtnet modify -p <pf> device -f +<feature_name>
virtnet modify -p 0 device -f +VIRTIO_F_RING_PACKED
```

- To disable a feature (**-**):

```
# Syntax: virtnet modify -p <pf> device -f -<feature_name>
virtnet modify -p 0 device -f -VIRTIO_F_RING_PACKED
```

- To set an explicit feature vector (bitmask) – You can overwrite the entire feature vector using a hex mask. For example, to add **VIRTIO_F_RING_PACKED** (bit 34) to a base vector of **0x8100032300EF182F** , you calculate the logical OR (**0x8100032300EF182F | 0x400000000 = 0x8100032700EF182F**) and apply it:

```
# Syntax: virtnet modify -p <pf> device -f <features_bitmask>
virtnet modify -p 0 device -f 0x8100032700EF182F
```

Verify Driver Negotiation

After modifying the features and rebonding the driver, you must verify that the host operating system successfully negotiated the new features.

A feature is only fully active if it appears in both the **device_feature** and **driver_feature** lists.

```
virtnet query -p 0 -b
```

Example output confirming **VIRTIO_F_RING_PACKED** (bit 34) was successfully negotiated:

```
"pci_bdf": "0f:00.2",

"device_feature": {
    "    34":         "VIRTIO_F_RING_PACKED"
},
"driver_feature": {
    "    34":         "VIRTIO_F_RING_PACKED"
}
```

8.15.5.7.1.2 Queue Options

```
virtnet modify queue [-h] -e {event,cqe} -n PERIOD -c MAX_COUNT
```

Option	Abbr	Argument Type	Required	Description
--help	**-h**	String	No	Show the help message and exit

Option	Abbr	Argument Type	Required	Description
`--period_mode`	`-e`	String	No	RQ period mode: `event` or `cqe`. Default is selected by device for the best result.
`--period`	`-n`	Number	No	The event generation moderation timer for the queue in 1µsec granularity
`--max_count`	`-c`	Number	No	The max event generation moderation counter of the queue

8.15.5.7.1.3 Global Options

```
virtnet modify global dc -mode <async|sync> --poll_freq <seconds>
```

Option	Abbr	Argument Type	Required	Description
`--help`	`-h`	String	No	Show the help message and exit
`--mode=<async|sync>`	N/A	String	Yes	Set the drop counter query mode. `async` enables background polling with cached queries. `sync` (default) uses real-time hardware queries.
`--poll_freq`	N/A	Number	No	Polling interval in seconds when using `mode=async`. Valid range: 1–600. Default: 5.

Global output:

Entry	Type	Description
`mode`	String	Current drop counter query mode: `async` or `sync`
`poll_freq_sec`	Number	Polling interval in seconds (only shown when `dc mode` is `async`)

Global examples:

- Enable async mode with default polling interval:

```
[dpu]# virtnet modify global dc -mode=async
{
    "dc_mode": "async",
    "poll_freq_sec": 5
}
```

- Enable async mode with custom polling interval:

```
[dpu]# virtnet modify global dc -mode=async -poll_freq 10
{
    "dc_mode": "async",
    "poll_freq_sec": 10
```

```
}
```

- Disable async mode (return to sync):

```
[dpu]# virtnet modify global dc -mode=sync
{
    "dc_mode": "sync"
}
```

> ⚠ The async mode setting is persisted across controller restarts. No manual action is needed to restore it after a restart.

8.15.5.7.2 Output

Entry	Type	Description
`errno`	Number	Error number: • 0 – success • Non-0 – failed
`errstr`	String	Explanation of the error number

8.15.5.7.3 Example

To modify the link status of the first VF on the first PF to be down:

```
# virtnet modify -p 0 device -l 0
{'pf': '0x0', 'all': '0x0', 'subcmd': '0x0', 'link': '0x0'}
{
  "errno": 0,
  "errstr": "Success"
}
```

8.15.5.8 Log

This command manages the log level of virtio-net-controller.

8.15.5.8.1 Syntax

```
virtnet log [-h] -l {info,err,debug}
```

Option	Abbr	Argument Type	Required	Description
`--help`	`-h`	N/A	No	Show the help message and exit
`--level`	`-l`	String	Yes	Change the log level of `virtio_net_controller` from the journal. Default is DEBUG.

8.15.5.8.2 Output

Entry	Type	Description
Stdout	String	Success or failed with message

8.15.5.8.3 Example

To change the log level to info:

```
# virtnet log -l info
{'level': 'info'}
"Success"
```

To monitor current log output of the controller service with the latest 100 lines printed out:

```
$ journalctl -u virtio-net-controller -f -n 100
```

8.15.5.9 Validate

This command validates configurations of virtio-net-controller.

8.15.5.9.1 Syntax

```
virtnet validate [-h] -f PATH_TO_FILE
```

Option	Abbr	Argument Type	Required	Description
--help	**-h**	N/A	No	Show the help message and exit
--file	**-f**	String	No	Validate the JSON format of the **virtnet.conf** file of the **virtio_net_controller**

8.15.5.9.2 Output

Entry	Type	Description
Stdout	String	Success or failed with message

8.15.5.9.3 Example

To check if **virtnet.conf** is a valid JSON file:

```
# virtnet validate -f /opt/mellanox/mlnx_virtnet/virtnet.conf
```

```
/opt/mellanox/mlnx_virtnet/virtnet.conf is valid
```

8.15.5.10 Version

This command prints current and updated version of virtio-net-controller.

8.15.5.10.1 Syntax

```
virtnet version [-h]
```

Option	Abbr	Argument Type	Required	Description
--help	**-h**	N/A	No	Show the help message and exit

8.15.5.10.2 Output

Entry	Type	Description
Original Controller	String	The original controller version
Destination Controller	String	The to be updated controller version

8.15.5.10.3 Example

Check current and next available controller version:

```
# virtnet version
[
  {
    "Original Controller": "v24.10.17"
  },
  {
    "Destination Controller": "v24.10.19"
  }
]
```

8.15.5.11 Update

This command performs a live update to another version installed on the OS. Instead of a complete shutdown and recreating all existing devices, this procedure updates to the new version with minimal down time.

8.15.5.11.1 Syntax

```
virtnet update [-h] [-s [-f] | -t]
```

Option	Abbr	Argument Type	Required	Description
--help	**-h**	N/A	No	Show the help message and exit
--start	**-s**	N/A	No	Start a coordinated live update and wait for in-flight RPC/CLI commands to finish

Option	Abbr	Argument Type	Required	Description
`--force`	`-f`	N/A	No	Start the live update to start immediately (not recommended)
`--status`	`-t`	N/A	No	Check live update status

8.15.5.11.2 Output

Entry	Type	Description
`stdout`	String	If the update started successfully

8.15.5.11.3 Example

To start a safe live update process after in-flight CLI commands have finished, run:

```
# virtnet update -s
{'start': '0x1'}
"Update started, use 'virtnet update -t' or check logs for status"
```

To start the live update process immediately, run:

```
# virtnet update -s -f
{'start': '0x1'}
"Update started, use 'virtnet update -t' or check logs for status"
```

> ⚠ `-f` performs a forced live update without waiting for in-flight CLI commands to finish, which may interrupt ongoing operations and leave the system in an inconsistent state.

To check the update status during the update process:

```
# virtnet update -t
{'status': '0x1'}
{
  "status": "inactive",
  "last live update status": "success",
  "time_used (s)": 0.604152
}
```

8.15.5.12 Restart

This command performs a fast restart of the virtio-net-controller service. Compared to regular restart (using `systemctl restart virtio-net-controller`) this command has shorter down time per device.

8.15.5.12.1 Syntax

```
virtnet restart [-h]
```

Option	Abbr	Argument Type	Required	Description
`--help`	`-h`	N/A	No	Show the help message and exit

8.15.5.12.2 Output

Entry	Type	Description
`stdout`	String	If the fast restart finishes successfully • SUCCESS • Failed to fast restart

8.15.5.12.3 Example

To start the live update process, run:

```
# virtnet restart
SUCCESS
```

8.15.5.13 Health

This command shows health information for given devices.

> ⚠ The virtio-net driver must be loaded for this command to show valid information.

8.15.5.13.1 Syntax

```
virtnet health [-h] {[-a] | [-p PF] [-v VF] | [-u VUID]} [show]
```

> ℹ The options `--pf`, `--vf`, `--vuid`, and `--all` are mutually exclusive (except `--pf` and `--vf` which can be used together), but one of them must be applied.

Option	Abbr	Argument Type	Required	Description
`--help`	`-h`	N/A	No	Show the help message and exit
`--all`	`-a`	N/A	No	Query all the detailed information for all available devices. It can be time consuming if a large number of devices is available.
`--pf`	`-p`	Number	No	Unique device ID for the PF. Can be retrieved by using `virtnet list`.

Option	Abbr	Argument Type	Required	Description
`--vf`	`-v`	Number	No	Unique device ID for the VF. Can be retrieved by using `virtnet list`.
`--vuid`	`-u`	String	No	Unique device SN for the device (PF/VF). Can be retrieved by using `virtnet list`.

Sub-command	Required	Description
`show`	Yes	Show health information for given devices

8.15.5.13.2 Output

Entry	Type	Description
`pf_id`	Number	Physical function ID
`type`	String	Function type: Static PF, hotplug PF, VF
`vuid`	String	Unique device SN, it can be used as an index to query/modify/unplug a device
`dev_status`	String	Device status field bit masks according to the virtio spec: • `ACKNOWLEDGE (bit 0)` • `DRIVER (bit 1)` • `DRIVER_OK (bit 2)` • `FEATURES_OK (bit 3)` • `DEVICE_NEEDS_RESET (bit 6)` • `FAILED (bit 7)`
`health_status`	String	• Good • Fatal
`health_recover_counter`	Number	The number of recoveries has been performed

Entry	Type	Description
dev_health_details	Dictionary	Two types of health information are included: **control_plane_errors** and **data_plane_errors** , where **control_plane_errors** has following specific errors reported, with value either 0 or 1: • **sf_rqt_update_err** • **sf_drop_create_err** • **sf_tir_create_err** • **steer_rx_domain_err** • **steer_rx_table_err** • **sf_flows_apply_err** • **aarfs_flow_init_err** • **vlan_flow_init_err** • **drop_cnt_config_err** and **data_plane_errors** has following specific errors reported, with value either 0 or 1: • **sq_stall** • **dma_q_stall** • **spurious_db_invoke** • **aux_not_invoked** • **dma_q_errors** • **host_read_errors** Detailed descriptions of each error can be found in Health Statistics.

8.15.5.13.3 Example

The following is an example of showing the information of the first PF:

```
# virtnet health -p 0 show
{'pf': '0x0', 'all': '0x0', 'subcmd': '0x0'}
{
  "pf_id": 0,
  "type": "static PF",
  "vuid": "MT2306XZ00BPVNETS0D0F1",
  "dev_status": {
    "value": "0xf",
    "    0": "ACK",
    "    1": "DRIVER",
    "    2": "DRIVER_OK",
    "    3": "FEATURES_OK"
  },
  "health_status": "Good",
  "health_recover_counter": 0,
  "dev_health_details": {
    "control_plane_errors": {
      "sf_rqt_update_err": 0,
      "sf_drop_create_err": 0,
      "sf_tir_create_err": 0,
      "steer_rx_domain_err": 0,
      "steer_rx_table_err": 0,
      "sf_flows_apply_err": 0,
      "aarfs_flow_init_err": 0,
      "vlan_flow_init_err": 0,
      "drop_cnt_config_err": 0
    },
    "data_plane_errors": {
      "sq_stall": 0,
      "dma_q_stall": 0,
      "spurious_db_invoke": 0,
      "aux_not_invoked": 0,
      "dma_q_errors": 0,
      "host_read_errors": 0
    }
  }
}
```

8.15.5.14 Error Code

CLI commands will return non-zero error code upon failure. All error numbers are negative. When an error occurs from the log, it could return an error number as well.

If the error number is greater than `-1000`, it is standard error. Please refer to Linux error code at errno

If the error number is less or equal `-1000`, please refer to the table below for the explanation.

Errno	Error Name	Error Description
`-1000`	`VIRTNET_ERR_DEV_FEATURE_VALIDATE`	Failed to validate device feature
`-1001`	`VIRTNET_ERR_DEV_NOT_FOUND`	Failed to find device
`-1002`	`VIRTNET_ERR_DEV_NOT_PLUGGED`	Failed - Device is not hotplugged
`-1003`	`VIRTNET_ERR_DEV_NOT_STARTED`	Failed - Device did not start
`-1004`	`VIRTNET_ERR_DRIVER_PROBED`	Failed - Virtio driver should not be loaded
`-1005`	`VIRTNET_ERR_EPOLL_ADD`	Failed to add epoll
`-1006`	`VIRTNET_ERR_ID_OUT_OF_RANGE`	Failed - ID input exceeds the max range
`-1007`	`VIRTNET_ERR_VUID_INVALID`	Failed - VUID is invalid
`-1008`	`VIRTNET_ERR_MAC_INVALID`	Failed - MAC is invalid
`-1009`	`VIRTNET_ERR_MSIX_INVALID`	Failed - MSIX is invalid
`-1010`	`VIRTNET_ERR_MTU_INVALID`	Failed - MTU is invalid
`-1011`	`VIRTNET_ERR_PORT_CONTEXT_NOT_FOUND`	Failed to find port context
`-1012`	`VIRTNET_ERR_REC_CONFIG_LOAD`	Failed to load config from recovery file
`-1013`	`VIRTNET_ERR_REC_CONFIG_SAVE`	Failed to save config into recovery file

Errno	Error Name	Error Description
-1014	VIRTNET_ERR_REC_FILE_CREATE	Failed to create recovery file
-1015	VIRTNET_ERR_REC_MAC_DEL	Failed to delete MAC in recovery file
-1016	VIRTNET_ERR_REC_MAC_LOAD	Failed to load MAC from recovery file
-1017	VIRTNET_ERR_REC_MAC_SAVE	Failed to save MAC into recovery file
-1018	VIRTNET_ERR_REC_MQ_SAVE	Failed to save MQ into recovery file
-1019	VIRTNET_ERR_REC_PFNUM_LOAD	Failed to load PF number from recovery file
-1020	VIRTNET_ERR_REC_RX_MODE_SAVE	Failed to save RX mode into recovery file
-1021	VIRTNET_ERR_REC_SF_SAVE	Failed to save PF and SF number into recovery file
-1022	VIRTNET_ERR_REC_SFNUM_LOAD	Failed to load SF number from recovery file
-1023	VIRTNET_ERR_SF_MAC_FLOW_APPLY	Failed to apply MAC flow by SF
-1024	VIRTNET_ERR_SF_MQ_UPDATE	Failed to update MQ by SF
-1025	VIRTNET_ERR_SF_RX_MODE_SET	Failed to set RX mode by SF
-1026	VIRTNET_ERR_SNAP_NET_CTRL_OPEN	Failed to open SNAP device control
-1027	VIRTNET_ERR_SNAP_CROSS_MKEY_CREATE	Failed to create SNAP cross mkey
-1028	VIRTNET_ERR_SNAP_DMA_Q_CREATE	Failed to create SNAP DMA Q
-1029	VIRTNET_ERR_SNAP_NET_DEV_QUERY	Failed to query SNAP device
-1030	VIRTNET_ERR_SNAP_NET_DEV_MODIFY	Failed to modify SNAP device
-1031	VIRTNET_ERR_SNAP_PF_HOTPLUG	Failed to hotplug SNAP PF

Errno	Error Name	Error Description
-1032	VIRTNET_ERR_VQ_PERIOD_UPDATE	Failed to update VQ period
-1033	VIRTNET_ERR_QUEUE_SIZE_INVALID	Failed - Queue size is invalid
-1034	VIRTNET_ERR_SF_PORT_ADD	Failed to add SF port
-1035	VIRTNET_ERR_WQ_WORKQUEUE_ALLOC	Failed to alloc workqueue
-1036	VIRTNET_ERR_ETH_VQS_OPERATION_ALLOC	Failed to alloc eth VQS operation
-1037	VIRTNET_ERR_ETH_VQS_OPERATION_COMP	Failed to complete eth VQS operation
-1038	VIRTNET_ERR_JSON_OBJ_NOT_EXIST	Failed - JSON obj does not exist
-1039	VIRTNET_ERR_DEV_LOAD_PREP	Failed to prepare device load
-1040	VIRTNET_ERR_DEV_SW_MIGRATION	Failed to sw migrate a device
-1041	VIRTNET_ERR_DEV_IS_SW_MIGRATING	Failed - Device is migrating
-1042	VIRTNET_ERR_MAX_QUEUE_SIZE	Error - queue size must be greater than 2 and is power of 2
-1043	VIRTNET_ERR_MSIX_LESS_EQUAL_THRE	Warning - this device won't function, don't try to probe with virtio driver
-1044	VIRTNET_ERR_SF_POOL_CREATING	SF pool is creating try again later
-1046	VIRTNET_ERR_INVALID_OPTION	Option is not supported
-1047	VIRTNET_ERR_SF_CREATE	Failed to create SF
-1048	VIRTNET_ERR_DEV_SF_NUM_OUT_OF_RANGE	SF number for hotplug device should be between 2000 and 2999
-1049	VIRTNET_ERR_DEV_SF_NUM_USED	SF number is already used
-1050	VIRTNET_ERR_QUEUE_NUMBER_INVALID	Queue index is invalid

Errno	Error Name	Error Description
-1051	VIRTNET_ERR_SPEED_INVALID	Invalid speed please check help menu for supported link speeds
-1052	VIRTNET_ERR_SUPPORTED_HASH_TYPES_INVALID	Invalid hash types please check help menu for supported hash types
-1053	VIRTNET_ERR_RSS_MAX_KEY_SIZE_INVALID	Invalid rss max key size supported key size is 40
-1054	VIRTNET_ERR_REC_OFFLOADS_SAVE	Failed to save OFFLOADS into recovery file
-1055	VIRTNET_ERR_SF_OFFLOADS_UPDATE	Failed to update OFFLOADS by SF
-1056	VIRTNET_ERR_READ_LINK	Failed to readlink
-1057	VIRTNET_ERR_PATH_FORMAT	Error - Path format is invalid
-1058	VIRTNET_ERR_Q_COUNTER_ALLOC	Failed to alloc q counter
-1059	VIRTNET_ERR_REC_DIRTY_LOG_SAVE	Failed to save dirty log
-1060	VIRTNET_ERR_REC_DIRTY_LOG_DEL	Failed to delete dirty log
-1061	VIRTNET_ERR_REC_LM_STATUS_SAVE	Failed to save LM status
-1062	VIRTNET_ERR_REC_LM_STATUS_REC	Failed to found LM status record
-1063	VIRTNET_ERR_REC_DEV_MODE_SAVE	Failed to save dev mode
-1064	VIRTNET_ERR_REC_DEV_MODE_REC	Failed to found dev mode record
-1065	VIRTNET_ERR_UNPLUG_NOT_READY	Error - Device is not ready to be unplugged please check host and retry
-1066	VIRTNET_ERR_REC_MAC_TABLE_DEL	Failed to delete MAC table in recovery file
-1067	VIRTNET_ERR_REC_MAC_TABLE_LOAD	Failed to load MAC table from recovery file
-1068	VIRTNET_ERR_REC_MAC_TABLE_SAVE	Failed to save MAC table into recovery file

Errno	Error Name	Error Description
-1069	VIRTNET_ERR_REC_HASH_CFG_DEL	Failed to delete hash cfg in recovery file
-1070	VIRTNET_ERR_REC_HASH_CFG_LOAD	Failed to load hash cfg from recovery file
-1071	VIRTNET_ERR_REC_HASH_CFG_SAVE	Failed to save hash cfg into recovery file
-1072	VIRTNET_ERR_DEV_VF_GET	Failed to get VF device
-1073	VIRTNET_ERR_MAX_QUEUES_INVALID	Failed - QUEUES is invalid
-1074	VIRTNET_ERR_DEBUGFS_SAVE	Failed to save into debugfs file
-1075	VIRTNET_ERR_DEBUGFS_DEL	Failed to delete from debugfs file

8.15.5.15 Debug

The **virtnet_cli** tool provides debug commands for the event publisher.

- To view publisher configuration:

```
virtnet debug vnet_event config
```

- To view publisher counters:

```
virtnet debug vnet_event stats
```

Counter	Meaning
enqueued	Total events successfully enqueued to the publish ring buffer.
dropped_queue_full	Events dropped because the queue was at **max_queue_depth**.
json_encode_fail	Events that failed JSON serialization (should always be 0).
transport_publish_fail	Events that the worker thread failed to publish to NATS.
reconnect_attempts	Number of times the worker attempted to reconnect to the broker.
last_error	Last negative errno from a failed operation (0 = no error).

- To enable verbose debug logging (runtime):

```
virtnet debug vnet_event --log_level 1
```

This enables per-event syslog traces showing the NATS subject, JSON preview, and publish outcome. Set back to `0` to disable.

8.15.6 Feature Guidance

8.15.6.1 Counters

8.15.6.1.1 Packet Statistics

To query the packet counters, use `stats` command.

```
[dpu]# virtnet stats [-h] {[-p PF] [-v VF] | [-u VUID]} [-q QUEUE_ID]
```

The options `--pf`, `--vf` and `--vuid` are mutually exclusive, but one of them must be applied.

Option	Abbr	Argument Type	Required	Description
`--help`	`-h`	N/A	No	Show the help message and exit
`--pf`	`-p`	Number	No	Unique device ID for the PF. Can be retrieved by using `virtnet list`.
`--vf`	`-v`	Number	No	Unique device ID for the VF. Can be retrieved by using `virtnet list`.
`--vuid`	`-u`	String	No	Unique device SN for the device (PF/VF). Can be retrieved by using `virtnet list`.
`--queue_id`	`-q`	Number	No	Queue index of the device RQs or SQs

This command is recommended for obtaining all packet counter information. The existing packet counter information available through the `virtnet list` and `virtnet query` commands will be deprecated in the future.

The following command queries PF 0 and VQ 0 (i.e., RQ):

```
[dpu]# virtnet stats -p 0 -q 0
```

Output:

```
# virtnet stats -p 0 -q 0
{'pf': '0x0', 'queue_id': '0x0'}
{
  "device": {
    "pf_id": 0,
    "packet_counters": "Enabled",
    "queues-stats": [
      {
```

```
        "VQ Index": 0,
        "rx_64_or_less_octet_packets": 0,
        "rx_65_to_127_octet_packets": 259,
        "rx_128_to_255_octet_packets": 0,
        "rx_256_to_511_octet_packets": 0,
        "rx_512_to_1023_octet_packets": 0,
        "rx_1024_to_1522_octet_packets": 0,
        "rx_1523_to_2047_octet_packets": 0,
        "rx_2048_to_4095_octet_packets": 199,
        "rx_4096_to_8191_octet_packets": 0,
        "rx_8192_to_9022_octet_packets": 0,
        "received_desc": "4096",
        "completed_desc": "0",
        "bad_desc_errors": "0",
        "error_cqes": "0",
        "exceed_max_chain": "0",
        "invalid_buffer": "0",
        "batch_number": "64",
        "dma_q_used_number": "0",
        "handler_schd_number": "44",
        "aux_handler_schd_number": "43",
        "max_post_desc_number": "0",
        "total_bytes": "0",
        "err_handler_schd_num": "0",
        "rq_cq_max_count": "0",
        "rq_cq_period": "0",
        "rq_cq_period_mode": "1"
      }
    ]
  }
}
```


The output has two sections.

- The first section, wrapped by **device**, are device details along with the packet counter statics enable state.

Entry	Type	Description
device	String	Entries under this section is per device information
pf_id	String	Physical function ID
packet_counters	String	packet counters feature: enabled/disabled

- The second section, wrapped by **queues-stats**, are information for each receive VQ.

Entry	Type	Description
VQ Index	Number	The VQ index starts at 0 (the first RQ) and continues up to the last SQ
rx_64_or_less_octet_packets	Number	The number of packets received with a size of 0 to 64 bytes. Relevant for BlueField-3 RQ when packet counter is enabled.
rx_65_to_127_octet_packets	Number	The number of packets received with a size of 65 to 127 bytes. Relevant for BlueField-3 RQ when packet counter is enabled.
rx_128_to_255_octet_packets	Number	The number of packets received with a size of 128 to 255 bytes. Relevant for BlueField-3 RQ when packet counter is enabled.
rx_256_to_511_octet_packets	Number	The number of packets received with a size of 256 to 511 bytes. Relevant for BlueField-3 RQ when packet counter is enabled.
rx_512_to_1023_octet_packets	Number	The number of packets received with a size of 512 to 1023 bytes. Relevant for BlueField-3 RQ when packet counter is enabled.

Entry	Type	Description
rx_1024_to_1522_octet_packets	Number	The number of packets received with a size of 1024 to 1522 bytes. Relevant for BlueField-3 RQ when packet counter is enabled.
rx_1523_to_2047_octet_packets	Number	The number of packets received with a size of 1523 to 2047 bytes. Relevant for BlueField-3 RQ when packet counter is enabled.
rx_2048_to_4095_octet_packets	Number	The number of packets received with a size of 2048 to 4095 bytes. Relevant for BlueField-3 RQ when packet counter is enabled.
rx_4096_to_8191_octet_packets	Number	The number of packets received with a size of 4096 to 8191 bytes. Relevant for BlueField-3 RQ when packet counter is enabled.
rx_8192_to_9022_octet_packets	Number	The number of packets received with a size of 8192 to 9022 bytes. Relevant for BlueField-3 RQ when packet counter is enabled.
received_desc	Number	Total number of received descriptors by the device on this VQ
completed_desc	Number	Total number of completed descriptors by the device on this VQ
bad_desc_errors	Number	Total number of bad descriptors received on this VQ
error_cqes	Number	Total number of errors CQ entries on this VQ
exceed_max_chain	Number	Total number of chained descriptors received that exceed the max allowed chain by the device
invalid_buffer	Number	Total number of times device tried to read or write buffer that is not registered to the device
batch_number	Number	The number of RX descriptors for the last received packet. Relevant for BlueField-3.
dma_q_used_number	Number	The DMA q index used for this VQ. Relevant for BlueField-3.
handler_schd_number	Number	Scheduler number for this VQ. Relevant for BlueField-3.
aux_handler_schd_number	Number	Aux scheduler number for this VQ. Relevant for BlueField-3.
max_post_desc_number	Number	Maximum number of posted descriptors on this VQ. Relevant for DPA.
total_bytes	Number	Total number of bytes handled by this VQ. Relevant for BlueField-3.
rq_cq_max_count	Number	Event generation moderation counter of the queue. Relevant for RQ.
rq_cq_period	Number	Event generation moderation timer for the queue in 1μsec granularity. Relevant for RQ.

Entry	Type	Description
rq_cq_period_mode	Number	Current period mode for RQ • 0x0 – **default_mode** – use device best defaults • 0x1 – **upon_event** – **queue_period** timer restarts upon event generation • 0x2 – **upon_cqe** – **queue_period** timer restarts upon completion generation The second section wrapped by queues-stats IS information for each receive VQ.

8.15.6.1.1.1 VQ Statistics

To query Rx VQ statistics, use the corresponding VQ index. For example, If there are 3 queues configured then to query Rx, VQ uses queue 0, Tx VQ uses queue 1, and Ctrl VQ uses queue 2.

The following is the command to query PF 0, VF 0, and VQ 0 (i.e., Rx).

```
[dpu]# virtnet query -p 0 -v 0 -q 0
```

Output:

```
"enabled-queues-info": [
  {
    "index": "0",
    "size": "256",
    "msix_vector": "0x1",
    "enable": "1",
    "notify_offset": "0",
    "descriptor_address": "0xffffe000",
    "driver_address": "0xfffff000",
    "device_address": "0xfffff240",
    "received_desc": "256",
    "completed_desc": "19",
    "bad_desc_errors": "0",
    "error_cqes": "0",
    "exceed_max_chain": "0",
    "invalid_buffer": "0",
    "batch_number": "64",
    "dma_q_used_number": "0",
    "handler_schd_number": "4",
    "aux_handler_schd_number": "3",
    "max_post_desc_number": "0",
    "total_bytes": "6460",
    "rq_cq_max_count": "0",
    "rq_cq_period": "0",
    "rq_cq_period_mode": "1"
  }
```

The following are some of the important VQ counters:

Counter Name	Description
total_bytes	Number of bytes received
received_desc	Number of available descriptors received by device
completed_desc	Number of available descriptors completed by the device
error_cqes	Number of error CQEs received on the queue
bad_desc_errors	Number of bad descriptors received
exceed_max_chain	Number of chained descriptors received that exceed the max allowed chain by device

Counter Name	Description
invalid_buffer	Number of times device tried to read or write buffer that is not registered to the device

8.15.6.1.2 RQ Drop Counter

When DPA is the data path provider, each RQ has its corresponding drop counter, which counts the number of packets dropped inside the DPA virtio RQs.

> The drop could also happen from the uplink or SF.

The drop counter only increments (initial value being 0), and its value gets reset to 0 when disabled.

8.15.6.1.2.1 Enabling/Disabling Drop Counters

RQ drop counter can be enabled and disabled per device as follows (using VF 0 on PF 0):

```
[dpu]# virtnet modify -p 0 -v 0 device -dc enable
[dpu]# virtnet modify -p 0 -v 0 device -dc disable
```

> Drop counter is attached to a RQ, thus RQ must be created first. This means that the virtio-net device should be probed by the driver on the host OS before running the commands above.

8.15.6.1.2.2 Querying Drop Counters

Per-device Query

To query the drop counter value(s) for a specific device, run:

```
[dpu]# virtnet query -p 0 -v 0 | grep num_desc_drop_pkts
```

If there is more than one RQ for a device, the drop count is the sum of all RQs' values.

Global Drop Counter Summary

To query the total drop count across all devices with a single command:

```
virtnet query rx_drops
```

Output:

Entry	Type	Description
packets	Number	Total RX drop count across all devices

Entry	Type	Description
devices_with_drops	Array	List of devices with non-zero drop counts. Each entry contains **pf**, **type** (**PF** or **VF**), **packets**, and optionally **vf**.

Examples:

- Query total drops across all devices:

```
[dpu]# virtnet query -p 0 -v 0 | grep num_desc_drop_pkts
```

JSON output:

```
{
    "packets": 98777777,
    "devices_with_drops": [
        { "pf": 0, "type": "PF", "packets": 12345 },
        { "pf": 0, "vf": 3, "type": "VF", "packets": 45678 },
        { "pf": 0, "vf": 17, "type": "VF", "packets": 23456 }
    ]
}
```

- Query with no drops present:

```
[dpu]# virtnet query -p 0 -v 0 | grep num_desc_drop_pkts
```

JSON output:

```
{
    "packets": 0,
    "devices_with_drops": []
}
```

Per-device Total Query

To query the total drop count for a single PF or VF (sum of all its RQs):

```
virtnet query -p <PF> [-v <VF>] rx_drops
```

Output:

Entry	Type	Description
pf	Number	PF index
vf	Number	VF index (only present for VF devices)
type	String	**PF** or **VF**
packets	Number	Total RX drop count for this device (sum of all RQs)

Examples:

- Query total drops for a specific VF:

```
[dpu]# virtnet query -p 0 -v 3 rx_drops
```

JSON output:

```
{
    "pf": 0,
    "vf": 3,
    "type": "VF",
    "packets": 45678
}
```

- Query total drops for a PF (includes the PF's own drops only; does not include VFs):

```
[dpu]# virtnet query -p 0 rx_drops
```

JSON output:

```
{
    "pf": 0,
    "type": "PF",
    "packets": 12345
}
```

> ⚠ Querying a PF returns only that PF's own drop counters. It does not include drops from its VFs.

Devices with Drops and Per-queue Detail

To list only devices that have drops, with per-RQ breakdown:

```
virtnet query rx_drops --drops-only
```

Example:

```
[dpu]# virtnet query rx_drops --drops-only
```

JSON output:

```
{
    "packets": 69134,
    "devices_with_drops": [
        {
            "pf": 0, "vf": 3, "type": "VF", "packets": 45678,
            "queues": [
                { "index": 0, "packets": 12345 },
                { "index": 2, "packets": 23456 },
                { "index": 4, "packets": 9877 },
                { "index": 6, "packets": 0 }
            ]
        },
        {
            "pf": 0, "vf": 17, "type": "VF", "packets": 23456,
            "queues": [
                { "index": 0, "packets": 23456 },
                { "index": 2, "packets": 0 }
            ]
        }
    ]
}
```

Check Async Polling Status

```
virtnet query rx_drops status
```

Output:

Entry	Type	Description
`dc_mode`	String	Current drop counter query mode: `async` or `sync`
`poll_freq_sec`	Number	Polling interval in seconds (only shown when `dc_mode` is `async`)
`last_poll_time_ms`	Number	Timestamp of last completed poll cycle in milliseconds (only shown when `dc_mode` is `async` and at least one poll has completed)

Example:

```
[dpu]# virtnet query rx_drops status
```

JSON output:

```
{
    "dc_mode": "async",
    "poll_freq_sec": 5,
    "last_poll_time_ms": 1740394532123
}
```

8.15.6.1.2.3 Async Drop Counter Mode

By default, drop counter queries are performed synchronously — each query reads the counter value directly from hardware firmware. This is accurate but can be slow in large-scale deployments with hundreds of VFs.

Async mode starts a background polling thread that periodically queries hardware and caches the results. When async mode is enabled, all drop counter queries (including `virtnet query` and `virtnet query rx_drops`) return cached values instantly.

Enable Async Mode

```
virtnet modify global dc --mode=<async|sync> --poll_freq <seconds>
```

Parameters:

Option	Argument Type	Required	Description
`--mode=<async`	sync>	String	Yes
`--poll_freq`	Number	No	Polling interval in seconds when using `mode=async`. Valid range: 1–600. Default: 5.

Output:

Entry	Type	Description
`mode`	String	Current drop counter query mode: `async` or `sync`
`poll_freq_sec`	Number	Polling interval in seconds (only shown when `dc_mode` is `async`)

Examples:

- Enable with default polling interval:

```
[dpu]# virtnet modify global dc -mode=async
```

JSON output:

```
{
    "mode": "async",
    "poll_freq_sec": 5
}
```

- Enable with custom polling interval:

```
[dpu]# virtnet modify global dc --mode=async --poll_freq 10
```

JSON output:

```
{
    "mode": "async",
    "poll_freq_sec": 10
}
```

Interval	Trade-off
1–5 seconds	Near real-time, slightly higher CPU
5–30 seconds	Good balance for most deployments
30–600 seconds	Minimal CPU, suitable for infrequent monitoring

Disable Async Mode (Return to Sync)

```
[dpu]# virtnet modify global dc --mode=sync
```

JSON output:

```
{
    "dc_mode": "sync"
}
```

Check Current Mode

```
[dpu]# virtnet query rx_drops status
```

Output when async:

```
{
    "dc_mode": "async",
    "poll_freq_sec": 5,
```

```
    "last_poll_time_ms": 1740394532123
}
```

Output when sync:

```
{
    "dc_mode": "sync"
}
```

> The async mode setting is persisted across controller restarts via the recovery file at `/opt/mellanox/mlnx_virtnet/recovery/global_config`. No manual action is needed to restore the setting after a restart.

8.15.6.1.2.4 Large-scale Deployment Example

For deployments with many VFs (e.g., 576), use async mode to avoid performance bottlenecks:

```
# Step 1: Enable drop counters on all VFs
for vf in $(seq 0 575); do
    virtnet modify -p 0 -v $vf device -dc enable
done

# Step 2: Enable async polling
virtnet modify global dc --mode=async --poll_freq 5

# Step 3: Monitor drops — single command replaces 576 individual queries
virtnet query rx_drops

# Step 4: Drill down to a specific VF if needed
virtnet query -p 0 -v 3 rx_drops
```

8.15.6.1.3 Packet Counter

> Relevant for BlueField-3 only.

The packet counter feature helps the user query the byte-wise packet counters for each Rx queue.

By default, byte-wise packet counters are disabled as that negatively impacts performance. When the user is interested in the debug, enable the packet counter feature using the below command

Packet counter can be enabled and disabled as follows (using VF 0 on PF 0):

```
[dpu]# virtnet modify -p 0 -v 0 device -pkt_cnt enable
[dpu]# virtnet modify -p 0 -v 0 device -pkt_cnt disable
```

- When enabled, byte-wise packet counters are initialized to zero.
- When disabled, the previous values are retained for debugging purposes. The command will still return these old, disabled counter values.

> Packet counters are attached to an RQ. Thus, RQ must be created first. This means that the virtio-net device should be probed by the driver on the host OS before running the commands above.

8.15.6.1.4 Health Statistics

> ⚠ Relevant for BlueField-3 only.

The health statistics are for displaying real-time health information of a specific device.

Output example (using VF 0 on PF 0):

```
[dpu]# virtnet health -p 0 -v 0 show
{
  "pf_id": 0,
  "vf_id": 0,
  "type": "VF",
  "vuid": "MT2306XZ00BPVNETS0D0F2",
  "dev_status": {
    "value": "0xf",
    "    0": "ACK",
    "    1": "DRIVER",
    "    2": "DRIVER_OK",
    "    3": "FEATURES_OK"
  },
  "health_status": "Good",
  "health_recover_counter": 0,
  "dev_health_details": {
    "control_plane_errors": {
      "sf_rqt_update_err": 0,
      "sf_drop_create_err": 0,
      "sf_tir_create_err": 0,
      "steer_rx_domain_err": 0,
      "steer_rx_table_err": 0,
      "sf_flows_apply_err": 0,
      "aarfs_flow_init_err": 0,
      "vlan_flow_init_err": 0,
      "drop_cnt_config_err": 0
    },
    "data_plane_errors": {
      "sq_stall": 0,
      "dma_q_stall": 0,
      "spurious_db_invoke": 0,
      "aux_not_invoked": 0,
      "dma_q_errors": 0,
      "host_read_errors": 0
    }
  }
```

Where:

- **health_status** represents the overall status of the device (**Good** or **Fatal**)
- **dev_health_details** has two sections, **control_plane_errors** and **data_plane_errors** , as explained in the following table:

Counter Name	Description
Control Plane Errors	
sf_rqt_update_err	Counter tallying receive queue table update failures
sf_drop_create_err	Counter tallying drop RQ creation failures
sf_tir_create_err	Counter tallying TIR create failures
steer_rx_domain_err	Counter tallying RX steering rule creation failures
steer_rx_table_err	Counter tallying RX table creation failures

sf_flows_apply_err	Counter tallying packet flow rule creation failures
aarfs_flow_init_err	Counter tallying packet flow initialization failures
vlan_flow_init_err	Counter tallying VLAN flow rule initialization failures
drop_cnt_config_err	Counter tallying drop counter configuration failures
Data Plane Errors	
sq_stall	One or more network send queues stalled without getting completions. This leads traffic stalling for packets flowing over this VQ.
dma_q_stall	QP which is paired to itself issues a read request from the DPA to the host to read either available index or descriptor table. This request does not result in a completion and hangs in a loop waiting for a response.
spurious_db_invoke	Doorbell handler is repeatedly invoked but DPA finds no new data to be read and posted. This could be due to a faulty driver or issue on the DPA side.
aux_not_invoked	To speed up descriptor processing, an auxiliary execution (EU) unit is used if available. The primary thread invokes this EU and waits for the expected thread to run on the auxiliary execution unit. If this EU is not invoked, the primary thread hangs.
dma_q_errors	QP which is paired to itself issues a read request from the DPA to the host to read either an available index or the descriptor table. This request results in an error and the QP becomes unavailable. An internal mechanism detects this error QP and recycles it for use at later stage.

8.15.6.2 Dynamic Interruption Moderation

Dynamic Interrupt Moderation (DIM) adjusts the interrupt moderation settings to optimize packet processing. For guest OS kernels older than version 6.8, DIM offloads this function to the DPU, reducing the interrupt rate from the guest OS.

By lowering the interrupt rate in high-bandwidth traffic scenarios, DIM enhances CPU utilization for both the hypervisor and guest VMs, while maintaining nearly the same bandwidth.

> ⚠ DIM is only supported on BlueField-3.

For example, the following table shows the benefit of using DIM:

	Tx Interrupt Rate (K irq/s)	Rx Interrupt Rate (K irq/s)	Tx Throughput (Gb/s)	Rx Throughput (Gb/s)
DIM Enabled	7.3	7.5	171	181
DIM Disabled	7.5	23.7	175	181

The following test parameters:

- Guest OS kernel version – 5.11.0
- Number of virtio-net device – 1
- Number of QPs – 31
- Queue depth – 1024
- MTU – 1500
- Benchmark – iPerf with 31 streams

8.15.6.2.1 Configuring DIM

DIM is a per-device configuration. To enable or disable it, use this command:

```
[dpu]# virtnet modify -p <pf> [-v <vf>] device -dim {enable | disable}
```

Configuration example:

1. Unload drivers from the guest-OS side:

   ```
   [host]# modprobe -rv virtio_net && modprobe -rv virtio_pci
   ```

2. Enable DIM:

   ```
   [dpu]# virtnet modify -p 0 device -dim enable
   {'pf': '0x0', 'all': '0x0', 'subcmd': '0x0', 'dim_config': 'enable'}
   {
     "errno": 0,
     "errstr": "Success"
   }
   ```

   Using **disable** disables DIM.

3. Load the drivers:

   ```
   [host]# modprobe -v virtio_pci && modprobe -v virtio_net
   ```

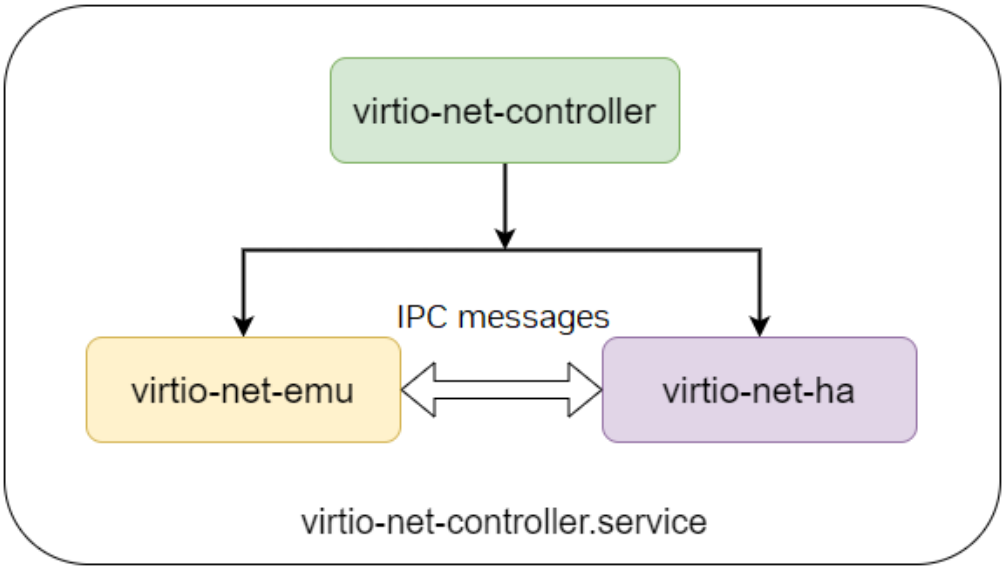
4. Query the device to verify **dim** is enabled:

   ```
   [dpu]# virtnet query -p 0 -b | grep -i dim
         "dim": "enabled"
   ```

8.15.6.3 High Availability

High availability (HA) is essential in network infrastructure to ensure continuous performance with minimal downtime, even during failures.

To support HA, the **virtio-net-controller** process creates the auxiliary processes **virtio-net-emu** and **virtio-net-ha**. The **virtio-net-emu** process handles primary controller functions, while **virtio-net-ha** manages HA. **virtio-net-ha** saves and oversees critical resources from **virtio-net-emu** and restores it to a working state if a failure occurs. The two processes communicate through IPC messages.



High availability is only supported on BlueField-3 and after.

The following table provides possible expected behaviors:

Scenarios	Behavior	Downtime Per Device (sec)	Fallback Action
Virtio-net-emu process crashes (e.g., Segfault)	The **virtio-net-ha** process tries to automatically recover all devices	< 1	The **virtnet restart** command if recovery failed
Device/VQ/SF create/ destroy failures	HA makes sure the existing device is not affected	N/A	Retry or restart service
DPA command timeout	No action from HA; DPA is likely stuck	N/A	The **virtnet restart** command

8.15.6.4 Jumbo MTU

Jumbo MTU is critical for increasing the efficiency of Ethernet and network processing by reducing the protocol overhead (ratio of headers and payload size).

To enable support for jumbo MTU, run the following virtnet command:

```
[dpu]# virtnet modify -p 0 -v 0 device -t 9216
```

The example sets the MTU to 9126 for VF 0 on PF 0.

Jumbo MTU is only supported starting from the following version:

	Release

Upstream	VM kernel: 4.18.0-193.el8.x86_64 (VM Linux version supports big MTU after 4.11)
Ubuntu	DOCA_2.5.0_BSP_4.5.0_Ubuntu_22.04
Virtnet controller	v1.7 or v1.6.26

To configure jumbo MTU (e.g., using VF 0 on PF 0):

1. Change the MTU of the uplink and SF representor from the BlueField:

   ```
   [dpu]# ifconfig p0 mtu 9216
   [dpu]# ifconfig en3f0pf0sf3000 mtu 9216
   ```

   If a bond is configured, change the MTU of the bond rather than **p0**:

   ```
   [dpu]# ifconfig bond0 mtu 9216
   [dpu]# ifconfig en3f0pf0sf3000 mtu 9216
   ```

2. Restart the virtio-net-controller from the BlueField:

   ```
   [dpu]# systemctl restart virtio-net-controller
   ```

3. Unload the virtio driver from the host OS:

   ```
   [host]# modprobe -rv virtio-net
   ```

4. Change the corresponding device MTU on the BlueField:

   ```
   [dpu]# virtnet modify -p 0 -v 0 device -t 9216
   ```

5. Reload virtio driver from the host OS:

   ```
   [host]# modprobe -v virtio-net
   ```

6. Check virtqueue MTU configuration is correct on the BlueField:

   ```
   [dpu]# virtnet query -p 0 -v 0 --dbg_stats | grep jumbo_mtu
       "jumbo_mtu": 1
       "jumbo_mtu": 1
   ```

7. Change the MTU of virtio-net interface from the host OS:

   ```
   [host]# ifconfig <vnet> mtu 9216
   ```

8.15.6.5 Link Aggregation

It is common to use link aggregation (LAG) or bond interfaces to increase reliability, availability, or bandwidth of networking devices. Virtio-net devices support this mode via DPU-side LAG configurations.

To configure the virtio-net-controller in LAG mode must follow a specific procedure due to the dependency on mlx5 RDMA device:

1. Stop the virtio-net-controller to avoid resource leakage (which would be caused by LAG destroying the existing mlx5 RDMA device and creating a new bond RDMA device).

```
[dpu]# systemctl stop virtio-net-controller.service
```

2. Configure the LAG interface for two uplink interfaces from the DPU side. Refer to the "Link Aggregation" page for detailed steps.

   The virtio-net-controller service starts by default. If DPU is rebooted during LAG configuration, it is necessary to stop the controller before creating a bond interfaces from the DPU side.

3. Update the controller configuration file to use bond interface.

```
[dpu]# cat /opt/mellanox/mlnx_virtnet/virtnet.conf
{
  "ib_dev_lag": "mlx5_bond_0",
  "ib_dev_for_static_pf": "mlx5_bond_0",
  "is_lag": 1,
}
```

   Refer to page "Configuration File" for details.

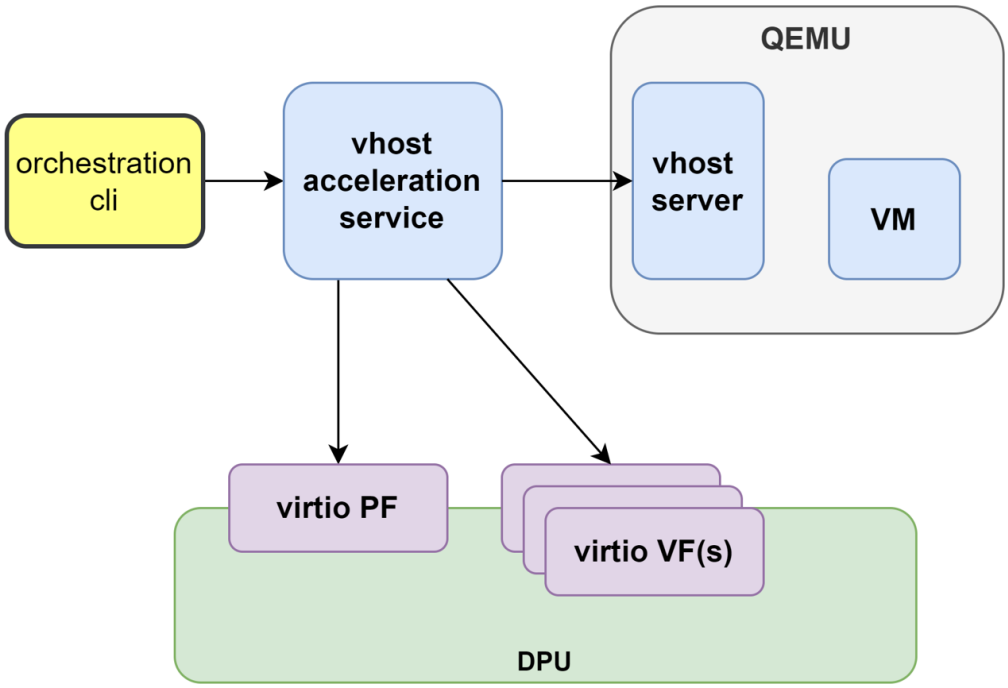
4. Start the controller for the new configuration to take effect.

```
[dpu]# systemctl start virtio-net-controller.service
```

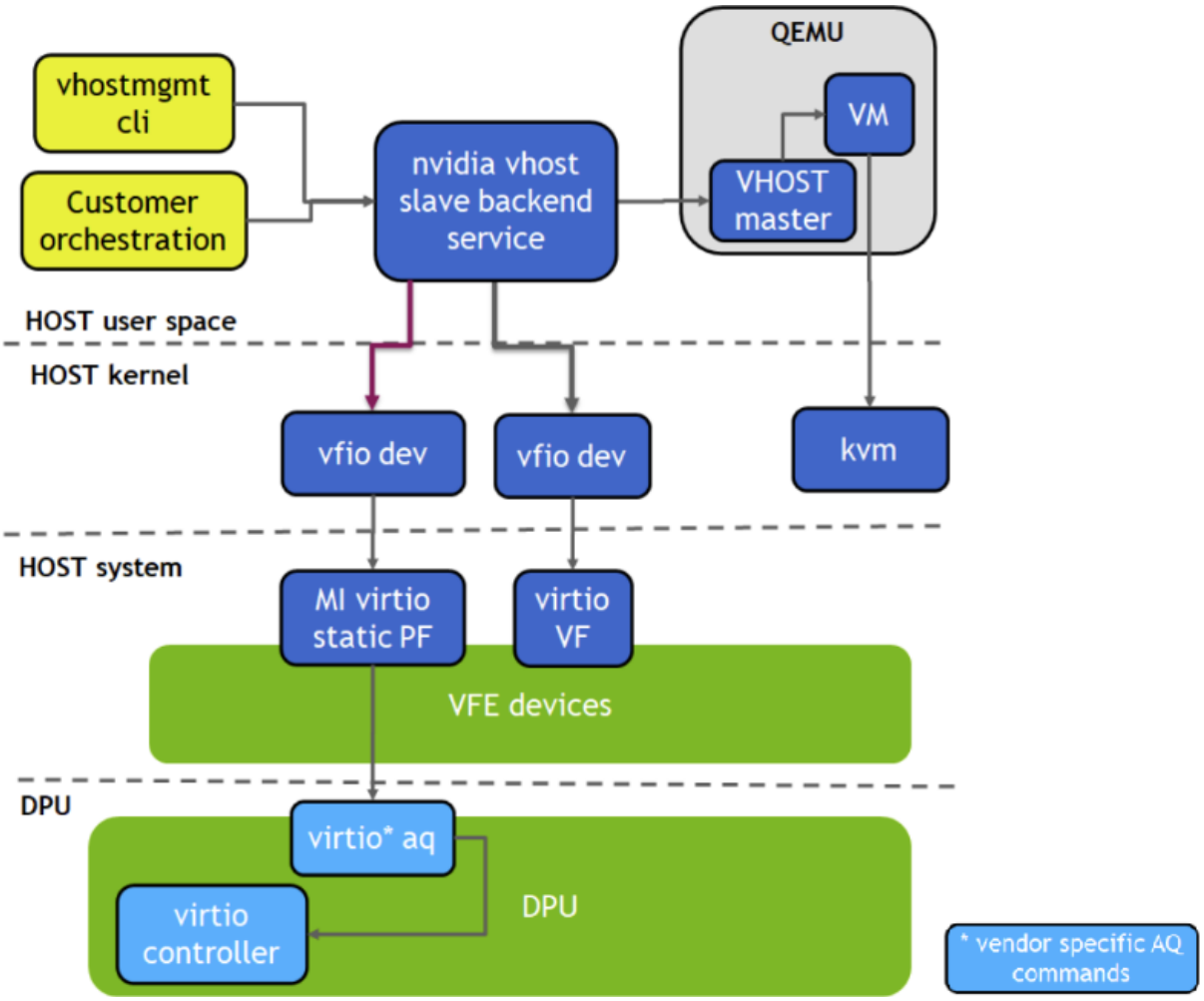
8.15.6.6 Live Migration

8.15.6.6.1 Live Migration Using vHost Acceleration Software Stack

Virtio VF PCIe devices can be attached to the guest VM using the vhost acceleration software stack. This enables performing live migration of guest VMs.



This section provides the steps to enable VM live migration using virtio VF PCIe devices along with vhost acceleration software.



8.15.6.6.1.1 Prerequisites

- Minimum hypervisor kernel version – Linux kernel 5.15 (for VFIO SR-IOV support)
- To use high-availability (the additional **vfe-vhostd-ha** service which can persist datapath when **vfe-vhostd** crashes), this kernel patch must be applied.

8.15.6.6.1.2 Install vHost Acceleration Software Stack

Vhost acceleration software stack is built using open-source BSD licensed DPDK.

- To install vhost acceleration software:

1. a. Clone the software source code:

```
[host]# git clone https://github.com/Mellanox/dpdk-vhost-vfe
```

> The latest release tag is **vfe-24.10.0-rc2**.

b. Build software:

```
[host]# apt-get install libev-dev -y
[host]# apt-get install libev-libevent-dev  -y
[host]# apt-get install uuid-dev  -y
[host]# apt-get install libnuma-dev -y
[host]# meson build --debug -Denable_drivers=vdpa/virtio,common/virtio,common/virtio_mi,common/
virtio_ha
[host]# ninja -C build install
```

- To install QEMU:

> Upstream QEMU later than 8.1 can be used or the following NVIDIA QEMU.

1. a. Clone NVIDIA QEMU sources.

```
[host]# git clone git@github.com:Mellanox/qemu.git -b stable-8.1-presetup
[host]# git checkout 24aaba9255
```

> Latest stable commit is **24aaba9255**.

b. Build NVIDIA QEMU.

```
[host]# mkdir bin
[host]# cd bin
[host]# ../configure --target-list=x86_64-softmmu --enable-kvm
[host]# make -j24
```

8.15.6.6.1.3 Configure vHost on Hypervisor

1. a. Configure 1G huge pages:

```
[host]# mkdir /dev/hugepages1G
[host]# mount -t hugetlbfs -o pagesize=1G none /dev/hugepages1G
[host]# echo 16 > /sys/devices/system/node/node0/hugepages/hugepages-1048576kB/nr_hugepages
[host]# echo 16 > /sys/devices/system/node/node1/hugepages/hugepages-1048576kB/nr_hugepages
```

b. Enable **qemu:commandline** in VM XML by adding the **xmlns:qemu** option:

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
```

c. Assign a memory amount and use 1GB page size for huge pages in VM XML:

```
 <memory unit='GiB'>4</memory>
 <currentMemory unit='GiB'>4</currentMemory>
 <memoryBacking>
    <hugepages>
      <page size='1' unit='GiB'/>
    </hugepages>
 </memoryBacking>
```

d. Set the memory access for the CPUs to be shared:

```
<cpu mode='custom' match='exact' check='partial'>
  <model fallback='allow'>Skylake-Server-IBRS</model>
  <numa>
    <cell id='0' cpus='0-1' memory='4' unit='GiB' memAccess='shared'/>
  </numa>
</cpu>
```

e. Add a virtio-net interface in VM XML:

```
<qemu:commandline>
  <qemu:arg value='-chardev'/>
```

```
  <qemu:arg value='socket,id=char0,path=/tmp/vhost-net0,server=on'/>
  <qemu:arg value='-netdev'/>
  <qemu:arg value='type=vhost-user,id=vhost1,chardev=char0,queues=4'/>
  <qemu:arg value='-device'/>
  <qemu:arg value='virtio-net-pci,netdev=vhost1,mac=00:00:00:00:33:00,vectors=10,page-per-
vq=on,rx_queue_size=1024,tx_queue_size=1024,mq=on,disable-legacy=on,disable-modern=off'/>
</qemu:commandline>
```

8.15.6.6.1.4 Run vHost Acceleration Service

1. Bind the virtio PF devices to the **vfio-pci** driver:

```
[host]# modprobe vfio vfio_pci
[host]# echo 1 > /sys/module/vfio_pci/parameters/enable_sriov
[host]# echo 0x1af4 0x1041 > /sys/bus/pci/drivers/vfio-pci/new_id
[host]# echo 0x1af4 0x1042 > /sys/bus/pci/drivers/vfio-pci/new_id
[host]# echo <pf_bdf> > /sys/bus/pci/drivers/virtio-pci/unbind
[host]# echo <vf_bdf> > /sys/bus/pci/drivers/virtio-pci/unbind
[host]# echo <pf_bdf> > /sys/bus/pci/drivers/vfio-pci/bind
[host]# echo <vf_bdf> > /sys/bus/pci/drivers/vfio-pci/bind
[host]# lspci -vvv -s <pf_bdf> | grep "Kernel driver"
Kernel driver in use: vfio-pci
[host]# lspci -vvv -s <vf_bdf> | grep "Kernel driver"
Kernel driver in use: vfio-pci
```

> Example of **<pf_bdf>** or **<vf_bdf>** format: **0000:af:00.3**

2. Run the vhost acceleration software service by starting the **vfe-vhostd** service:

```
[host]# systemctl start vfe-vhostd
```

> A log of the service can be viewed by running the following:
>
> ```
> [host]# journalctl -u vfe-vhostd
> ```

3. Provision the virtio-net PF:

```
[host]# /usr/local/bin/vfe-vhost-cli mgmtpf -a <pf_bdf>
```

   Wait on the virtio-net-controller to finish handling PF FLR.

4. Enable SR-IOV and create a VF (or more):

```
[host]# echo 1 > /sys/bus/pci/devices/<pf_bdf>/sriov_numvfs
[host]# lspci | grep Virtio
0000:af:00.1 Ethernet controller: Red Hat, Inc. Virtio network device
0000:af:00.3 Ethernet controller: Red Hat, Inc. Virtio network device
```

5. Add a VF representor to the OVS bridge on the BlueField:

```
[dpu]# virtnet query -p 0 -v 0 | grep sf_rep_net_device
"sf_rep_net_device": "en3f0pf0sf3000",
[dpu]# ovs-vsctl add-port ovsbr1 en3f0pf0sf3000
```

6. Provision the virtio-net VF:
   a. On BlueField, change VF MAC address or other device options:

```
[dpu]# virtnet modify -p 0 -v 0 device -m 00:00:00:00:33:00
```

   b. Add VF into vfe-dpdk

```
[host]# /usr/local/bin/vfe-vhost-cli vf -a <vf_bdf> -v /tmp/vhost-net0
```

> ⚠ If the SR-IOV is disabled and reenabled, the user must re-provision the VFs. `00:00:00:00:33:00` is a virtual MAC address used in VM XML.

8.15.6.6.1.5 Start the VM

```
[host]# virsh start <vm_name>
```

8.15.6.6.1.6 HA Service

Running the `vfe-vhostd-ha` service allows the datapath to persist should `vfe-vhostd` crash:

```
[host]# systemctl start vfe-vhostd-ha
```

8.15.6.6.1.7 Simple Live Migration

1. Prepare two identical hosts and perform the provisioning of the virtio device to DPDK on both.
2. Boot the VM on one server:

```
[host]# virsh migrate --verbose --live --persistent <vm_name> qemu+ssh://<dest_node_ip_addr>/system --
unsafe
```

8.15.6.6.1.8 Remove Device

When finished with the virtio devices, use following commands to remove them from DPDK:

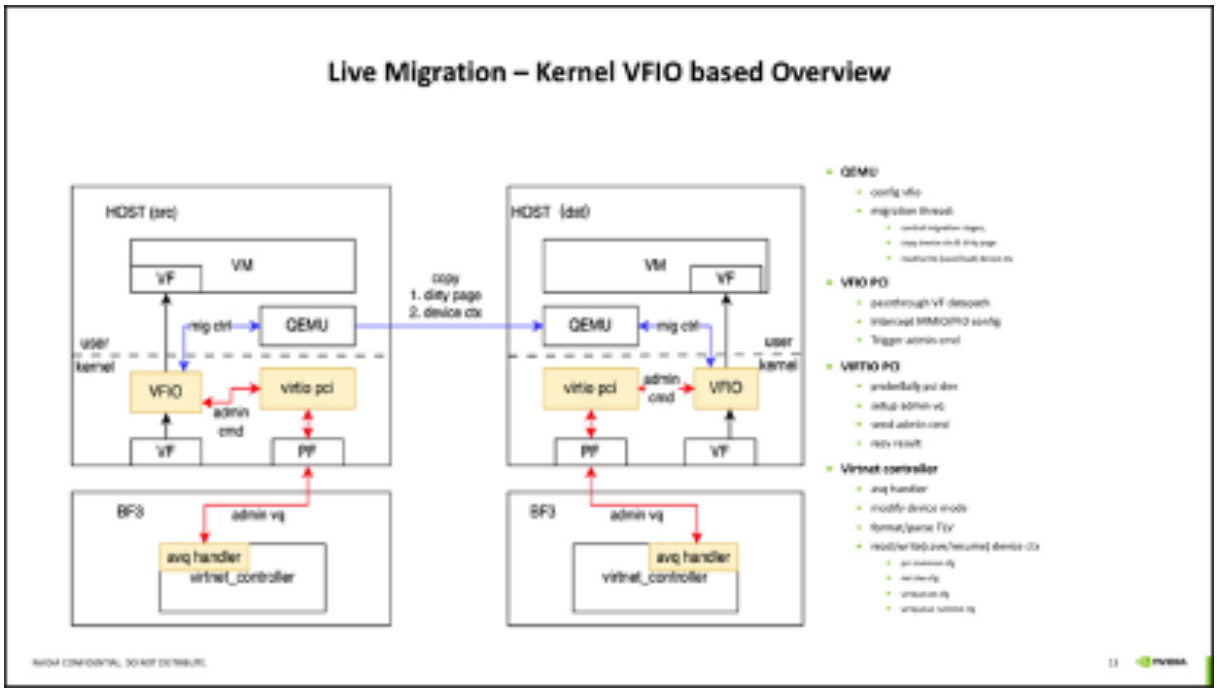
```
[host]# /usr/local/bin/vfe-vhost-cli vf -r <vf_bdf>
[host]# /usr/local/bin/vfe-vhost-cli mgmtpf -r <pf_bdf>
```

> ⚠ During live migration, the device state may change temporarily. As a result, Linux NetworkManager may reset the associated network interface properties (e.g., IP address).
>
> To prevent NetworkManager from managing a specific interface, run:
>
> ```
> nmcli device set {device-interface} managed no
> ```

8.15.6.6.2 Live Migration Using VFIO With Full Emulation

Virtio VF PCIe devices can be attached to the guest VM using the `virtio-vfio-pci` driver. This enables performing live migration of guest VMs.

This section demonstrates how to perform basic live migration of a QEMU VM with a virtio VF assigned to it. It does not explain how to create VMs using libvirt or directly via QEMU.



8.15.6.6.2.1 Prerequisites

1. Minimum Hypervisor kernel version - Linux kernel 6.13-rc2 with **virtio_vfio_pci** and IOMMU dirty page tracking
2. Minimum qemu version - 9.1
3. Minimum libvirt version - 9.2

8.15.6.6.2.2 DPU Configuration

1. Install new **virtio-net-controller** (version **25.04** or newer) on source and destination systems.
2. Add the following flags on the source and destination systems.

```
[dpu]# vim /opt/mellanox/mlnx_virtnet/virtnet.conf
{
  ...
  "virtio_spec_admin_legacy": 1,
  "virtio_spec_admin_lm": 1
}
```

3. Restart the controller
4. Provision device attributes. [After loading **virtio-pci-vfio** driver and before starting the VM]
5. Get the MAC of the source device

```
[dpu]# virtnet query -p $pf_id -v $vf_id | grep "\"mac"
```

6. Set the MAC of the destination device

```
[dpu]# virtnet modify -p $dst_pf_id -v $dst_vf_id device -m $mac
```

8.15.6.6.2.3 Kernel Configuration

Needs to be compiled with the driver **virtio_vfio_pci** enabled. (i.e. **CONFIG_VIRTIO_VFIO_PCI**).

To load the driver, run:

```
[host]# modprobe virtio_vfio_pci
```

8.15.6.6.2.4 QEMU Configuration

1. Needs to be compiled with VFIO_PCI enabled (this is enabled by default).
2. Add the following to qemu.conf:

```
user = "root"
group = "root"
cgroup_device_acl = [
    "/dev/null", "/dev/full", "/dev/zero",
    "/dev/random", "/dev/urandom",
    "/dev/ptmx", "/dev/kvm",
    "/dev/iommu", "/dev/vfio/devices/vfio0",
    "/dev/vfio/devices/vfio1"
]
```

3. Restart libvirt

8.15.6.6.2.5 Host Preparation

As stated earlier, creating the VMs is beyond the scope of this guide, and we assume that they have already been created. However, the VM configuration should be a migratable configuration, similarly to how it is done without virtio VFs.

> ⚠ The steps below should be done before running the VMs.

1. Create the VFs that will be assigned to the VMs.

```
[host]# echo "<NUM_OF_VFS>" > /sys/bus/pci/devices/<PF_BDF>/sriov_numvfs
```

2. Unbind the VFs from **virtio-pci**, run:

```
[host]# echo '<VF_BDF>' > /sys/bus/pci/drivers/virtio-pci/unbind
```

3. Assign the VFs to the VMs
    a. Edit the VMs XML file, run:

```
[host]# virsh edit <VM_NAME>
```

    b. Enable **qemu:commandline** in VM XML by adding the **xmlns:qemu** option:

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
```

    c. Assign the VFs to the VM by adding the following under the **device** tag:

```
<hostdev mode='subsystem' type='pci' managed='no'>
    <driver name='vfio'/>
    <source>
    <address domain='0x0000' bus='0xb1' slot='0x00' function='0x4'/>
    </source>
```

```
    <alias name='hostdev0'/>
    <address type='pci' domain='0x0000' bus='0x01' slot='0x01' function='0x0'/>
</hostdev>
```

> ⚠ The domain, bus, slot, and function values above are dummy values; replace them with your VFs values.

4. Set the source VM
   a. Edit the source VM XML file, run:

```
[host]# virsh edit <VM_NAME>
```

   b. Set up the source VM by adding the following under **domain** tag:

```
<qemu:commandline>
    <qemu:arg value='-object'/>
    <qemu:arg value='iommufd,id=iommufd0'/>
    <qemu:arg value='-snapshot'/>
</qemu:commandline>
<qemu:override>
    <qemu:device alias='hostdev0'>
        <qemu:frontend>
            <qemu:property name='enable-migration' type='string' value='on'/>
            <qemu:property name='iommufd' type='string' value='iommufd0'/>
        </qemu:frontend>
    </qemu:device>
</qemu:override>
```

> ⚠ To save the file, the above "xmlns:qemu" attribute of the "domain" tag must also be added.

5. Set the destination VM in incoming mode
   a. Edit the destination VM XML file, run:

```
[host]# virsh edit <VM_NAME>
```

   b. Set the destination VM in migration incoming mode by adding the following under the **domain** tag:

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
[...]
    <qemu:commandline>
        <qemu:arg value='--incoming'/>
        <qemu:arg value='tcp:<DEST_IP>:<DEST_PORT>'/>
        <qemu:arg value='-object'/>
        <qemu:arg value='iommufd,id=iommufd0'/>
    </qemu:commandline>
    <qemu:override>
        <qemu:device alias='hostdev0'>
        <qemu:frontend>
            <qemu:property name='enable-migration' type='string' value='on'/>
            <qemu:property name='iommufd' type='string' value='iommufd0'/>
        </qemu:frontend>
        </qemu:device>
    </qemu:override>
</domain>
```

> ⚠ To save the file, the above "xmlns:qemu" attribute of the "domain" tag must also be added.

6. Bind the VFs to **virtio_vfio_pci** driver
   a. Detach the VFs from **libvirt** management, run:

```
[host]# virsh nodedev-detach pci_<VF_BDF>
```

b. Unbind the VFs from **vfio-pci** driver (the VFs are automatically bound to it after running "**virsh nodedev-detach**"), run:

```
[host]# echo '<VF_BDF>' > /sys/bus/pci/drivers/vfio-pci/unbind
```

c. Set driver override, run:

```
[host]# echo 'virtio_vfio_pci' > /sys/bus/pci/devices/<VF_BDF>/driver_override
```

d. Bind the VFs to **virtio_vfio_pci** driver, run:

```
[host]# echo '<VF_BDF>' > /sys/bus/pci/drivers/virtio_vfio_pci/bind
```

8.15.6.6.2.6 Running the Migration

1. Start the VMs in source and in destination, run:

```
[host]# virsh start <VM_NAME>
```

2. Enable **switchover-ack** QEMU migration capability. Run the following commands both in the source and the destination:

```
[host]# virsh qemu-monitor-command <VM_NAME> --hmp "migrate_set_capability return-path on"
[host]# virsh qemu-monitor-command <VM_NAME> --hmp "migrate_set_capability switchover-ack on"
```

3. [Optional] Configure the migration bandwidth and downtime limit on the source side:

```
[host]# virsh qemu-monitor-command <VM_NAME> --hmp "migrate_set_parameter max-bandwidth <VALUE>"
[host]# virsh qemu-monitor-command <VM_NAME> --hmp "migrate_set_parameter downtime-limit <VALUE>"
```

4. Start migration by running the migration command on the source side:

```
[host]# virsh qemu-monitor-command <VM_NAME> --hmp "migrate -d tcp:<DEST_IP>:<DEST_PORT>"
```

5. Check the migration status by running the info command on the source side:

```
[host]# virsh qemu-monitor-command <VM_NAME> --hmp "info migrate"
```

When the migration status is **completed** it means the migration has finished successfully.

During live migration, the device state may change temporarily. As a result, Linux NetworkManager may reset the associated network interface properties (e.g., IP address).

To prevent NetworkManager from managing a specific interface, run:

```
nmcli device set {device-interface} managed no
```

8.15.6.7 Live Update

Live update minimizes network interface downtime by performing online upgrade of the virtio-net controller without necessitating a full restart.

8.15.6.7.1 Requirements

To perform a live update, the user must install a newer version of the controller either using the **rpm** or **deb** package (depending on the OS distro used). Run:

For Ubuntu/ Debian	`[dpu]# dpkg --force-all -i virtio-net-controller-x.y.z-1.mlnx.aarch64.deb`
For CentOS/ RedHat	`[dpu]# rpm -Uvh virtio-net-controller-x.y.z-1.mlnx.aarch64.rpm --force`

8.15.6.7.2 Check Versions

Before staring live update, the following command can be used to check the version of the original and destination controllers:

```
[dpu]# virtnet version
  {
    "Original Controller": "v24.10.13"
  },
  {
    "Destination Controller": "v24.10.16"
  }
```

8.15.6.7.3 Start Updating

If no errors occur, issue the following command to start the live update process:

```
[dpu]# virtnet update -s
```

If an error indicates that the **update** command is unsupported, this means the controller version you are attempting to install is outdated. Reinstalling the correct version resolves the issue.

8.15.6.7.4 Check Status

During the update process, the following command may be used to check the update status:

```
[dpu]# virtnet update -t
```

Example output:

```
{
  "status": "inactive",                    # updating status, whether live update is finished or ongoing
```

```
    "last live update status": "success",       # last live update status
    "time_used (s)": 1.655439                   # time cost for last live update
}
```

During the update, it is recommended to not issue any virtnet CLI command.

When the update process completes successfully, the command **virtnet update status** reflects the status accordingly

If a device is actively migrating, the existing **virtnet** commands appear as "migrating" for that specific device so that the user can retry later.

Limitation

When live update is in progress, hotplug/unplug and VF creation/deletion are not supported.

8.15.6.8 Mergeable Rx Buffer

The Mergeable Rx Buffer is a receive-side-only performance enhancement. When successfully negotiated with the driver, this feature allows the device to utilize multiple descriptors to accommodate a single jumbo-sized packet received from the network. It significantly improves memory utilization and throughput in environments configured for large Maximum Transmission Units (MTUs), such as 9K jumbo frames.

8.15.6.8.1 Configuration

Administrators control this feature using the **VIRTIO_NET_F_MRG_RXBUF** (bit 15) feature flag.

- Default State: Disabled.
- Scope: Can be enabled on a per-device basis.

Refer to the "DOCA Virtio-net Service Guide#Enabling and Disabling Virtio-net Features" section for the exact **virtnet modify** command syntax.

8.15.6.8.2 Limitations

Before enabling the Mergeable Rx Buffer, carefully review the following environmental constraints:

Limitation	Description
Strict MTU ceiling	The absolute maximum supported MTU when utilizing this feature is 9000. The feature will fail to operate if the MTU is set to 9216.
Performance degradation (standard MTU)	Because the number of descriptors per Work Queue Entry (WQE) depends on the MTU size, enabling this feature with a default MTU (1500) is not recommended and will negatively impact performance.
Performance degradation (small packets)	The system will experience a performance drop when processing high rates of small-sized packets (e.g., 64 bytes) from the wire. Reserve this feature exclusively for heavy jumbo-frame traffic.

Limitation	Description
Feature Incompatibility	The Mergeable Rx Buffer feature is strictly incompatible with the Packed Virtqueue feature (`VIRTIO_F_RING_PACKED`). You cannot enable both simultaneously on the same device.

8.15.6.9 Performance Tuning

8.15.6.9.1 Number of Queues and MSIX

8.15.6.9.1.1 Driver Configuration

The virtio-net driver can configure the number of combined channels via ethtool. This determines how many virtqueues (VQs) can be used for the netdev. Normally, more VQs result in better overall throughput when multi-threaded (e.g., iPerf with multiple streams).

```
[host]# ethtool -l eth0
Channel parameters for eth0:
Pre-set maximums:
RX:             n/a
TX:             n/a
Other:          n/a
Combined:       31
Current hardware settings:
RX:             n/a
TX:             n/a
Other:          n/a
Combined:       15
```

Therefore, it is common to pick a larger number (less than pre-set maximums) of channels using the following command.

Normally, configuring the combined number of channels to be the same as number of CPUs available on the guest OS will yield good performance.

```
[host]# ethtool -L eth0 combined 31
[host]# ethtool -l eth0
Channel parameters for eth0:
Pre-set maximums:
RX:             n/a
TX:             n/a
Other:          n/a
Combined:       31
Current hardware settings:
RX:             n/a
TX:             n/a
Other:          n/a
Combined:       31
```

8.15.6.9.1.2 Device Configuration

To reach the best performance, it is required to make sure each tx/rx queue has an assigned MSIX. Check the information of a particular device and make sure `num_queues` is less than `num_msix` .

```
[dpu]# virtnet query -p 0 -b | grep -i num_
      "num_msix": "64",
      "num_queues": "8",
```

If **num_queues** is greater than **num_msix**, it is necessary to change mlxconfig to reserve more MSIX than queues. It is determined by the **VIRTIO_NET_EMULATION_NUM_VF_MSIX** and **VIRTIO_NET_EMULATION_NUM_MSIX**. Please refer to the "DOCA Virtio-net Service Guide#Virtio-net Deployment" page for more information.

8.15.6.9.2 Queue Depth

By default, queue depth is set to 256. It is common to use a larger queue depth (e.g., 1024). This cannot be requested from the driver side but must be done from the device side.

Refer to the "DOCA Virtio-net Service Guide#Virtnet CLI Commands" page to learn how to modify device **max_queue_size**.

8.15.6.9.3 MTU

To improve performance, the user can use jumbo MTU. Refer to "DOCA Virtio-net Service Guide#Jumbo MTU" page for information regarding MTU configuration.

8.15.6.10 Device State Recovery

The recovery process is critical for restoring both control plane and data plane statuses during disruptive events, such as a controller restart, a live update, or a live migration.

The system relies on persistent JSON files stored in the **/opt/mellanox/mlnx_virtnet/recovery** directory. Each physical function (PF) or virtual function (VF) device maintains a corresponding recovery file, uniquely named after the device's VUID.

8.15.6.10.1 Recovery File Structure

The controller saves the following configuration states to the recovery file and automatically restores them when necessary:

Entry Name	Type	Description
port_ib_dev	String	The RDMA device name on which the **virtio-net** device is created.
pf_id	Number	The ID of the Physical Function (PF).
vf_id	Number	The ID of the Virtual Function (VF). *(Valid for VFs only).*
function_type	String	Identifies the function as either a **pf** or **vf**.
bdf_raw	Number	The **virtio-net** device Bus:Device:Function (BDF) represented as a **uint16** type.
device_type	String	Specifies if the device is **static** or **hotplug**. *(Valid for PFs only).*
mac	String	The MAC address of the device.

Entry Name	Type	Description
`pf_num`	Number	The PCIe function number.
`sf_num`	Number	The Sub-Function (SF) number utilized for this `virtio-net` device.
`mq`	Number	The number of multi-queues created for this `virtio-net` device.
`rx_mode`	Number	A 32-bit value representing reception modes. Bits 0–5 correspond to: • 0: Promiscuous • 1: All multicast • 2: All unicast • 3: No multicast • 4: No unicast • 5: No broadcast
`vlan_tags`	Array	An array storing VLAN IDs (0–4095) configured for VLAN filtering.
`mac_table`	Array	Contains MAC addresses for filtering, alongside metadata (e.g., entry count, first multicast index, and overflow flags).
`offloads`	Number	Stores the active state of hardware offload features (e.g., `VIRTIO_NET_F_GUEST_CSUM`).
`sf_parent_device`	String	The Sub-Function (SF) parent device identifier.
`rx_mode_cmd`	Number	The RX mode command state.
`mac_cmd`	Number	The MAC command state.
`vlan_table_cmd`	Number	The VLAN table command state.
`announce_cmd`	Number	The Announce command state.
`announce`	Number	The Announce flag status.
`lm_status`	Number	The current Live Migration status.
`dev_mode`	Number	The operational device mode.
`dirty_log`	String	Dirty log tracking information (vital for live migration).
`transitional`	Number	Transitional mode flags.
`sys_path`	String	The system path required for SF restoration.
`sys_name`	String	The system name required for SF restoration.
`hash_cfg`	Object	The active Hash configuration.

Entry Name	Type	Description
pkt_cnt	Number	Packet count mode flags.
net_dim	Number	Net-DIM (Dynamic Interrupt Moderation) mode flags.
aarfs	Number	aRFS (Accelerated Receive Flow Steering) mode flags.
hp_host_awareness_mode	Number	Hotplug host awareness mode configuration (e.g., AB mode status).

8.15.6.10.2 Example Recovery File

The following JSON payload illustrates a standard recovery file for a hotplugged PF device:

```
{
  "port_ib_dev": "mlx5_0",
  "pf_id": 0,
  "function_type": "pf",
  "bdf_raw": 57611,
  "device_type": "hotplug",
  "mac": "0c:c4:7a:ff:22:93",
  "pf_num": 0,
  "sf_num": 2000,
  "mq": 3
}
```

8.15.6.10.3 Use Cases

Depending on the actions of the BlueField or host, recovery may or may not be performed. Please refer to the following table for individual scenarios:

	DPU Actions			Host Actions				
	Restart Controller	Live Update	Hot Unplug	Destroy VFs	Unload Driver	Power Cycle Host & DPU	Warm Reboot	Live Migration
Static PF	Recover	Recover	N/A	N/A	Recover	No recover	Recover	Recover
Hotplug PF	Recover	Recover	No recover	N/A	Recover	No recover	Recover	Recover
VF	Recover	Recover	N/A	Recovery file deleted	No Recover	No recover	No recover	Recover

These recovery files are internal to the controller and should not be modified.

Controller recovery is enabled by default and does not need user configuration or intervention. When the **mlxconfig** settings used by the controller take effect, the newly started controller service automatically deletes all recovery files.

Recovery File Validation and Corruption Handling

During startup, the controller strictly validates the integrity of all stored JSON recovery files before applying any state restorations. If the controller detects corrupted, malformed, or invalid data in any single recovery file during the startup sequence, it will automatically purge all recovery files in the directory and perform a fresh restart.

8.15.6.11 Transitional Device

A transitional device is a virtio device which supports drivers conforming to virtio specification **1.x** and legacy drivers operating under virtio specification **0.95** (i.e., legacy mode) so servers with old Linux kernels can still utilize virtio-based technology.

Currently, only transitional VF devices are supported.

Host kernel version must be newer than v6.9.

When using this feature, vfe-vdpa-dpdk solutions cannot be used anymore, including vfe-vdpa-dpdk live migration.

Libvirt does not support the **virtio_vfio_pci** kernel driver. Use the QEMU command line to start the VM instead.

8.15.6.11.1 Transitional Virtio-net VF Device

1. Configure virtio-net SR-IOV. Refer to "DOCA Virtio-net Service Guide#Virtio-net Deployment" for details.
2. Modify the configuration file to add the **"virtio_spec_admin_legacy": 1** option.

```
[dpu]# cat /opt/mellanox/mlnx_virtnet/virtnet.conf
{
...
"virtio_spec_admin_legacy": 1,
...
}
```

3. Restart the virtio-net controller for the configuration to take effect:

```
[dpu]# systemctl restart virtio-net-controller.service
```

4. Create virtio-net VF devices on the host:

```
[host]# modprobe -r virtio_net
[host]# modprobe -r virtio_pci
[host]# modprobe virtio_net
[host]# modprobe virtio_pci
[host]# echo <vf_num> > /sys/bus/pci/devices/<pf_bdf>/sriov_numvfs
```

5. Bind the VF devices with the **virtio_vfio_pci** kernel driver:

```
[host]# echo <vf_bdf> > /sys/bus/pci/devices/<vf_bdf>/driver/unbind
[host]# echo 0x1af4 0x1041 > /sys/bus/pci/drivers/virtio_vfio_pci/new_id
[host]# modprobe -v virtio_vfio_pci
[host]# lspci -s <vf_bdf> -vvv | grep -i virtio_vfio_pci
Kernel driver in use: virtio_vfio_pci
```

6. Add the following option into the QEMU command line to passthrough the VF device into the VM:

```
-device vfio-pci,host=<vf_bdf>,id=hostdev0,bus=pci.<#BUS_IN_VM>,addr=<#FUNC_IN_VM>
```

7. Load virtio-net driver as legacy mode inside the VM:

```
[vm]# modprobe -r virtio_net
[vm]# modprobe -r virtio_pci
[vm]# modprobe virtio_pci force_legacy=1
[vm]# modprobe virtio_net
[vm]# lspci -s <vf_bdf_in_vm> -n
00:0a.0 0200: 1af4:1000
```

8. Verify that the VF is a transitional device:

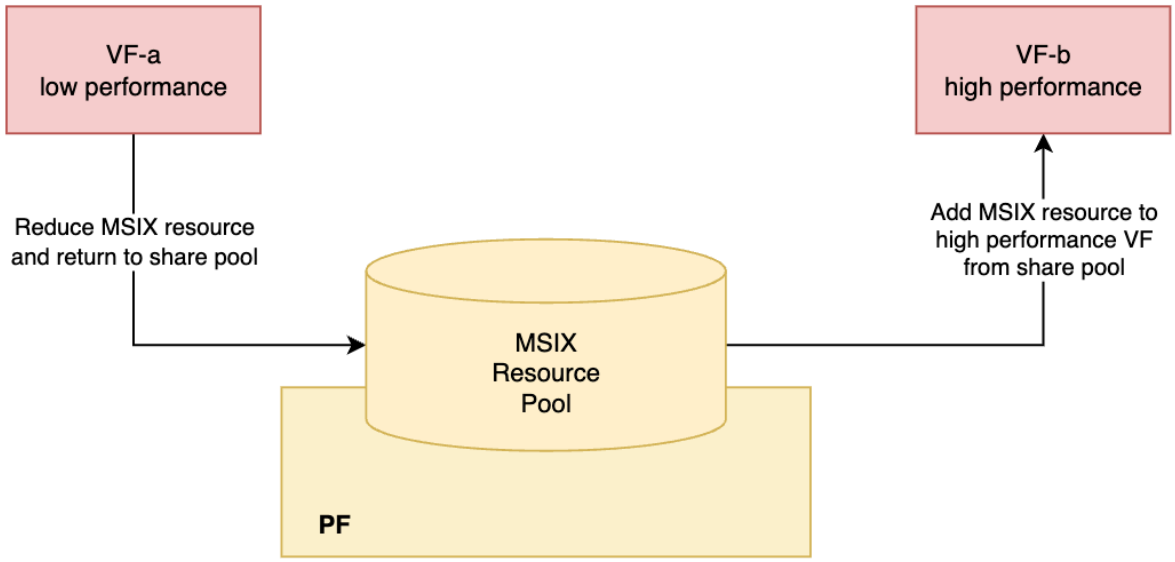
```
[dpu]# virtnet query -p <pf_id> -v <vf_id> | grep transitional
      "transitional": 1,
```

8.15.6.12 VF Dynamic MSIX

In virtio-net controller, each VF gets the same number of MSIX and virtqueues (VQs) so that each data VQ has a MSIX assigned. This means that changing the number of MSIX updates the number of VQs.

By default, each VF is assigned with the same number of MSIX, the default number is determined by the minimum of **NUM_VF_MSIX** and **VIRTIO_NET_EMULATION_NUM_MSIX**.

Using dynamic VF MSIX, a VF can be assigned with more MSIX/queues than its default. MSIX hardware resources of all VF devices are managed by PF via a shared MSIX pool. The user can reduce the MSIX of one VF, thus releasing its MSIX resources to the shared pool. On the other hand, another VF can be assigned with more MSIX than its default to gain more performance.



8.15.6.12.1 Firmware Configuration

The emulation VF device uses **VIRTIO_NET_EMULATION_NUM_VF_MSIX** to set the MSIX number.

VIRTIO_NET_EMULATION_NUM_VF_MSIX is available to set the MSIX number of the emulation VF device. For the emulation VF device, uses the new configuration VIRTIO_NET_EMULATION_NUM_VF_MSIX instead of the old configuration NUM_VF_MSIX.

- If **VIRTIO_NET_EMULATION_NUM_VF_MSIX** !=0, **VIRTIO_NET_EMULATION_NUM_ MSIX** is used for the PF only, and VF uses **VIRTIO_NET_EMULATION_NUM_VF_MSIX** .
  For example, to configure the default MSIX number for a VF to 32:

  ```
  [dpu]# mlxconfig -y -d 03:00.0 s VIRTIO_NET_EMULATION_NUM_ MSIX=32 VIRTIO_NET_EMULATION_NUM_VF_MSIX=32
  ```

- If **VIRTIO_NET_EMULATION_NUM_VF_MSIX** ==0, **VIRTIO_NET_EMULATION_NUM_ MSIX** is used for the PF and VF.

The default number of MSIX for each VF is determined by **minimum(NUM_VF_MSIX, VIRTIO_NET_EMULATION_NUM_MSIX)** . For example, to configure the default MSIX number for a VF to 32:

```
[dpu]# mlxconfig -y -d 03:00.0 s VIRTIO_NET_EMULATION_NUM_MSIX=32 NUM_VF_MSIX=32
```

Power cycle the BlueField and host to have the **mlxconfig** taking effect.

8.15.6.12.2 MSIX

8.15.6.12.2.1 MSIX Capability

The MSIX pool for VFs is managed by their PF. To check the share pool size, run the following command (using PF 0 as example):

```
[dpu]# virtnet list | grep -i '"pf_id": 0' -A 8 | grep -i msix_num_pool_size
```

By default, the share pool size is empty (0), since all MSIX resources have already been allocated to VFs evenly. Upon reducing the MSIX of one or more VFs, the reduced MSIX is released back to the pool.

However, the number of MSIX can be assigned to a given VF is also bound by capability. To check those caps, run the following command:

```
[dpu]# virtnet list | grep -i '"pf_id": 0' -A 10 | grep -i max_msix_num
[dpu]# virtnet list | grep -i '"pf_id": 0' -A 10 | grep -i min_msix_num
```

To check the currently assigned number of MSIX, run the following command:

```
[dpu]# virtnet query -p 0 -v 0 | grep num_msix
```

If **num_msix** is less than **max_msix_num** cap, more MSIX can be assigned to the VF.

8.15.6.12.2.2 Reallocating VF MSIX

To allocate more MSIX to one VF, there should be MSIX available from the pool. This is done by reducing the MSIX from another VF(s).

The following example shows the steps to reallocate MSIX from VF1 to VF0, assuming that each VF has **32** MSIX available as default:

1. Unbind both VF devices from host driver.

   ```
   [host]# echo <vf0_bdf> > /sys/bus/pci/drivers/virtio-pci/unbind
   [host]# echo <vf1_bdf> > /sys/bus/pci/drivers/virtio-pci/unbind
   ```

2. Reduce the MSIX of VF1.

   ```
   [dpu]# virtnet modify -p 0 -v 1 device -n 4
   ```

3. Check pool size of PF0.

   ```
   [dpu]# virtnet list | grep -i '"pf_id": 0' -A 8 | grep -i msix_num_pool_size
   ```

   Confirm the reduced MSIX are added to the share pool.

4. Increase the MSIX of VF0.

   ```
   [dpu]# virtnet modify -p 0 -v 0 device -n 48
   ```

5. Check the MSIX of VF0.

   ```
   [dpu]# virtnet query -p 0 -v 0 | grep -i num_msix
   ```

6. Bind both VF devices to host driver.

   ```
   [host]# echo <vf0_bdf> > /sys/bus/pci/drivers/virtio-pci/bind
   [host]# echo <vf1_bdf> > /sys/bus/pci/drivers/virtio-pci/bind
   ```

> ⚠ The number of MSIX must be an even number greater than 4.

8.15.6.12.2.3 MSIX Limitations

- MSIX and QP configuration is mutually exclusive (i.e., only one of them can be configured at a time). For example, the following **modify** command should result in failure:

```
[dpu]# virtnet modify -p 0 -v 1 device -qp 2 -n 6
```

- To use a VF, make sure to assign a valid MSIX number:

```
[dpu]# virtnet modify -p 0 -v 1 device -n 10
```

  The minimum number of MSIX resources required for the VF to load the host driver is 4 if **VIRTIO_NET_F_CTRL_VQ** is negotiated, or 2 if it is not.

- The MSIX resources of a VF can be reduced to 0, but doing so prevents the VF from functioning.

```
[dpu]# virtnet modify -p 0 -v 1 device -n 0
```

8.15.6.12.3 Queue Pairs

Queue pairs (QPs) are the number of data virtio queue (VQ) pairs. Each VQ pair has one transmit (TX) queue and one receive (RX) queue. These pairs are dedicated to handling data traffic and do not include control or admin VQs.

8.15.6.12.3.1 QP Capability

The QP pool for VFs is managed by their PF.

To check the shared pool size, run the following command (using PF 0 as example):

```
[dpu]# virtnet list | grep -i '"pf_id": 0' -A 13 | grep -i qp_pool_size
```

By default, the shared pool size is empty (0), since all QP resources have already been allocated to VFs evenly. Upon reducing the QP of one or more VFs, the reduced QP is released back into the pool.

However, the number of QPs assignable to a VF depends on its supported capabilities. To verify these capabilities, run the following command:

```
[dpu]# virtnet list | grep -i '"pf_id": 0' -A 12 | grep -i max_num_of_qp
[dpu]# virtnet list | grep -i '"pf_id": 0' -A 12 | grep -i min_num_of_qp
```

To check the currently assigned number of QPs, run the following command:

```
[dpu]# virtnet query -p 0 -v 0 | grep max_queue_pairs
```

If **max_queue_pairs** is less than **max_num_of_qp** cap, then more QPs can be assigned to the VF.

8.15.6.12.3.2 Reallocating VF QPs

To allocate more QPs to one VF, there should be QPs available from the pool as explained in the previous section.

The following example illustrates the process of reallocating a QP from VF1 to VF0, assuming that each VF initially has 32 QPs available by default:

1. Unbind both VF devices from the host driver:

```
[host]# echo <vf0_bdf> > /sys/bus/pci/drivers/virtio-pci/unbind
[host]# echo <vf1_bdf> > /sys/bus/pci/drivers/virtio-pci/unbind
```

2. Reduce the number of QPs VF1 has:

```
[dpu]# virtnet modify -p 0 -v 1 device -qp 1
```

3. Check the pool size of PF0 and confirm that the reduced number of QPs are added to the shared pool:

```
[dpu]# virtnet list | grep -i '"pf_id": 0' -A 13 | grep -i qp_pool_size
```

4. Increase the number of QPs VF0 has:

```
[dpu]# virtnet modify -p 0 -v 0 device -qp 23
```

5. Check the number of QPs VF0 has:

```
[dpu]# virtnet query -p 0 -v 0 | grep -i max_queue_pairs
```

6. Bind both VF devices to the host driver:

```
[host]# echo <vf0_bdf> > /sys/bus/pci/drivers/virtio-pci/bind
[host]# echo <vf1_bdf> > /sys/bus/pci/drivers/virtio-pci/bind
```

⚠ The number of QPs must be greater than 0.

8.15.6.12.3.3 QP Limitations

- QP and MSIX configuration is mutually exclusive (i.e., only one of them can be configured at a time). For example, the following **modify** command should result in failure:

```
[dpu]# virtnet modify -p 0 -v 1 device -qp 2 -n 6
```

- To use a VF, assign it with a valid QP number:

```
[dpu]# virtnet modify -p 0 -v 1 device -n 4
```

  The minimum number of QP resources which allows the VF to load the host driver is 1.
- The QP resources of a VF can be reduced to 0. However, the VF would not be functional in this case.

```
[dpu]# virtnet modify -p 0 -v 1 device -qp 0
```

8.15.6.13 Virt Queue Types

Virt queues (VQs) are the mechanism for bulk data transport on virtio devices. Each device can have zero or more VQs.

VQs can be in one of the following modes:

- Split
- Packed

When changing the supported VQ types, make sure to unload the guest driver first so the device can modify the supported feature bits.

8.15.6.13.1 Split VQ

Currently the default VQ type. Split VQ format is the only format supported by version 1.0 of the virtio spec.

In split VQ mode, each VQ is separated into three parts:

- Descriptor table – occupies the descriptor area
- Available ring – occupies the driver area
- Used ring – occupies the device area

Each of these parts is physically-contiguous in guest memory. Split VQ has a very simple design, but its sparse memory usage puts pressure on CPU cache utilization and requires several PCIe transactions for each descriptor.

8.15.6.13.1.1 Configuration

The following shows how the output of the **virtnet list** command appears only when split VQ mode is enabled:

```
"supported_virt_queue_types": {
     "value": "0x1",
     "    0": "SPLIT"
   },
```

8.15.6.13.2 Packed VQ

Packed Virtqueue addresses the inherent limitations of the legacy Split VQ design by merging the three separate descriptor rings into a single, contiguous location within the virtual environment's guest memory.

This streamlined memory layout significantly reduces the number of PCIe transactions required and improves CPU cache utilization per descriptor access, leading to better overall network performance.

8.15.6.13.2.1 Prerequisites

Packed VQ is supported from kernel 5.0 onwards, specifically requiring the virtio-support-packed-ring commit within the guest operating system.

8.15.6.13.2.2 Configuration

Administrators control this feature using the **VIRTIO_F_RING_PACKED** (bit 34) feature flag.

- Default State: Disabled.
- Scope: Can be enabled on a per-device basis.

Refer to the "DOCA Virtio-net Service Guide#Enabling and Disabling Virtio-net Features" section for the exact **virtnet modify** command syntax.

8.15.6.13.2.3 Limitations

The following features are not currently supported when packed VQ is enabled:

- Mergeable Rx buffer
- Jumbo MTU
- UDP segmentation offload (USO)
- RSS hash report

8.15.6.14 Virtio-net Feature Bits

Per virtio spec, virtio the device negotiates with the virtio driver on the supported features when the driver probes the device. The final negotiated features are a subset of the features supported by the device.

From the controller's perspective, all feature bits can be supported by a device are populated by **virtnet list** . Each individual virtio-net device is able to choose the feature bits supported by itself.

The following is a list of the feature bits currently supported by controller:

- **VIRTIO_NET_F_CSUM**
- **VIRTIO_NET_F_GUEST_CSUM**
- **VIRTIO_NET_F_CTRL_GUEST_OFFLOADS**
- **VIRTIO_NET_F_MTU**
- **VIRTIO_NET_F_MAC**
- **VIRTIO_NET_F_HOST_TSO4**
- **VIRTIO_NET_F_HOST_TSO6**
- **VIRTIO_NET_F_MRG_RXBUF**
- **VIRTIO_NET_F_STATUS**
- **VIRTIO_NET_F_CTRL_VQ**
- **VIRTIO_NET_F_CTRL_RX**
- **VIRTIO_NET_F_CTRL_VLAN**
- **VIRTIO_NET_F_GUEST_ANNOUNCE**
- **VIRTIO_NET_F_MQ**

- `VIRTIO_NET_F_CTRL_MAC_ADDR`
- `VIRTIO_F_VERSION_1`
- `VIRTIO_F_IOMMU_PLATFORM`
- `VIRTIO_F_RING_PACKED`
- `VIRTIO_F_ORDER_PLATFORM`
- `VIRTIO_F_SR_IOV`
- `VIRTIO_F_NOTIFICATION_DATA`
- `VIRTIO_F_RING_RESET`
- `VIRTIO_F_ADMIN_VQ`
- `VIRTIO_NET_F_HOST_USO`
- `VIRTIO_NET_F_HASH_REPORT`
- `VIRTIO_NET_F_GUEST_HDRLEN`
- `VIRTIO_NET_F_SPEED_DUPLEX`

For more information on these bits, refer to the *VIRTIO Version 1.2 Specifications*.

8.15.6.15 Virtio-net Event Notifications

Virtio-net Event Notifications provide real-time, asynchronous notifications of VF lifecycle and state changes from the virtio-net-controller on the DPU to external consumers, such as orchestrators, monitoring systems, and management agents.

Events are published as JSON messages over a NATS message broker. The design is best-effort with bounded queues, meaning the controller's critical data-path and Live Migration (LM) paths are never blocked by event delivery.

8.15.6.15.1 Supported Event Types

Event Type	When Emitted
`VF_CREATED`	VF device successfully opened
`VF_DESTROYED`	VF device closed/torn down
`VF_SUSPENDED`	VF suspended (LM quiesce)
`VF_RESUMED`	VF resumed (LM un-quiesce)
`VF_DRIVER_STATE_CHANGED`	VF driver state transition (de-duplicated)
`VF_LM_STATE_CHANGED`	LM state transition

8.15.6.15.2 Prerequisites

- The `virtio-net-controller` RPM package (provided by NVIDIA). NATS support is built-in by default; no additional build-time setup is required.

- A running NATS broker on the same DPU (localhost), listening on `127.0.0.1:4222`. The `nats-server` binary is not included in the BFB image and must be installed separately by the user.
- Tested versions:

Component	Version	Notes
nats-server	2.12.4	Broker binary (user-installed)
nats.c (C client)	3.12.0	Build-time dependency (pre-installed in BFB)

The feature relies only on basic NATS publish/subscribe functionality. Newer compatible versions of nats-server are expected to work but have not been validated.

> **Localhost Only**
>
> The NATS broker must run on the same DPU as the virtio-net-controller. Remote broker connections are not supported at this time. The event channel does not currently implement TLS encryption or authentication, so NVIDIA does not take responsibility for securing remote connections. Binding the broker to `127.0.0.1` ensures that event traffic stays local to the DPU.

8.15.6.15.2.1 Setting Up the NATS Broker

The NATS broker (`nats-server`) is a lightweight, standalone binary. It is not included in the BFB image and must be installed by the user. It must run on the DPU itself, bound to `127.0.0.1` (localhost only).

1. Install nats-server:
   - Option A: Package manager (recommended for production)

```
# Ubuntu / Debian
sudo apt-get install nats-server

# RHEL / Rocky / CentOS
sudo dnf install nats-server
```

   - Option B: Download prebuilt binary

```
NATS_VER=2.12.4
ARCH=linux-arm64   # or linux-amd64

curl -fL -o nats-server.tar.gz \
  "https://github.com/nats-io/nats-server/releases/download/v${NATS_VER}/nats-server-v${NATS_VER}-${ARCH}.tar.gz"

tar -xzf nats-server.tar.gz
sudo cp nats-server-v${NATS_VER}-${ARCH}/nats-server /usr/local/bin/
```

2. Start the broker:

```
nats-server -a 127.0.0.1 -p 4222 &
```

3. Verify the broker is running:

```
# Quick check -- NATS exposes a monitoring HTTP endpoint:
curl http://localhost:8222/varz 2>/dev/null | head -5

# Or simply:
nats-server --help  # Confirms the binary is installed
```

4. For production, run `nats-server` as a systemd service:

```
# /etc/systemd/system/nats-server.service
[Unit]
Description=NATS messaging server
After=network.target

[Service]
ExecStart=/usr/local/bin/nats-server -a 127.0.0.1 -p 4222
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
```

```
sudo systemctl daemon-reload
sudo systemctl enable --now nats-server
sudo systemctl status nats-server
```

8.15.6.15.3 NATS Subject Scheme

Events are published to NATS subjects with the following structure:

```
<subject_prefix>.<pf_index>.<vf_index>.<category>.<name>
```

8.15.6.15.3.1 Subject Mapping

Event Type	Subject Suffix
`VF_CREATED`	`lifecycle.created`
`VF_DESTROYED`	`lifecycle.destroyed`
`VF_SUSPENDED`	`lm.suspended`
`VF_RESUMED`	`lm.resumed`
`VF_DRIVER_STATE_CHANGED`	`driverstate.changed`
`VF_LM_STATE_CHANGED`	`lm.statechanged`

Example:

```
virtio.vf.0.1.lifecycle.created       # PF 0, VF 1 created
virtio.vf.0.3.lm.suspended           # PF 0, VF 3 suspended for LM
virtio.vf.1.0.driverstate.changed     # PF 1, VF 0 driver state change
```

8.15.6.15.3.2 Subscribing with Wildcards

NATS wildcard subjects allow flexible filtering:

```
virtio.vf.>              # All events, all VFs
virtio.vf.0.>            # All events for PF 0
virtio.vf.0.2.lifecycle.* # All lifecycle events for PF 0, VF 2
virtio.vf.*.*.lm.* # All LM events across all PFs/VFs
```

8.15.6.15.4 JSON Event Schema (v1)

Every event is published as a single JSON object. Schema version is `1`.

- Example: VF created

```
{
  "schema_version": 1,
  "type": "VF_CREATED",
  "timestamp_ns": "123456789012345",
  "vuid": "MT2333ABCDEF0123",
  "pf_index": 0,
  "vf_index": 1,
  "driver_state": "UNKNOWN"
}
```

- Example: VF suspended (LM)

```
{
  "schema_version": 1,
  "type": "VF_SUSPENDED",
  "timestamp_ns": "223456789012345",
  "vuid": "MT2333ABCDEF0123",
  "pf_index": 0,
  "vf_index": 1,
  "lm_state": "SUSPENDED",
  "driver_state": "DRIVER_OK"
}
```

- Example: Driver state changed

```
{
  "schema_version": 1,
  "type": "VF_DRIVER_STATE_CHANGED",
  "timestamp_ns": "323456789012345",
  "vuid": "MT2333ABCDEF0123",
  "pf_index": 0,
  "vf_index": 1,
  "driver_state": "DRIVER_OK"
}
```

Field reference:

Field	Type	Always Present	Description
`schema_version`	number	Yes	Always `1` for v1
`type`	string	Yes	Event type name
`timestamp_ns`	string	Yes	`CLOCK_MONOTONIC` nanoseconds (string to preserve uint64 precision)
`vuid`	string	Yes	VF unique identifier
`pf_index`	number	Yes	Physical function index
`vf_index`	number	Yes	Virtual function index
`lm_state`	string	No	Present only when relevant. Values: `SUSPENDED`, `RUNNING`
`driver_state`	string	Yes	Values: `UNKNOWN`, `RESET`, `ACKNOWLEDGE`, `DRIVER`, `FEATURES_OK`, `DRIVER_OK`, `DEVICE_NEEDS_RESET`, `FAILED`

8.15.6.15.5 Consuming Events

There are two ways to consume VF events from the NATS broker:

- Native NATS client – use any NATS client library (Go, Python, C, Java, etc.) to subscribe directly to the broker. This is the recommended approach for most integrations.
- vnet-event subscriber API (**libvnet_event**) – a C library provided with the virtio-net-controller that handles NATS transport, JSON decoding, bounded queuing, and delivers parsed **struct vnet_event** to a callback. Useful for C/C++ consumers that want structured event access.

8.15.6.15.5.1 Option A: Native NATS Client (any language)

Any standard NATS client library can subscribe to the event subjects. The consumer receives raw JSON and parses it according to the schema.

Python example (using the **nats-py** package):

```
import asyncio
import json
import nats

async def main():
    nc = await nats.connect("nats://127.0.0.1:4222")

    async def on_event(msg):
        event = json.loads(msg.data.decode())
        print(f"[{msg.subject}] type={event['type']} "
              f"vuid={event['vuid']} pf={event['pf_index']} vf={event['vf_index']} "
              f"driver_state={event.get('driver_state', 'N/A')} "
              f"lm_state={event.get('lm_state', 'N/A')}")

    # Subscribe to all VF events:
    await nc.subscribe("virtio.vf.>", cb=on_event)

    # Or subscribe to specific events:
    # await nc.subscribe("virtio.vf.0.*.lm.*", cb=on_event)

    # Run until interrupted
    try:
        await asyncio.Event().wait()
    except KeyboardInterrupt:
        pass
    finally:
        await nc.drain()

asyncio.run(main())
```

Go example (using the **nats.go** package):

```
package main

import (
    "encoding/json"
    "fmt"
    "log"
    "os"
    "os/signal"

    "github.com/nats-io/nats.go"
)

type VNetEvent struct {
    SchemaVersion int    `json:"schema_version"`
    Type          string `json:"type"`
    TimestampNs   string `json:"timestamp_ns"`
    VUID          string `json:"vuid"`
    PFIndex       int    `json:"pf_index"`
    VFIndex       int    `json:"vf_index"`
    LMState       string `json:"lm_state,omitempty"`
    DriverState   string `json:"driver_state"`
}

func main() {
    nc, err := nats.Connect("nats://127.0.0.1:4222")
    if err != nil {
        log.Fatal(err)
    }
    defer nc.Drain()
```

```
    nc.Subscribe("virtio.vf.>", func(msg *nats.Msg) {
        var ev VNetEvent
        if err := json.Unmarshal(msg.Data, &ev); err != nil {
            log.Printf("decode error: %v", err)
            return
        }
        fmt.Printf("[%s] type=%s vuid=%s pf=%d vf=%d driver_state=%s\n",
            msg.Subject, ev.Type, ev.VUID, ev.PFIndex, ev.VFIndex, ev.DriverState)
    })

    sig := make(chan os.Signal, 1)
    signal.Notify(sig, os.Interrupt)
    <-sig
}
```

8.15.6.15.5.2 Option B: Vnet-event Subscriber API (C Library)

The **libvnet_event** library provides a C subscriber API that handles NATS transport internally and delivers events via a callback. The library manages connection retry, bounded queuing, and optional JSON-to-struct parsing.

Installed paths:

- Library: **/usr/lib/libvnet_event.a** (or **/usr/lib64/**)
- Header: **/usr/include/vnet_event.h**
- Reference subscriber tool: **/usr/sbin/vnet_event_subscriber**

API lifecycle:

```
vnet_event_sub_create()   -- allocate handle, configure broker/filter/queue
        |
vnet_event_sub_start()    -- start worker + transport threads, register callback
        |
   (callback invoked for each received event)
        |
vnet_event_sub_destroy()  -- stop threads, free resources
```

Key types:

```
#include "vnet_event.h"

/* Opaque subscriber handle. */
typedef struct vnet_event_sub *vnet_event_sub_t;

/* Subscriber configuration. */
struct vnet_event_sub_cfg {
    char     broker_url[256];       /* NATS broker URL */
    char     subject_filter[128];   /* NATS subject filter */
    uint32_t connect_timeout_ms;    /* 0 => 2000; range 100..30000 */
    uint32_t reconnect_backoff_ms;  /* 0 => 1000; range 100..60000 */
    uint32_t max_queue_depth;       /* 0 => 4096; range 16..65536 */
    bool     deliver_parsed;        /* true: parse JSON into struct */
};

/* Callback signature. */
typedef void (*vnet_event_cb)(const struct vnet_event *ev,   /* parsed (or NULL) */
                              const char *json,              /* raw JSON */
                              size_t json_len,               /* JSON length */
                              void *cb_arg);                 /* user context */
```

C example – subscribe and print events:

```
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>

#include "vnet_event.h"

static volatile sig_atomic_t g_stop;

static void on_signal(int sig) { (void)sig; g_stop = 1; }

static void on_event(const struct vnet_event *ev,
                     const char *json, size_t json_len, void *arg)
{
    (void)arg;
```

```
    if (ev) {
        printf("type=%-24s vuid=%-20s pf=%u vf=%u driver_state=%u\n",
               ev->type == 1 ? "VF_CREATED" :
               ev->type == 2 ? "VF_DESTROYED" :
               ev->type == 3 ? "VF_SUSPENDED" :
               ev->type == 4 ? "VF_RESUMED" : "OTHER",
               ev->vuid, ev->pf_index, ev->vf_index, ev->driver_state);
    }

    /* Raw JSON is always available: */
    printf("  json(%zu): %.*s\n", json_len, (int)json_len, json);
}

int main(void)
{
    struct vnet_event_sub_cfg cfg = {};
    vnet_event_sub_t sub = NULL;
    int ret;

    signal(SIGINT, on_signal);
    signal(SIGTERM, on_signal);

    /* Configure the subscriber. */
    snprintf(cfg.broker_url, sizeof(cfg.broker_url),
             "nats://127.0.0.1:4222");
    snprintf(cfg.subject_filter, sizeof(cfg.subject_filter),
             "virtio.vf.>");
    cfg.max_queue_depth = 1024;
    cfg.deliver_parsed  = true;

    ret = vnet_event_sub_create(&cfg, &sub);
    if (ret) {
        fprintf(stderr, "sub_create failed: %d\n", ret);
        return 1;
    }

    ret = vnet_event_sub_start(sub, on_event, NULL);
    if (ret) {
        fprintf(stderr, "sub_start failed: %d\n", ret);
        vnet_event_sub_destroy(sub);
        return 1;
    }

    printf("Listening on %s filter='%s' ... Ctrl+C to stop\n",
           cfg.broker_url, cfg.subject_filter);

    while (!g_stop)
        sleep(1);

    /* Query subscriber health before shutdown. */
    {
        struct vnet_event_sub_stats st = {};

        if (vnet_event_sub_stats_get(sub, &st) == 0) {
            printf("stats: enq=%lu drop=%lu decode_fail=%lu"
                   " conn_fail=%lu sub_fail=%lu"
                   " nextmsg_fail=%lu reconnect=%lu"
                   " last_err=%d depth=%u queued=%u\n",
                   st.enqueued, st.dropped_queue_full,
                   st.decode_fail, st.connect_fail,
                   st.subscribe_fail, st.next_msg_fail,
                   st.reconnect_attempts, st.last_error,
                   st.max_queue_depth,
                   st.current_queue_count);
        }
    }

    vnet_event_sub_destroy(sub);
    return 0;
}
```

8.15.6.15.5.3 Subscriber Stats API

The subscriber exposes runtime health counters via **vnet_event_sub_stats_get()**. This function is thread-safe while the subscription is active.

Counter	What to Look For
dropped_queue_full increasing	Consumer callback is too slow, or queue depth is too small.
connect_fail increasing	Broker is unreachable. Check **broker_url** and broker status.
subscribe_fail > 0	Subject filter may be invalid, or broker rejected the subscription.
next_msg_fail > 0	Connection was lost after a successful connect.
reconnect_attempts growing	Transport is cycling through connect/backoff retries.

Counter	What to Look For
`current_queue_count` near `max_queue_depth`	Consumer is falling behind; drops are imminent.

8.15.6.15.5.4 Reference Subscriber Tool

The package includes a ready-to-use subscriber at `/usr/sbin/vnet_event_subscriber`.

```
vnet_event_subscriber [options]

Options:
  --broker-url URL         (default: nats://127.0.0.1:4222)
  --subject-filter FILTER  (default: virtio.vf.>)
  --parsed                 Print parsed event fields (default)
  --raw                    Print raw JSON
  --count N                Exit after N events (0 = run forever)
  --timeout-sec SEC        Exit after SEC seconds (0 = no timeout)
```

Examples:

```
# Watch all events with parsed output + raw JSON:
vnet_event_subscriber --parsed --raw

# Watch only LM events for PF 0:
vnet_event_subscriber --subject-filter 'virtio.vf.0.*.lm.*'

# Capture 10 events then exit:
vnet_event_subscriber --parsed --count 10
```

On exit, the tool prints subscriber stats (enqueued, drops, errors) to stderr.

8.15.6.15.6 Troubleshooting

8.15.6.15.6.1 Syslog Messages

All event subsystem messages are prefixed with `vnet_event:` in syslog.

Level	Message Pattern	Meaning
`INFO`	`publisher worker started`	Worker thread is running.
`INFO`	`publisher worker stopped`	Worker thread exited cleanly during shutdown.
`WARNING`	`queue full: dropping newest`	Queue overflow; consider increasing `max_queue_depth`.
`WARNING`	`transport publish failed`	NATS publish failed (broker unreachable). Worker will retry.
`WARNING`	`json encode failed`	Internal serialization error (should not happen).
`DEBUG`	`published subject=... json=...`	Per-event trace (only when `log_level >= 1`).

8.15.6.15.6.2 Common Scenarios

- `dropped_queue_full` increasing: The controller is generating events faster than the worker can publish. Possible causes include a slow or unreachable NATS broker (check `reconnect_attempts`), `max_queue_depth` being too small for the workload, or a standard burst of VF operations during mass hotplug.
- `transport_publish_fail` increasing: The NATS broker is unreachable or rejecting messages. Check if `nats-server` is running (`systemctl status nats-server` or `pgrep nats-server`), verify the `broker_url` is correct and reachable (`ping <broker-host>`), and review firewall rules on port 4222.
- `reconnect_attempts` growing steadily: The worker is repeatedly failing to connect. The backoff interval is `reconnect_backoff_ms` (default 1000ms). Verify broker availability and network path.
- No events received by subscriber:
  - Confirm `enabled` is `true` in `virtnet.conf` and the controller was restarted.
  - Check `virtnet debug vnet_event stats` to see if `enqueued` is incrementing.
  - Verify the subscriber's `subject_filter` matches the publisher's `subject_prefix`.
  - Confirm subscriber is connected to the same NATS broker as the publisher.

8.15.7 Virtio-net Service Guide Release Notes

The following subsections provide information on virtio-net service new features, interoperability, known issues, and bug fixes.

8.15.7.1 Changes and New Features in This Release

Title	Description
List Loaded Controller Configuration	The controller can now list the settings currently loaded from `virtnet.conf`.
Host Driver Bind and Unbind During Live Update	Host driver bind and unbind are now supported during live updates with `virtio-net update -s`. The controller pauses the emulator before the emulator process restarts to avoid interrupting in-flight commands. A recovery watchdog handles cases in which the command and health services do not resume. Use `-f` or `--force` to bypass the handshake and force an immediate update.

8.15.7.2 Bug Fixes

Ref #	Issue Details
5191525	Description: On the virtio-net DPA transmit path, packets batched into a single multi-packet work queue entry used the checksum-offload setting from the first packet. If a batch mixed packets that required hardware checksum offload with packets that the guest had already checksummed, packets that required hardware offload were transmitted with incomplete checksums.
	Detected in Version: 24.10.52.1

Ref #	Issue Details
5176674	Description: The event subscriber now stops retrying when a NATS broker URL is invalid and reports the configuration error.
	Detected in Version: v26.07.8
5172846	Description: The controller now validates recovered SF paths against the current SF number and recreates stale entries. This prevents VF SF initialization failures after switching the `sf_recovery_shm` path back to its default.
	Detected in Version: v26.07.8-2-g13041d79-dirty
5166167	Description: On Ubuntu 24.04, creating an SF port no longer fails with a `File exists` error (`EEXIST`) after using `sf_pool_force_destroy` .
	Detected in Version: v26.07.8
5155365	Description: VF MSI-X configuration now accepts odd values that match the firmware configuration.
	Detected in Version: v26.07.7-2-gf9cd9979-dirty
5150787	Description: Hotplug, unplug, and modify requests during a live update are now rejected with a retryable error until the update completes.
	Detected in Version: v26.07.6
5105457	Description: The `vfs_per_pf` configuration value now rejects values below the valid lower bound.
	Detected in Version: v26.07.3-1-g8ab0f88e-dirty
5017732	Description: Packed virtqueues now process complete descriptor chains, preventing transmit-ring state corruption and potential connectivity loss.
	Detected in Version: v26.04.5
4999933	Description: Drop-counter queries are now synchronized with concurrent enable and disable operations, preventing access to invalid counter state during configuration changes.
	Detected in Version: 26.04.5
4995698	Description: Running `virtnet list` while the controller restarts no longer causes a device-close timeout.
	Detected in Version: v26.04.4
4992806	Description: Dynamic Interrupt Moderation teardown now waits for DPA execution units to stop accessing its data structures, preventing a use-after-free that could crash the DPA application.
	Detected in Version: v26.04.4
4964304	Description: When the controller encounters an RX descriptor address of zero, it now reports the transmit error and rolls back the send queue state, preventing the DPA transmit process from hanging.
	Detected in Version: 26.04.1-1
4627124	Description: A virtio-net VF now enters transitional mode after its guest loads a legacy driver. Previously, the VF could remain in modern mode.
	Detected in Version: v25.10.1

8.15.7.3 Known Issues

The following are known limitations of this NVIDIA® BlueField® virtio-net software version.

Ref #	Issue Details
4849313	Description: Creating a hotplug device can show an outdated **sf_rep_net_device** name in the command output.
	Workaround: Run **virtnet query** to view the current **sf_rep_net_device** name.
	Keyword: Hotplug; **sf_rep_net_device**
	Reported in Version: 26.07
5013051	Description: Following a NATS broker restart, the virtio-net-controller relies on lazy reconnection. Consequently, the first event generated while disconnected is lost. However, this dropped event triggers a successful reconnection, allowing all subsequent events to be delivered normally.
	Workaround: N/A
	Keyword: LM; vnet_event
	Reported in version: 26.04
4898379	Description: Manually issuing inactive and active state commands (e.g., via devlink) on a SF is not supported while the virtio-net controller is active. Executing these state changes abruptly tears down the underlying firmware and hardware resources without properly notifying the controller. This causes an immediate loss of traffic for all connections associated with the SF, and the system cannot automatically recover from this state.
	Workaround: Avoid manually toggling SF states while the virtio-net controller is running. If an SF is inadvertently toggled and traffic drops, you must perform a full manual reinitialization of both the affected SF and the controller to restore network connectivity.
	Keyword: Scalable function; recovery
	Reported in version: 26.04
4961952	Description: If the virtio-net controller is restarted during an active vDPA live migration, transient RQT modify errors (e.g., "Remote I/O error") may appear in the system logs on the migration source VF. These error messages are strictly cosmetic and have no functional impact.
	Workaround: N/A
	Keyword: Live migration; vDPA
	Reported in version: 26.04
4914672	Description: During rapid, repeated virtual machine stress testing (e.g., executing **virsh destroy** followed by **virsh start**), Windows VMs may appear unresponsive or fail to answer network pings.
	Workaround: Increase the boot wait time in your automation scripts to a minimum of 200 seconds to provide the Windows VM sufficient time to fully complete its crash recovery, finish the boot process, and initialize its virtio-net interfaces before network connectivity is verified.
	Keyword: Windows VM; reboot
	Reported in version: 26.04

Ref #	Issue Details
4914672	Description: CentOS 7 virtual machines running older kernels (specifically kernel 3.10) may experience a transient soft lockup in `virtnet_send_command` when the `virtio-net` controller undergoes rapid, successive restarts. During this event, the system may report that the CPU is stuck for approximately 22 seconds. This behavior is isolated to older kernel versions; VMs running newer kernels, such as modern Ubuntu or Windows releases, are unaffected.
	Workaround: N/A
	Keyword: Kernel 3.10.0; CentOS 7; lockup
	Reported in version: 26.04
4797496	Description: Virtio-net does not support Packed Virtqueues (`"packed_vq": 1`) in the following scenarios: • It is not supported on any VF configured for live migration. • It is not supported on any PF that has the Admin Queue (AQ) enabled.
	Workaround: Explicitly disable Packed VQs by setting `"packed_vq": 0` in the `virtnet.conf` configuration file for any PFs using AQs or VFs intended for live migration. Use the default Split VQ (`split_vq`) mode instead.
	Keyword: VQ; live migration
	Reported in version: 25.10
4498529	Description: Windows VM may take a lot of time to load when VLAN tagging is enabled.
	Workaround: Disable VLAN tagging.
	Keyword: Windows; VLAN
	Reported in version: 25.10
4534273	Description: After installing a new version, `virtnet -v` or `--version` displays the version of the updated CLI, not the source or target upgrade versions.
	Workaround: Run `virtnet version` to view both the original and destination versions.
	Keyword: CLI; version; update
	Reported in version: 25.07
3879093	Description: When creating a large number of virtio-net VFs, the representor name of the SF may not be renamed.
	Workaround: Use the `ip` command to rename the representor manually.
	Keyword: Representor
	Reported in version: 24.10
3943905	Description: Host OS kernel <3.19 does not support 31 hotplug devices.
	Workaround: Avoid hotplugging more than 20 devices if host OS kernel is <3.19, or upgrade the kernel to ≥3.19.
	Keyword: Host OS; kernel; hotplug
	Reported in version: 24.07
4022160	Description: Feature bit `VIRTIO_NET_F_CTRL_VLAN` is not supported. Enabling it from the hotplug device may results in anomalous behavior.

Ref #	Issue Details
	Workaround: Disable `VIRTIO_NET_F_CTRL_VLAN` .
	Keyword: Feature bit
	Reported in version: 24.07
4001261	Description: The `virtnet.conf` file does not check invalid values such as negative numbers or 0.
	Workaround: N/A
	Keyword: Virtnet; config; invalid value
	Reported in version: 24.07
3965598	Description: Admin-VQ-based transitional VF show a `vf_get` error when the controller is restarted. However, VF functionality is not affected.
	Workaround: N/A
	Keyword: Admin VQ; transitional device
	Reported in version: 24.07
3961951	Description: Out-of-memory call trace occurs when creating many (>300) VFs on a BlueField running OpenEuler or CentOS 7.6.
	Workaround: Update the kernel to support shared RQ.
	Keyword: OOM; OpenEuler; CentOS 7.6; virtual function
	Reported in version: 24.07
3862683	Description: Creating VFs and hotplug PFs in parallel can lead to controller crash.
	Workaround: Create VFs followed by hotplug PF or vice versa.
	Keyword: Virtio-net emulation
	Reported in version: 1.9.0
3665070	Description: Virtio-net controller fails to load if `DPA_AUTHENTICATION` is enabled.
	Workaround: N/A
	Keywords: Virtio-net; DPA
	Reported in version: DOCA 2.5.0
3538486	Description: When removing LAG configuration from BlueField, a kernel warning for `uverbs_destroy_ufile_hw` is observed if virtio-net-controller is still running.
	Workaround: Stop virtio-net-controller service before cleaning up bond configuration.
	Keywords: Virtio-net; LAG
	Reported in version: DOCA 2.2.0
3683801	Description: Starting from kernel 5.14, the virtio-net TX path has a logic which may trigger infinite loop when `vq` is broken (e.g., device is removed) under heavy traffic.
	Workaround: N/A
	Keyword: Virtio-net
	Reported in version: DOCA 1.8.0

Ref #	Issue Details
3714522	Description: When creating/destroying VFs back to back, make sure the virtio-net controller side does not see any alive VF before recreating them from the guest OS (i.e., **virtnet query**).
	Workaround: N/A
	Keyword: Virtio-net; VFs
	Reported in version: DOCA 1.8.0
3694402	Description: When restarting the virtio-net-controller from the DPU while the guest OS is booting, the guest OS may see kernel call trace while the controller is preparing the device. It recovers once the controller starts.
	Workaround: N/A
	Keyword: Virtio-net; hotplug; restart
	Reported in version: DOCA 1.8.0
3633453	Description: Jumbo MTU is only supported on a guest OS with kernel 4.11 and above.
	Workaround: N/A
	Keyword: Virtio-net; jumbo MTU
	Reported in version: DOCA 1.7.0
3021967	Description: When rebooting a DPU with a large number of VFs created on host, VF recovery may fail due to timeout.
	Workaround: Restart the driver on the host after the DPU is up.
	Keyword: Reboot; VFs
	Reported in version: DOCA 1.7.0
3232444	Description: After live migration of virtio-net devices using the VFE driver, the **max_queues_size** output from the **virtnet list** may be wrong. This does not affect the actual value.
	Workaround: N/A
	Keywords: Virtio-net; live migration
	Reported in version: DOCA 1.4.0
2801780	Description: When running virtio-net-controller with host kernel older than 3.10.0-1160.el7, host virtio driver may get error (**Unexpected TXQ (13) queue failure: -28**) from dmesg in traffic stress test.
	Workaround: N/A
	Keywords: Virtio-net; error
	Reported in version: DOCA 1.2.0
2870213	Description: Servers do not recover after configuring **PCI_SWITCH_EMULATION_NUM_PORT** to 32 followed by power cycle.
	Workaround: Clear NVRAM and reset mlxconfig to default
	Keywords: Virtio-net; power cycle
	Reported in version: DOCA 1.2.0

Ref #	Issue Details
2685191	Description: Once virtio-net is enabled, the mlx5 Windows VF becomes unavailable.
	Workaround: N/A
	Keywords: Virtio-net; virtual function; WinOF-2
	Reported in version: DOCA 1.2.0
2702395	Description: When a device is hot-plugged from the virtio-net controller, the host OS may hang when warm reboot is performed on the host and Arm at the same time.
	Workaround: Reboot the host OS first and only then reboot DPU.
	Keywords: Virtio-net controller; hot-plug; reboot
	Reported in version: DOCA 1.2.0

8.16 OpenvSwitch Acceleration - OVS in DOCA

This page provides an overview of Open vSwitch (OVS) acceleration on NVIDIA® BlueField® DPUs, explaining how OVS tasks are offloaded to hardware (using OVS-Kernel, OVS-DPDK, or OVS-DOCA) to improve performance and network efficiency.

Note on naming conventions:

- OVS – Refers to the Open vSwitch distribution within DOCA framework
- OVS-DOCA – Describes the datapath offloading layer (DPIF) that utilizes the DOCA Flow library for offloading tasks. This layer is a component of OVS, along with additional DPIF implementations that facilitate offloading via DPDK or Kernel, known respectively as OVS-DPDK and OVS-Kernel.

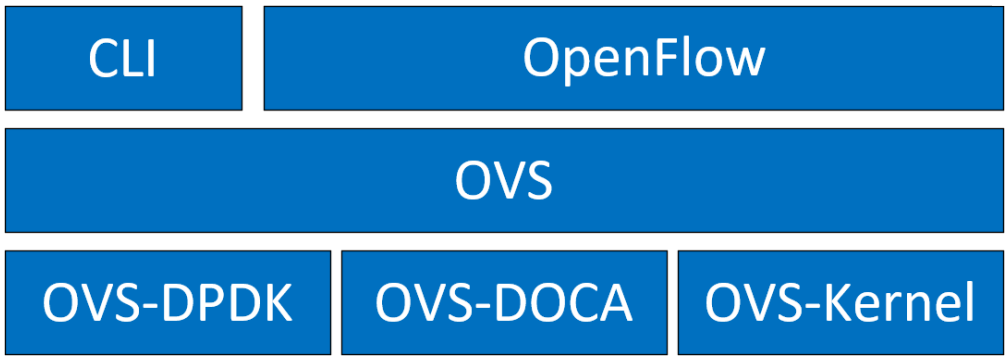
NVIDIA advises utilizing the OVS-DOCA DPIF to maximize efficiency, performance, scalability, and feature support.

The DPDK and Kernel DPIFs are maintained in their current form primarily for backward compatibility and are not planned to be updated with new features.

8.16.1 Introduction

Open vSwitch (OVS) is a software-based network technology that enhances virtual machine (VM) communication within internal and external networks. Typically deployed in the hypervisor, OVS employs a software-based approach for packet switching, which can strain CPU resources, impacting system performance and network bandwidth utilization. Addressing this, NVIDIA's Accelerated Switching and Packet Processing (ASAP^2) technology offloads OVS data-plane tasks to specialized hardware, like the embedded switch (eSwitch) within the NIC subsystem, while maintaining an unmodified OVS control-plane. This results in notably improved OVS performance without burdening the CPU.

NVIDIA's DOCA-OVS extends the traditional OVS-DPDK and OVS-Kernel data-path offload interfaces (DPIF), introducing OVS-DOCA as an additional DPIF implementation. DOCA-OVS, built upon NVIDIA's networking API, preserves the same interfaces as OVS-DPDK and OVS-Kernel while utilizing the DOCA Flow library with the additional OVS-DOCA DPIF. Unlike the use of the other DPIFs (DPDK, Kernel), OVS-DOCA DPIF exploits unique hardware offload mechanisms and application techniques, maximizing performance and features for NVIDA NICs and DPUs. This mode is especially efficient due to its architecture and DOCA library integration, enhancing e-switch configuration and accelerating hardware offloads beyond what the other modes can achieve.



NVIDIA OVS installation contains all three OVS flavors. The following subsections describe the three flavors (default is OVS-Kernel) and how to configure each of them.

8.16.2 OVS and Virtualized Devices

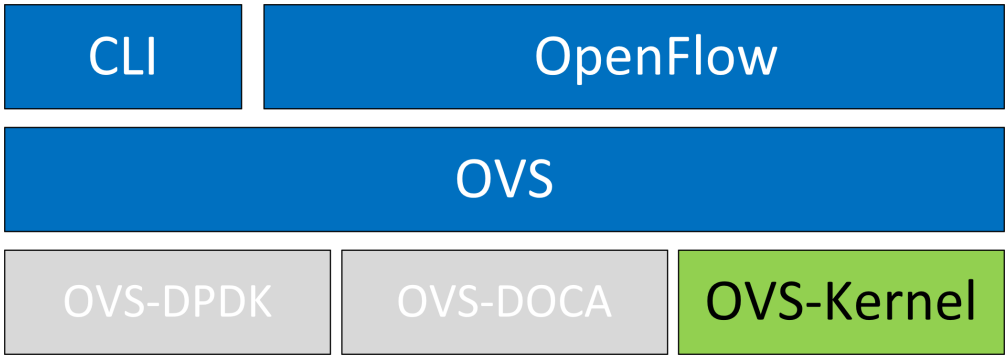
When OVS is combined with NICs and DPUs (such as NVIDIA® ConnectX®-6 Lx/Dx and NVIDIA® BlueField®-2 and later), it utilizes the hardware data plane of ASAP^2. This data plane can establish connections to VMs using either SR-IOV virtual functions (VFs) or virtual host data path acceleration (vDPA) with virtio.

In both scenarios, an accelerator engine within the NIC accelerates forwarding and offloads the OVS rules. This integrated solution accelerates both the infrastructure (via VFs through SR-IOV or virtio) and the data plane. For DPUs (which include a NIC subsystem), an alternate virtualization technology implements full virtio emulation within the DPU, enabling the host server to communicate with the DPU as a software virtio device.

- When using ASAP^2 data plane over SR-IOV virtual functions (VFs), the VF is directly passed through to the VM, with the NVIDIA driver running within the VM.
- When using vDPA, the vDPA driver allows VMs to establish their connections through VirtIO. As a result, the data plane is established between the SR-IOV VF and the standard virtio driver within the VM, while the control plane is managed on the host by the vDPA application.

8.16.3 OVS-Kernel Hardware Acceleration

OVS-Kernel is enabled by default on your NVIDIA device.



8.16.3.1 Switchdev Configuration

1. Make sure no VFs exists:

```
# echo 0 > /sys/class/net/enp4s0f0/device/sriov_numvfs
```

> VMs with attached VFs must be powered off to unbind the VFs.

2. Ensure that all VFs are unbound. Then change the eSwitch mode from **legacy** to **switchdev** on the PF device:

```
# devlink dev eswitch set pci/0000:3b:00.0 mode switchdev
```

This also creates the VF representor netdevices in the host OS.

> To return to SR-IOV legacy mode, run:
>
> ```
> # devlink dev eswitch set pci/0000:3b:00.0 mode legacy
> ```
>
> This also removes the VF representor netdevices.

For operating systems or kernels that do not support devlink, you can move to switchdev mode using sysfs:

```
# echo switchdev > /sys/class/net/enp4s0f0/compat/devlink/mode
```

3. Turn on SR-IOV on the PF device:

```
# echo 2 > /sys/class/net/enp4s0f0/device/sriov_numvfs
```

4. At this stage, VFs and VF representors have been created. To map a representor to its VF, obtain the representor's **switchid** and **portname**:

```
# ip -d link show eth4
41: enp0s8f0_1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group default
 qlen 1000
```

```
    link/ether ba:e6:21:37:bc:d4 brd ff:ff:ff:ff:ff:ff promiscuity 0 addrgenmode eui64 numtxqueues 10
numrxqueues 10 gso_max_size 65536 gso_max_segs 65535 portname pf0vf1 switchid f4ab580003a1420c
```

Where:

- **switchid** is used to map a representor to a device. Both device PFs have the same **switchid.**
- **portname** is used to map a representor to PF and VF. The returned value is **pf<X>vf<Y>**, where **X** is the PF number and **Y** is the VF number.

8.16.3.2 Switchdev Performance Tuning

Switchdev tuning improves its performance.

8.16.3.2.1 Steering Mode

OVS-kernel supports two steering modes for rule insertion into hardware:

- SMFS (software-managed flow steering) is the default mode; rules are inserted directly to the hardware by the software (driver). This mode is optimized for rule insertion.
- DMFS (device-managed flow steering) rule are inserted using firmware commands. This mode is optimized for throughput with a small number of rules in the system.

The steering mode can be configured via **sysfs** or **devlink** API in kernels that support it:

- For **sysfs**:

```
echo <smfs|dmfs> > /sys/class/net/<pf-netdev>/compat/devlink/steering_mode
```

- For **devlink**:

```
devlink dev param set pci/0000:00:08.0 name flow_steering_mode value "<smfs|dmfs>" cmode runtime
```

Notes:

- The mode should be set prior to moving to **switchdev**, by echoing to the **sysfs** or invoking the **devlink** command.
- Only when moving to **switchdev** will the driver use the mode configured.
- The mode cannot be changed after moving to **switchdev**.
- The steering mode is applicable for **switchdev** mode only (it does not affect legacy SR-IOV or other configurations).

8.16.3.2.2 Troubleshooting SMFS

mlx5 debugfs displays software steering resources. **dr_domain** including its tables, matchers, and rules. The interface is read-only.

> ⚠ New steering rules cannot be inserted or deleted while the dump is being created,

The steering information is dumped in CSV form in the following format: **<object_type>,<object_ID>, <object_info>,...,<object_info>**.

This data can be read at the following path: **/sys/kernel/debug/mlx5/<BDF>/steering/fdb/<domain_handle>**.

Example:

```
# cat /sys/kernel/debug/mlx5/0000:82:00.0/steering/fdb/dmn_000018644
3100,0x55caa4621c50,0xee802,4,65533
3101,0x55caa4621c50,0xe0100008
```

You can then use the steering dump parser to make the output human-readable. The parser can be found in this GitHub repository.

8.16.3.2.3 vPort Match Mode

OVS-kernel supports two modes that define how the rules match on vPort.

Mode	Description
Metadata	Rules match on metadata instead of vPort number (default mode). This mode is required to support SR-IOV live migration and dual-port RoCE. ⚠ Metadata matching might impact performance.
Legacy	Rules match on vPort number. Legacy matching does not affect performance to the extent that metadata matching does. It can be used only if SR-IOV live migration or dual port RoCE are enabled/used.

vPort match mode can be controlled via **sysfs**:

- Set legacy:

```
echo legacy > /sys/class/net/<PF netdev>/compat/devlink/vport_match_mode
```

- Set metadata:

```
echo metadata > /sys/class/net/<PF netdev>/compat/devlink/vport_match_mode
```

⚠ This mode must be set prior to moving to **switchdev**.

8.16.3.2.4 Flow Table Large Group Number

Offloaded flows, including connection tracking (CT), are added to the virtual switch forwarding data base (FDB) flow tables. FDB tables have a set of flow groups, where each flow group saves the same traffic pattern flows. For example, for CT offloaded flow, TCP and UDP are different traffic patterns which end up in two different flow groups.

A flow group is limited in size to save flow entries. By default, the driver has 15 big FDB flow groups. Each of these big flow groups can save 4M/(15+1)=256k different 5-tuple flow entries at

most. For scenarios with more than 15 traffic patterns, the driver provides a module parameter (**num_of_groups**) to allow customization and performance tuning.

The mode can be controlled via module param or devlink API for kernels that support it:

- Module param:

```
echo <num_of_groups> > /sys/module/mlx5_core/parameters/num_of_groups
```

- Devlink:

```
devlink dev param set pci/0000:82:00.0 name fdb_large_groups cmode driverinit value 20
```


The change takes effect immediately if the FDB table is empty (no traffic and all offloaded flows have aged out). You can switch this dynamically without reloading the driver. However, if there are still offloaded flows when you change this parameter, the change will take effect only after all flows have aged out.

8.16.3.3 Open vSwitch Configuration

OVS configuration is a simple OVS bridge configuration with **switchdev**.

1. Run the OVS service:

```
systemctl start openvswitch
```

2. Create an OVS bridge (named **ovs-sriov** here):

```
ovs-vsctl add-br ovs-sriov
```

3. Enable hardware offload (disabled by default):

```
ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
```

4. Restart the OVS service:

```
systemctl restart openvswitch
```

This step is required for hardware offload changes to take effect.

5. Add the PF and the VF representor netdevices as OVS ports:

```
ovs-vsctl add-port ovs-sriov enp4s0f0
ovs-vsctl add-port ovs-sriov enp4s0f0_0
ovs-vsctl add-port ovs-sriov enp4s0f0_1
```

Make sure to bring up the PF and representor netdevices:

```
ip link set dev enp4s0f0 up
ip link set dev enp4s0f0_0 up
ip link set dev enp4s0f0_1 up
```

The PF represents the uplink (wire):

```
# ovs-dpctl show
```

```
system@ovs-system:
        lookups: hit:0 missed:192 lost:1
        flows: 2
        masks: hit:384 total:2 hit/pkt:2.00
        port 0: ovs-system (internal)
        port 1: ovs-sriov (internal)
        port 2: enp4s0f0
        port 3: enp4s0f0_0
        port 4: enp4s0f0_1
```

6. Run traffic from the VFs and observe the rules added to the OVS data-path:

```
# ovs-dpctl dump-flows

recirc_id(0),in_port(3),eth(src=e4:11:22:33:44:50,dst=e4:1d:2d:a5:f3:9d),
eth_type(0x0800),ipv4(frag=no), packets:33, bytes:3234, used:1.196s, actions:2

recirc_id(0),in_port(2),eth(src=e4:1d:2d:a5:f3:9d,dst=e4:11:22:33:44:50),
eth_type(0x0800),ipv4(frag=no), packets:34, bytes:3332, used:1.196s, actions:3
```

In this example, the ping is initiated from VF0 (OVS port 3) to the outer node (OVS port 2), where the VF MAC is **e4:11:22:33:44:50** and the outer node MAC is **e4:1d:2d:a5:f3:9d**. As previously shown, two OVS rules are added, one in each direction.

Users can also verify offloaded packets by adding **type=offloaded** to the command. For example:

```
ovs-appctl dpctl/dump-flows type=offloaded
```

8.16.3.4 OVS Performance Tuning

8.16.3.4.1 Flow Aging

The aging timeout of OVS is displayed in milliseconds and can be controlled by running:

```
ovs-vsctl set Open_vSwitch . other_config:max-idle=30000
```

8.16.3.4.2 TC Policy

TC policy specifies the policy used with hardware offloading:

- **none** – adds a TC rule to both the software and the hardware (default)
- **skip_sw** – adds a TC rule only to the hardware
- **skip_hw** – adds a TC rule only to the software

Example:

```
ovs-vsctl set Open_vSwitch . other_config:tc-policy=skip_sw
```


TC policy should only be used for debugging purposes.

8.16.3.4.3 max-revalidator

Specifies the maximum time (in milliseconds) for the revalidator threads to wait for kernel statistics before executing flow revalidation.

```
ovs-vsctl set Open_vSwitch . other_config:max-revalidator=10000
```

8.16.3.4.4 n-handler-threads

Specifies the number of threads for software datapaths to use to handle new flows.

```
ovs-vsctl set Open_vSwitch . other_config:n-handler-threads=4
```

The default value is the number of online CPU cores minus the number of revalidators.

8.16.3.4.5 n-revalidator-threads

Specifies the number of threads for software datapaths to use to revalidate flows in the datapath.

```
ovs-vsctl set Open_vSwitch . other_config:n-revalidator-threads=4
```

8.16.3.4.5.1 vlan-limit

Limits the number of VLAN headers that can be matched to the specified number.

```
ovs-vsctl set Open_vSwitch . other_config:vlan-limit=2
```

8.16.3.5 Basic TC Rules Configuration

Offloading rules can also be added directly---not only through OVS---using the **tc** utility.

To create an offloading rule using TC:

1. Create an ingress **qdisc** (queueing discipline) for each interface that you wish to add rules into:

   ```
   tc qdisc add dev enp4s0f0 ingress
   tc qdisc add dev enp4s0f0_0 ingress
   tc qdisc add dev enp4s0f0_1 ingress
   ```

2. Add TC rules using flower classifier in the following format:

   ```
   tc filter add dev NETDEVICE ingress protocol PROTOCOL prio PRIORITY [chain CHAIN] flower [MATCH_LIST]
   [action ACTION_SPEC]
   ```

   ⚠ A list of supported matches (specifications) and actions can be found in Classification Fields (Matches).

3. Dump the existing `tc` rules using flower classifier in the following format:

```
tc [-s] filter show dev NETDEVICE ingress
```

8.16.3.6 SR-IOV VF LAG

SR-IOV VF LAG allows the NIC's physical functions (PFs) to receive the rules that the OVS attempts to offload to the bond net-device, and then offloads them to the hardware e-switch.

The supported bond modes are as follows:

- Active-backup
- XOR
- LACP

SR-IOV VF LAG enables complete offload of the LAG functionality to the hardware. The bonding creates a single bonded PF port. Packets from the up-link can arrive from any of the physical ports and are forwarded to the bond device.

When hardware offload is used, packets from both ports can be forwarded to any of the VFs. Traffic from the VF can be forwarded to both ports according to the bonding state. This means that when in active-backup mode, only one PF is up, and traffic from any VF goes through this PF. When in XOR or LACP mode, if both PFs are up, traffic from any VF is split between these two PFs.

8.16.3.6.1 SR-IOV VF LAG Configuration on ASAP^2

To enable SR-IOV VF LAG, both physical functions of the NIC must first be configured to SR-IOV `switchdev` mode, and only afterwards bond the up-link representors.

The following example shows the creation of a bond interface over two PFs:

1. Switch the eSwitch mode from Legacy to SwitchDev on the PF devices.:

```
# devlink dev eswitch set pci/0000:3b:00.0 mode switchdev
# devlink dev eswitch set pci/0000:3b:00.1 mode switchdev
```

2. Load the bonding device and attach the uplink representor (currently the PF) network devices as as members of the bond.

```
# modprobe bonding mode=802.3ad
# Ifup bond0 (make sure ifcfg file is present with desired bond configuration)
# ip link set enp4s0f0 master bond0
# ip link set enp4s0f1 master bond0
```

3. Create the VFs:

```
# echo 2 > /sys/class/net/enp4s0f0/device/sriov_numvfs
```

4. Add the representor network devices as OVS ports. If tunneling is not used, include the bond device as well:

```
# ovs-vsctl add-port ovs-sriov bond0
# ovs-vsctl add-port ovs-sriov enp4s0f0_0
# ovs-vsctl add-port ovs-sriov enp4s0f1_0
```

5. Ensure that both the PF and the representor network devices are brought up:

```
# ip link set dev bond0 up
# ip link set dev enp4s0f0_0 up
# ip link set dev enp4s0f1_0 up
```

8.16.3.6.2 Scalable/Sub-Function (SF) LAG Configuration on ASAP^2

1. Switch the eSwitch mode from Legacy to SwitchDev on the PF devices.:

```
# devlink dev eswitch set pci/0000:3b:00.0 mode switchdev
# devlink dev eswitch set pci/0000:3b:00.1 mode switchdev
```

2. Load the bonding device and attach the uplink representor (currently the PF) network devices as as members of the bond.

```
# modprobe bonding mode=802.3ad
# Ifup bond0 (make sure ifcfg file is present with desired bond configuration)
# ip link set enp4s0f0 master bond0
# ip link set enp4s0f1 master bond0
```

3. Create the VFs:

```
# mlxdevm port add pci/0000:3b:00.0 flavour pcisf pfnum 0 sfnum 1000
# mlxdevm port function set pci/0000:3b:00.0/32768 state active
```

4. Add the representor network devices as OVS ports. If tunneling is not used, include the bond device as well:

```
# ovs-vsctl add-port ovs-sriov bond0
# ovs-vsctl add-port ovs-sriov en8f0pf0sf1000
```

5. Ensure that both the PF and the representor network devices are brought up:

```
# ip link set dev bond0 up
# ip link set dev en8f0pf0sf1000 up
```

For more information about SFs, refer to the BlueField Scalable Function User Guide or the scalable functions GitHub wiki.

8.16.3.6.3 Using TC with VF LAG

Both rules can be added either with or without shared block:

- With shared block (supported from kernel 4.16 and RHEL/CentOS 7.7 and above):

```
tc qdisc add dev bond0 ingress_block 22 ingress
tc qdisc add dev ens4p0 ingress_block 22 ingress
tc qdisc add dev ens4p1 ingress_block 22 ingress
```

a. Add drop rule:

```
# tc filter add block 22 protocol arp parent ffff: prio 3 \
    flower \
        dst_mac e4:11:22:11:4a:51 \
        action drop
```

b. Add redirect rule from bond to representor:

```
# tc filter add block 22 protocol arp parent ffff: prio 3 \
        flower \
        dst_mac e4:11:22:11:4a:50 \
```

```
        action mirred egress redirect dev ens4f0_0
```

c. Add redirect rule from representor to bond:

```
# tc filter add dev ens4f0_0 protocol arp parent ffff: prio 3 \
    flower \
        dst_mac ec:0d:9a:8a:28:42 \
        action mirred egress redirect dev bond0
```

- Without shared block (supported from kernel 4.15 and below):

a. Add redirect rule from bond to representor:

```
# tc filter add dev bond0 protocol arp parent ffff: prio 1 \
        flower \
        dst_mac e4:11:22:11:4a:50 \
        action mirred egress redirect dev ens4f0_0
```

b. Add redirect rule from representor to bond:

```
# tc filter add dev ens4f0_0 protocol arp parent ffff: prio 3 \
        flower \
        dst_mac ec:0d:9a:8a:28:42 \
        action mirred egress redirect dev bond0
```

8.16.3.7 Classification Fields (Matches)

OVS-Kernel supports multiple classification fields which packets can fully or partially match.

8.16.3.7.1 Ethernet Layer 2

- Destination MAC
- Source MAC
- Ethertype

Supported on all kernels.

In OVS dump flows:

```
skb_priority(0/0),skb_mark(0/0),in_port(eth6),eth(src=00:02:10:40:10:0d,dst=68:54:ed:00:af:de),eth_type(0x8100),
packets:1981, bytes:206024, used:0.440s, dp:tc, actions:eth7
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol arp pref 1 \
flower \
dst_mac e4:1d:2d:5d:25:35 \
src_mac e4:1d:2d:5d:25:34 \
action mirred egress redirect dev $NIC
```

8.16.3.7.2 IPv4/IPv6

- Source address
- Destination address
- Protocol
    - TCP/UDP/ICMP/ICMPv6
- TOS
- TTL (HLIMIT)

Supported on all kernels.

In OVS dump flows:

```
Ipv4:
ipv4(src=0.0.0.0/0.0.0.0,dst=0.0.0.0/0.0.0.0,proto=17,tos=0/0,ttl=0/0,frag=no)
Ipv6:
ipv6(src=::/::,dst=1:1:1::3:1040:1008,label=0/0,proto=58,tclass=0/0x3,hlimit=64),
```

Using TC rules:

```
IPv4:
tc filter add dev $rep parent ffff: protocol ip pref 1 \
flower \
dst_ip 1.1.1.1 \
src_ip 1.1.1.2 \
ip_proto TCP \
ip_tos 0x3 \
ip_ttl 63 \
action mirred egress redirect dev $NIC


IPv6:
tc filter add dev $rep parent ffff: protocol ipv6 pref 1 \
flower \
dst_ip 1:1:1::3:1040:1009 \
src_ip 1:1:1::3:1040:1008 \
ip_proto TCP \
ip_tos 0x3 \
ip_ttl 63\
action mirred egress redirect dev $NIC
```

8.16.3.7.3 TCP/UDP Source and Destination Ports and TCP Flags

- TCP/UDP source and destinations ports
- TCP flags

Supported on kernel >4.13 and RHEL >7.5.

In OVS dump flows:

```
TCP: tcp(src=0/0,dst=32768/0x8000),
UDP: udp(src=0/0,dst=32768/0x8000),
TCP flags: tcp_flags(0/0)
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol ip pref 1 \
flower \
ip_proto TCP \
dst_port 100 \
src_port 500 \
tcp_flags 0x4/0x7 \
action mirred egress redirect dev $NIC
```

8.16.3.7.4 VLAN

- ID
- Priority
- Inner vlan ID and Priority

Supported kernels: All (QinQ: kernel 4.19 and higher, and RHEL 7.7 and higher).

In OVS dump flows:

```
eth_type(0x8100),vlan(vid=2347,pcp=0),
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol 802.1Q pref 1 \
                    flower \
                    vlan_ethtype 0x800 \
                    vlan_id 100 \
                    vlan_prio 0 \
                    action mirred egress redirect dev $NIC
QinQ:
tc filter add dev $rep parent ffff: protocol 802.1Q pref 1 \
                    flower \
                    vlan_ethtype 0x8100  \
                    vlan_id 100 \
                    vlan_prio 0 \
                    cvlan_id 20 \
                    cvlan_prio 0 \
                    cvlan_ethtype 0x800 \
                    action mirred egress redirect dev $NIC
```

8.16.3.7.5 Tunnel

- ID (Key)
- Source IP address
- Destination IP address
- Destination port
- TOS (supported from kernel 4.19 and above & RHEL 7.7 and above)
- TTL (support from kernel 4.19 and above & RHEL 7.7 and above)
- Tunnel options (Geneve)

Supported kernels:

- VXLAN: All
- GRE: Kernel >5.0, RHEL 7.7 and above
- Geneve: Kernel >5.0, RHEL 7.7 and above

In OVS dump flows:

```
tunnel(tun_id=0x5,src=121.9.1.1,dst=131.10.1.1,ttl=0/0,tp_dst=4789,flags(+key))
```

Using TC rules:

```
# tc filter add dev $rep protocol 802.1Q parent ffff: pref 1
flower \
vlan_ethtype 0x800 \
vlan_id 100 \
vlan_prio 0 \
action mirred egress redirect dev $NIC
QinQ:
# tc filter add dev vxlan100 protocol ip parent ffff: \
                flower \
                         skip_sw \
                         dst_mac e4:11:22:11:4a:51 \
                         src_mac e4+:11:22:11:4a:50 \
                         enc_src_ip 20.1.11.1 \
                         enc_dst_ip 20.1.12.1 \
                         enc_key_id 100 \
                         enc_dst_port 4789 \
                         action tunnel_key unset \
                         action mirred egress redirect dev ens4f0_0
```

8.16.3.8 Supported Actions

8.16.3.8.1 Forward

Forward action allows for packet redirection:

- From VF to wire

- Wire to VF
- VF to VF

Supported on all kernels.

In OVS dump flows:

```
skb_priority(0/0),skb_mark(0/0),in_port(eth6),eth(src=00:02:10:40:10:0d,dst=68:54:ed:00:af:de),eth_type(0x8100),
packets:1981, bytes:206024, used:0.440s, dp:tc, actions:eth7
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol arp pref 1 \
                        flower \
                        dst_mac e4:1d:2d:5d:25:35 \
                        src_mac e4:1d:2d:5d:25:34 \
                        action mirred egress redirect dev $NIC
```

8.16.3.8.2 Drop

Drop action allows to drop incoming packets.

Supported on all kernels.

In OVS dump flows:

```
skb_priority(0/0),skb_mark(0/0),in_port(eth6),eth(src=00:02:10:40:10:0d,dst=68:54:ed:00:af:de),eth_type(0x8100),
packets:1981, bytes:206024, used:0.440s, dp:tc, actions:drop
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol arp pref 1 \
                           flower \
                           dst_mac e4:1d:2d:5d:25:35 \
                           src_mac e4:1d:2d:5d:25:34 \
                           action drop
```

8.16.3.8.3 Statistics

By default, each flow collects the following statistics:

- Packets – number of packets which hit the flow
- Bytes – total number of bytes which hit the flow
- Last used – the amount of time passed since last packet hit the flow

Supported on all kernels.

In OVS dump flows:

```
skb_priority(0/0),skb_mark(0/0),in_port(eth6),eth(src=00:02:10:40:10:0d,dst=68:54:ed:00:af:de),eth_type(0x8100),
packets:1981, bytes:206024, used:0.440s, dp:tc, actions:drop
```

Using TC rules:

```
#tc -s filter show dev $rep ingress

filter protocol ip pref 2 flower chain 0
filter protocol ip pref 2 flower chain 0 handle 0x2
eth_type ipv4
ip_proto tcp
```

```
src_ip 192.168.140.100
src_port 80
skip_sw
in_hw
    action order 1: mirred (Egress Redirect to device p0v11_r) stolen
    index 34 ref 1 bind 1 installed 144 sec used 0 sec
    Action statistics:
    Sent 388344 bytes 2942 pkt (dropped 0, overlimits 0 requeues 0)
    backlog 0b 0p requeues 0
```

8.16.3.8.4 Tunnels: Encapsulation/Decapsulation

OVS-kernel supports offload of tunnels using encapsulation and decapsulation actions.

- Encapsulation – pushing of tunnel header is supported on Tx
- Decapsulation – popping of tunnel header is supported on Rx

Supported Tunnels:

- VXLAN (IPv4/IPv6) – supported on all Kernels
- GRE (IPv4/IPv6) – supported on kernel 5.0 and above & RHEL 7.6 and above
- Geneve (IPv4/IPv6) – supported on kernel 5.0 and above & RHEL 7.6 and above

OVS configuration:

In case of offloading tunnel, the PF/bond should not be added as a port in the OVS datapath. It should rather be assigned with the IP address to be used for encapsulation.

The following example shows two hosts (PFs) with IPs 1.1.1.177 and 1.1.1.75, where the PF device on both hosts is enp4s0f0, and the VXLAN tunnel is set with VNID 98:

- On the first host:

```
# ip addr add 1.1.1.177/24 dev enp4s0f1
# ovs-vsctl add-port ovs-sriov vxlan0 -- set interface vxlan0 type=vxlan options:local_ip=1.1.1.177
 options:remote_ip=1.1.1.75 options:key=98
```

- On the second host:

```
# ip addr add 1.1.1.75/24 dev enp4s0f1
# ovs-vsctl add-port ovs-sriov vxlan0 -- set interface vxlan0 type=vxlan options:local_ip=1.1.1.75
 options:remote_ip=1.1.1.177 options:key=98
```

For a GRE IPv4 tunnel, use **type=gre**. For a GRE IPv6 tunnel, use **type=ip6gre**. For a Geneve tunnel, use **type=geneve**.

When encapsulating guest traffic, the VF's device MTU must be reduced to allow the host/hardware to add the encap headers without fragmenting the resulted packet. As such, the VF's MTU must be lowered by 50 bytes from the uplink MTU for IPv4 and 70 bytes for IPv6.

Tunnel offload using TC rules:

```
Encapsulation:
# tc filter add dev ens4f0_0 protocol 0x806 parent ffff: \
                flower \
                        skip_sw \
                        dst_mac e4:11:22:11:4a:51 \
                        src_mac e4:11:22:11:4a:50 \
                action tunnel_key set \
```

```
                src_ip 20.1.12.1 \
                dst_ip 20.1.11.1 \
                id 100 \
                action mirred egress redirect dev vxlan100

Decapsulation:
# tc filter add dev vxlan100 protocol 0x806 parent ffff: \
                flower \
                        skip_sw \
                        dst_mac e4:11:22:11:4a:51 \
                        src_mac e4:11:22:11:4a:50 \
                        enc_src_ip 20.1.11.1 \
                        enc_dst_ip 20.1.12.1 \
                        enc_key_id 100 \
                        enc_dst_port 4789 \
                action tunnel_key unset \
                action mirred egress redirect dev ens4f0_0
```

8.16.3.8.5 VLAN Push/Pop

OVS-kernel supports offload of VLAN header push/pop actions:

- Push – pushing of VLAN header is supported on Tx
- Pop – popping of tunnel header is supported on Rx

8.16.3.8.5.1 OVS Configuration

Add a tag=$TAG section for the OVS command line that adds the representor ports. For example, VLAN ID 52 is being used here.

```
# ovs-vsctl add-port ovs-sriov enp4s0f0
# ovs-vsctl add-port ovs-sriov enp4s0f0_0 tag=52
# ovs-vsctl add-port ovs-sriov enp4s0f0_1 tag=52
```

The PF port should not have a VLAN attached. This will cause OVS to add VLAN push/pop actions when managing traffic for these VFs.

Dump Flow Example

```
recirc_id(0),in_port(3),eth(src=e4:11:22:33:44:50,dst=00:02:c9:e9:bb:b2),eth_type(0x0800),ipv4(frag=no), \
packets:0, bytes:0, used:never, actions:push_vlan(vid=52,pcp=0),2

recirc_id(0),in_port(2),eth(src=00:02:c9:e9:bb:b2,dst=e4:11:22:33:44:50),eth_type(0x8100), \
vlan(vid=52,pcp=0),encap(eth_type(0x0800),ipv4(frag=no)), packets:0, bytes:0, used:never, actions:pop_vlan,3
```

VLAN Offload Using TC Rules Example

```
# tc filter add dev ens4f0_0 protocol ip parent ffff: \
                flower \
                        skip_sw \
                        dst_mac e4:11:22:11:4a:51 \
                        src_mac e4:11:22:11:4a:50 \
                action vlan push id 100 \
                action mirred egress redirect dev ens4f0

# tc filter add dev ens4f0 protocol 802.1Q parent ffff: \
                flower \
                        skip_sw \
                        dst_mac e4:11:22:11:4a:51 \
                        src_mac e4:11:22:11:4a:50 \
                        vlan_ethtype 0x800 \
                        vlan_id 100 \
                        vlan_prio 0 \
                action vlan pop \
                action mirred egress redirect dev ens4f0_0
```

8.16.3.8.5.2 TC Configuration

Example of VLAN Offloading with popping header on Tx and pushing on Rx using TC rules:

```
# tc filter add dev ens4f0_0 ingress protocol 802.1Q parent ffff: \
```

```
        flower \
                vlan_id 100 \
        action vlan pop \
        action tunnel_key set \
                src_ip 4.4.4.1 \
                dst_ip 4.4.4.2 \
                dst_port 4789 \
                id 42 \
        action mirred egress redirect dev vxlan0

# tc filter add dev vxlan0 ingress protocol all parent ffff: \
        flower \
                enc_dst_ip 4.4.4.1 \
                enc_src_ip 4.4.4.2 \
                enc_dst_port 4789 \
                enc_key_id 42 \
        action tunnel_key unset \
        action vlan push id 100 \
        action mirred egress redirect dev ens4f0_0
```

8.16.3.8.6 Header Rewrite

This action allows for modifying packet fields.

8.16.3.8.7 Ethernet Layer 2

- Destination MAC
- Source MAC

Supported kernels:

- Kernel 4.14 and above
- RHEL 7.5 and above

In OVS dump flows:

```
skb_priority(0/0),skb_mark(0/0),in_port(eth6),eth(src=00:02:10:40:10:0d,dst=68:54:ed:00:af:de),eth_type(0x8100),
packets:1981, bytes:206024, used:0.440s, dp:tc, actions: set(eth(src=68:54:ed:00:f4:ab,dst=fa:16:3e:dd:69:c4)),eth7
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol arp pref 1 \
                            flower \
                            dst_mac e4:1d:2d:5d:25:35 \
                            src_mac e4:1d:2d:5d:25:34 \
                    action pedit ex \
                    munge eth dst set 20:22:33:44:55:66 \
                    munge eth src set aa:ba:cc:dd:ee:fe \
                    action mirred egress redirect dev $NIC
```

8.16.3.8.8 IPv4/IPv6

- Source address
- Destination address
- Protocol
- TOS
- TTL (HLIMIT)

Supported kernels:

- Kernel 4.14 and above
- RHEL 7.5 and above

In OVS dump flows:

```
Ipv4:
    set(eth(src=de:e8:ef:27:5e:45,dst=00:00:01:01:01:01)),
    set(ipv4(src=10.10.0.111,dst=10.20.0.122,ttl=63))
Ipv6:
    set(ipv6(dst=2001:1:6::92eb:fcbe:f1c8,hlimit=63)),
```

Using TC rules:

```
IPv4:
tc filter add dev $rep parent ffff: protocol ip pref 1 \
                            flower \
                            dst_ip 1.1.1.1 \
                            src_ip 1.1.1.2 \
                            ip_proto TCP \
                            ip_tos 0x3 \
                            ip_ttl 63 \
                    pedit ex \
                    munge ip src set 2.2.2.1 \
                    munge ip dst set 2.2.2.2 \
                    munge ip tos set 0 \
                    munge ip ttl dec \
                    action mirred egress redirect dev $NIC


IPv6:
tc filter add dev $rep parent ffff: protocol ipv6 pref 1 \
                            flower \
                            dst_ip 1:1:1::3:1040:1009 \
                            src_ip 1:1:1::3:1040:1008 \
                            ip_proto tcp \
                            ip_tos 0x3 \
                            ip_ttl 63\
                    pedit ex \
                    munge ipv6 src set 2:2:2::3:1040:1009 \
                    munge ipv6 dst set 2:2:2::3:1040:1008 \
                    munge ipv6 hlimit dec \
                    action mirred egress redirect dev $NIC
```

⚠ IPv4 and IPv6 header rewrite is only supported with match on UDP/TCP/ICMP protocols.

8.16.3.8.8.1 TCP/UDP Source and Destination Ports

- TCP/UDP source and destinations ports

Supported kernels:

- Kernel 4.16 and above
- RHEL 7.6 and above

In OVS dump flows:

```
TCP:

        set(tcp(src= 32768/0xffff,dst=32768/0xffff)),
UDP:

        set(udp(src= 32768/0xffff,dst=32768/0xffff)),
```

Using TC rules:

```
TCP:

        tc filter add dev $rep parent ffff: protocol ip pref 1 \
                            flower \
                            dst_ip 1.1.1.1 \
                            src_ip 1.1.1.2 \
                            ip_proto tcp \
                            ip_tos 0x3 \
                            ip_ttl 63 \
                    pedit ex \
                    pedit ex munge ip tcp sport set 200
                    pedit ex munge ip tcp dport set 200
                    action mirred egress redirect dev $NIC

UDP:
        tc filter add dev $rep parent ffff: protocol ip pref 1 \
                            flower \
                            dst_ip 1.1.1.1 \
```

```
                        src_ip 1.1.1.2 \
                        ip_proto udp \
                        ip_tos 0x3 \
                        ip_ttl 63 \
                pedit ex \
                pedit ex munge ip udp sport set 200
                pedit ex munge ip udp dport set 200
                action mirred egress redirect dev $NIC
```

8.16.3.8.8.2 VLAN

- ID

Supported on all kernels.

In OVS dump flows:

```
Set(vlan(vid=2347,pcp=0/0)),
```

Using TC rules:

```
tc filter add dev $rep parent ffff: protocol 802.1Q pref 1 \
                    flower \
                    vlan_ethtype 0x800 \
                    vlan_id 100 \
                    vlan_prio 0 \
            action vlan modify id 11 pipe
            action mirred egress redirect dev $NIC
```

8.16.3.8.9 Connection Tracking

The TC connection tracking (CT) action performs CT lookup by sending the packet to netfilter conntrack module. Newly added connections may be associated, via the **ct commit** action, with a 32 bit mark, 128 bit label, and source/destination NAT values.

The following example allows ingress TCP traffic from the uplink representor to **vf1_rep**, while assuring that egress traffic from **vf1_rep** is only allowed on established connections. In addition, mark and source IP NAT is applied.

In OVS dump flows:

```
ct(zone=2,nat)
ct_state(+est+trk)
actions:ct(commit,zone=2,mark=0x4/0xffffffff,nat(src=5.5.5.5))
```

Using TC rules:

```
# tc filter add dev $uplink_rep ingress chain 0 prio 1 proto ip \
                    flower \
                    ip_proto tcp   \
                    ct_state -trk \
               action ct zone 2 nat pipe
               action goto chain 2
# tc filter add dev $uplink_rep ingress chain 2 prio 1 proto ip \
                     flower \
                     ct_state +trk+new \
               action ct zone 2 commit mark 0xbb nat src addr 5.5.5.7 pipe \
               action mirred egress redirect dev $vf1_rep
# tc filter add dev $uplink_rep ingress chain 2 prio 1 proto ip \
                    flower \
                    ct_zone 2 \
                    ct_mark 0xbb \
                    ct_state +trk+est \
                action mirred egress redirect dev $vf1_rep

// Setup filters on $vf1_rep, allowing only established connections of zone 2 through, and reverse nat (dst nat in
this case)

# tc filter add dev $vf1_rep ingress chain 0 prio 1 proto ip \
                     flower \
                     ip_proto tcp \
```

```
                    ct_state -trk \
                action ct zone 2 nat pipe \
                action goto chain 1
# tc filter add dev $vf1_rep ingress chain 1 prio 1 proto ip \
                    flower \
                    ct_zone 2 \
                    ct_mark 0xbb \
                    ct_state +trk+est \
                action mirred egress redirect dev eth0
```

8.16.3.8.9.1 CT Performance Tuning

- Max offloaded connections – specifies the limit on the number of offloaded connections.
  Example:

```
devlink dev param set pci/${pci_dev} name ct_max_offloaded_conns value $max cmode runtime
```

- Allow mixed NAT/non-NAT CT – allows offloading of the following scenario:

```
•   cookie=0x0, duration=21.843s, table=0, n_packets=4838718, n_bytes=241958846, ct_state=-
trk,ip,in_port=enp8s0f0 actions=ct(table=1,zone=2)
•   cookie=0x0, duration=21.823s, table=1, n_packets=15363, n_bytes=773526,
ct_state=+new+trk,ip,in_port=enp8s0f0 actions=ct(commit,zone=2,nat(dst=11.11.11.11)),output:"enp8s0f0_1" •
cookie=0x0, duration=21.806s, table=1, n_packets=4767594, n_bytes=238401190,
ct_state=+est+trk,ip,in_port=enp8s0f0 actions=ct(zone=2,nat),output:"enp8s0f0_1"
```

  Example:

```
echo enable > /sys/class/net/<device>/compat/devlink/ct_action_on_nat_conns
```

8.16.3.8.10 Forward to Chain (TC Only)

TC interface supports adding flows on different chains. Only chain 0 is accessed by default. Access to the other chains requires the `goto` action.

In this example, a flow is created on chain 1 without any match and redirect to wire. The second flow is created on chain 0 and match on source MAC and action `goto` chain 1.

This example simulates simple MAC spoofing:

```
#tc filter add dev $rep parent ffff: protocol all chain 1 pref 1 \
                    flower \
                action mirred egress redirect dev $NIC

#tc filter add dev $rep parent ffff: protocol all chain 1 pref 1 \
                    flower \
                    src_mac aa:bb:cc:aa:bb:cc \
                action goto chain 1
```

8.16.3.9 Port Mirroring: Flow-based VF Traffic Mirroring for ASAP²

Unlike para-virtual configurations, when the VM traffic is offloaded to hardware via SR-IOV VF, the host-side admin cannot snoop the traffic (e.g., for monitoring).

ASAP² uses the existing mirroring support in OVS and TC along with the enhancement to the offloading logic in the driver to allow mirroring the VF traffic to another VF.

The mirrored VF can be used to run traffic analyzer (e.g., tcpdump, wireshark, etc.) and observe the traffic of the VF being mirrored.

The following example shows the creation of port mirror on the following configuration:

```
# ovs-vsctl show
  09d8a574-9c39-465c-9f16-47d81c12f88a
      Bridge br-vxlan
                Port "enp4s0f0_1"
                   Interface "enp4s0f0_1"
                Port "vxlan0"
                    Interface "vxlan0"
                                type: vxlan
                                options: {key="100", remote_ip="192.168.1.14"}
                Port "enp4s0f0_0"
                    Interface "enp4s0f0_0"
                Port "enp4s0f0_2"
                    Interface "enp4s0f0_2"
                Port br-vxlan
                    Interface br-vxlan
                                type: internal
      ovs_version: "2.14.1"
```

- To set **enp4s0f0_0** as the mirror port and mirror all the traffic:

```
# ovs-vsctl -- --id=@p get port enp4s0f0_0 \
           -- --id=@m create mirror name=m0 select-all=true output-port=@p \
           -- set bridge br-vxlan mirrors=@m
```

- To set **enp4s0f0_0** as the mirror port, only mirror the traffic, and set **enp4s0f0_1** as the destination port:

```
# ovs-vsctl -- --id=@p1 get port enp4s0f0_0 \
           -- --id=@p2 get port enp4s0f0_1 \
           -- --id=@m create mirror name=m0 select-dst-port=@p2 output-port=@p1 \
           -- set bridge br-vxlan mirrors=@m
```

- To set **enp4s0f0_0** as the mirror port, only mirror the traffic, and set **enp4s0f0_1** as the source port:

```
# ovs-vsctl -- --id=@p1 get port enp4s0f0_0 \
           -- --id=@p2 get port enp4s0f0_1 \
           -- --id=@m create mirror name=m0 select-src-port=@p2 output-port=@p1 \
           -- set bridge br-vxlan mirrors=@m
```

- To set **enp4s0f0_0** as the mirror port and mirror all the traffic on **enp4s0f0_1** :

```
# ovs-vsctl -- --id=@p1 get port enp4s0f0_0 \
           -- --id=@p2 get port enp4s0f0_1 \
           -- --id=@m create mirror name=m0 select-dst-port=@p2 select-src-port=@p2 output-port=@p1 \
           -- set bridge br-vxlan mirrors=@m
```

To clear the mirror port:

```
ovs-vsctl clear bridge br-vxlan mirrors
```

Mirroring using TC:

- Mirror to VF:

```
tc filter add dev $rep parent ffff: protocol arp pref 1 \
                       flower \
                       dst_mac e4:1d:2d:5d:25:35 \
                       src_mac e4:1d:2d:5d:25:34 \
                       action mirred egress mirror dev $mirror_rep pipe \
                       action mirred egress redirect dev $NIC
```

- Mirror to tunnel:

```
tc filter add dev $rep parent ffff: protocol arp pref 1 \
                       flower \
                       dst_mac e4:1d:2d:5d:25:35 \
                       src_mac e4:1d:2d:5d:25:34 \
                 action tunnel_key set \
```

```
                src_ip 1.1.1.1 \
                dst_ip 1.1.1.2 \
                dst_port 4789 \
                id 768 \
                pipe \
                action mirred egress mirror dev vxlan100 pipe \
                action mirred egress redirect dev $NIC
```

8.16.3.10 Forward to Multiple Destinations

Forwarding to up 32 destinations (representors and tunnels) is supported using TC:

- Example 1: forwarding to 32 VFs:

```
tc filter add dev $NIC parent ffff: protocol arp pref 1 \
                        flower \
                        dst_mac e4:1d:2d:5d:25:35 \
                        src_mac e4:1d:2d:5d:25:34 \
                        action mirred egress mirror dev $rep0 pipe \
                        action mirred egress mirror dev $rep1 pipe \
...
                        action mirred egress mirror dev $rep30 pipe \
                        action mirred egress redirect dev $rep31
```

- Example 2: forwarding to 16 tunnels:

```
tc filter add dev $rep parent ffff: protocol arp pref 1 \
                        flower \
                        dst_mac e4:1d:2d:5d:25:35 \
                        src_mac e4:1d:2d:5d:25:34 \
                        action tunnel_key set src_ip $ip_src dst_ip $ip_dst \
                        dst_port 4789 id 0 nocsum \
                        pipe action mirred egress mirror dev vxlan0 pipe \
                        action tunnel_key set src_ip $ip_src dst_ip $ip_dst \
                        dst_port 4789 id 1 nocsum \
                        pipe action mirred egress mirror dev vxlan0 pipe \
                        ...
                        action tunnel_key set src_ip $ip_src dst_ip $ip_dst \
                        dst_port 4789 id 15 nocsum \
                        pipe action mirred egress redirect dev vxlan0
```

⚠ TC supports up to 32 actions.

⚠ If header rewrite is used, then all destinations should have the same header rewrite.

⚠ If VLAN push/pop is used, then all destinations should have the same VLAN ID and actions.

8.16.3.11 sFlow

sFlow allows for monitoring traffic sent between two VMs on the same host using an sFlow collector.

The following example assumes the environment is configured as described later.

```
# ovs-vsctl show
  09d8a574-9c39-465c-9f16-47d81c12f88a
      Bridge br-vxlan
                Port "enp4s0f0_1"
                  Interface "enp4s0f0_1"
                Port "vxlan0"
                  Interface "vxlan0"
                            type: vxlan
                            options: {key="100", remote_ip="192.168.1.14"}
                Port "enp4s0f0_0"
                  Interface "enp4s0f0_0"
                Port "enp4s0f0_2"
```

```
                Interface "enp4s0f0_2"
              Port br-vxlan
                Interface br-vxlan
                          type: internal
    ovs_version: "2.14.1"
```

To sample all traffic over the OVS bridge:

```
# ovs-vsctl -- --id=@sflow create sflow agent=\"$SFLOW_AGENT\" \
                                        target=\"$SFLOW_TARGET:$SFLOW_PORT\" \
                                        header=$SFLOW_HEADER \
                                        sampling=$SFLOW_SAMPLING polling=10 \
            -- set bridge br-vxlan sflow=@sflow
```

Parameter	Description
SFLOW_AGENT	Indicates that the sFlow agent should send traffic from **SFLOW_AGENT** 's IP address
SFLOW_TARGET	Remote IP address of the sFlow collector
SFLOW_HEADER	Size of packet header to sample (in bytes)
SFLOW_SAMPLING	Sample rate

To clear the sFlow configuration:

```
# ovs-vsctl clear bridge br-vxlan sflow
```

To list the sFlow configuration:

```
# ovs-vsctl list sflow
```

sFlow using TC:

```
Sample to VF
tc filter add dev $rep parent ffff: protocol arp pref 1 \
                        flower \
                        dst_mac e4:1d:2d:5d:25:35 \
                        src_mac e4:1d:2d:5d:25:34 \
                        action sample rate 10 group 5 trunc 96 \
                        action mirred egress redirect dev $NIC
```

A userspace application is needed to process the sampled packet from the kernel. An example is available on Github.

8.16.3.12 Rate Limit

OVS-kernel supports offload of VF rate limit using OVS configuration and TC.

The following example sets the rate limit to the VF related to representor **eth0** to 10Mb/s:

- OVS:

```
ovs-vsctl set interface eth0 ingress_policing_rate=10000
```

- TC:

```
tc_filter add dev eth0 root prio 1 protocol ip matchall skip_sw action police rate 10mbit burst 20k
```

8.16.3.13 Kernel Requirements

This kernel config should be enabled to support **switchdev** offload.

- **CONFIG_NET_ACT_CSUM** – needed for action csum
- **CONFIG_NET_ACT_PEDIT** – needed for header rewrite
- **CONFIG_NET_ACT_MIRRED** – needed for basic forward
- **CONFIG_NET_ACT_CT** – needed for CT (supported from kernel 5.6)
- **CONFIG_NET_ACT_VLAN** – needed for action vlan push/pop
- **CONFIG_NET_ACT_GACT**
- **CONFIG_NET_CLS_FLOWER**
- **CONFIG_NET_CLS_ACT**
- **CONFIG_NET_SWITCHDEV**
- **CONFIG_NET_TC_SKB_EXT** – needed for CT (supported from kernel 5.6)
- **CONFIG_NET_ACT_CT** – needed for CT (supported from kernel 5.6)
- **CONFIG_NFT_FLOW_OFFLOAD**
- **CONFIG_NET_ACT_TUNNEL_KEY**
- **CONFIG_NF_FLOW_TABLE** – needed for CT (supported from kernel 5.6)
- **CONFIG_SKB_EXTENSIONS** – needed for CT (supported from kernel 5.6)
- **CONFIG_NET_CLS_MATCHALL**
- **CONFIG_NET_ACT_POLICE**
- **CONFIG_MLX5_ESWITCH**

8.16.3.14 VF Metering

OVS-kernel supports offloading of VF metering (TX and RX) using **sysfs**. Metering for packets per second (PPS) and bytes per second (BPS) is supported.

The following example sets the Rx meter on VF 0 with value 10Mbps:

```
echo 10000000 > /sys/class/net/enp4s0f0/device/sriov/0/meters/rx/bps/rate
echo 65536 > /sys/class/net/enp4s0f0/device/sriov/0/meters/rx/bps/burst
```

The following example sets Tx meter on VF 0 with value 1000 PPS:

```
echo 1000 > /sys/class/net/enp4s0f0/device/sriov/0/meters/tx/pps/rate
echo 100 > /sys/class/net/enp4s0f0/device/sriov/0/meters/tx/pps/burst
```

Both **rate** and **burst** must be non-zero and **burst** may need to be adjusted according to requirements.

The following counters can be used to query the number dropped packet or bytes:

```
cat /sys/class/net/enp8s0f0/device/sriov/0/meters/rx/pps/packets_dropped
```

```
cat /sys/class/net/enp8s0f0/device/sriov/0/meters/rx/pps/bytes_dropped
cat /sys/class/net/enp8s0f0/device/sriov/0/meters/rx/bps/packets_dropped
cat /sys/class/net/enp8s0f0/device/sriov/0/meters/rx/bps/bytes_dropped
cat /sys/class/net/enp8s0f0/device/sriov/0/meters/tx/pps/packets_dropped
cat /sys/class/net/enp8s0f0/device/sriov/0/meters/tx/pps/bytes_dropped
cat /sys/class/net/enp8s0f0/device/sriov/0/meters/tx/bps/packets_dropped
cat /sys/class/net/enp8s0f0/device/sriov/0/meters/tx/bps/bytes_dropped
```

8.16.3.15 Representor Metering

Metering for uplink and VF representors traffic is supported.

Traffic going to a representor device can be a result of a miss in the embedded switch (eSwitch) FDB tables. This means that a packet which arrives from that representor into the eSwitch has not matched against the existing rules in the hardware FDB tables and must be forwarded to software to be handled there and is, therefore, forwarded to the originating representor device driver.

The meter allows to configure the max rate [packets per second] and max burst [packets] for traffic going to the representor driver. Any traffic exceeding values provided by the user are dropped in hardware. There are statistics that show the number of dropped packets.

The configuration of representor metering is done via **miss_rl_cfg**.

- Full path of the **miss_rl_cfg** parameter: **/sys/class/net//rep_config/miss_rl_cfg**
- Usage: **echo "<rate> <burst>" > /sys/class/net//rep_config/miss_rl_cfg**.
  - **rate** is the max rate of packets allowed for this representor (in packets/sec units)
  - **burst** is the max burst size allowed for this representor (in packets units)
  - Both values must be specified. Both of their default values is 0, signifying unlimited rate and burst.

To view the amount of packets and bytes dropped due to traffic exceeding the user-provided rate and burst, two read-only **sysfs** for statistics are available:

- **/sys/class/net//rep_config/miss_rl_dropped_bytes** – counts how many FDB-miss bytes are dropped due to reaching the miss limits
- **/sys/class/net//rep_config/miss_rl_dropped_packets** – counts how many FDB-miss packets are dropped due to reaching the miss limits

8.16.3.16 OVS Metering

There are two types of meters, kpps (kilobits per second) and pktps (packets per second). OVS-Kernel supports offloading both of them.

The following example is to offload a kpps meter.

1. Create OVS meter with a target rate:

```
ovs-ofctl -O OpenFlow13 add-meter ovs-sriov meter=1,kbps,band=type=drop,rate=204800
```

2. Delete the default rule:

```
ovs-ofctl del-flows ovs-sriov
```

3. Configure OpenFlow rules:

```
ovs-ofctl -O OpenFlow13 add-flow ovs-sriov 'ip,dl_dst=e4:11:22:33:44:50,actions= meter:1,output:enp4s0f0_0'
ovs-ofctl -O OpenFlow13 add-flow ovs-sriov 'ip,dl_src=e4:11:22:33:44:50,actions= output:enp4s0f0'
ovs-ofctl -O OpenFlow13 add-flow ovs-sriov 'arp,actions=normal'
```

Here, the VF bandwidth on the receiving side is limited by the rate configured in step 1.

4. Run iperf server and be ready to receive UDP traffic. On the outer node, run iperf client to send UDP traffic to this VF. After traffic starts, check the offloaded meter rule:

```
ovs-appctl dpctl/dump-flows --names type=offloaded

recirc_id(0),in_port(enp4s0f0),eth(dst=e4:11:22:33:44:50),eth_type(0x0800),ipv4(frag=no), packets:11626587,
bytes:17625889188, used:0.470s, actions:meter(0),enp4s0f0_0
```

To verify metering, the iperf client should set the target bandwidth to a number which is larger than the configured meter rate. Then it should apparent that packets are received with the limited rate on the server side and the extra packets are dropped by hardware.

8.16.3.17 Multiport eSwitch Mode

The multiport eSwitch mode allows adding rules on a VF representor with an action forwarding the packet to the physical port of the physical function. This can be used to implement failover or forward packets based on external information such as the cost of the route.

1. To configure multiport eSwitch mode, set the nvconfig parameter, **LAG_RESOURCE_ALLOCATION** .
2. After the driver loads, configure multiport eSwitch for each PF where **enp8s0f0** and **enp8s0f1** represent the netdevices for the PFs:

```
echo multiport_esw > /sys/class/net/enp8s0f0/compat/devlink/lag_port_select_mode
echo multiport_esw > /sys/class/net/enp8s0f1/compat/devlink/lag_port_select_mode
```

The mode becomes operational after entering **switchdev** mode on both PFs.

Rule example:

```
tc filter add dev enp8s0f0_0 prot ip root flower dst_ip 7.7.7.7 action mirred egress redirect dev enp8s0f1
```

8.16.4 OVS-DPDK Hardware Acceleration

OVS-DPDK builds and hardware acceleration have been officially discontinued and are no longer supported by the Open vSwitch (OVS) version bundled with the DOCA platform.

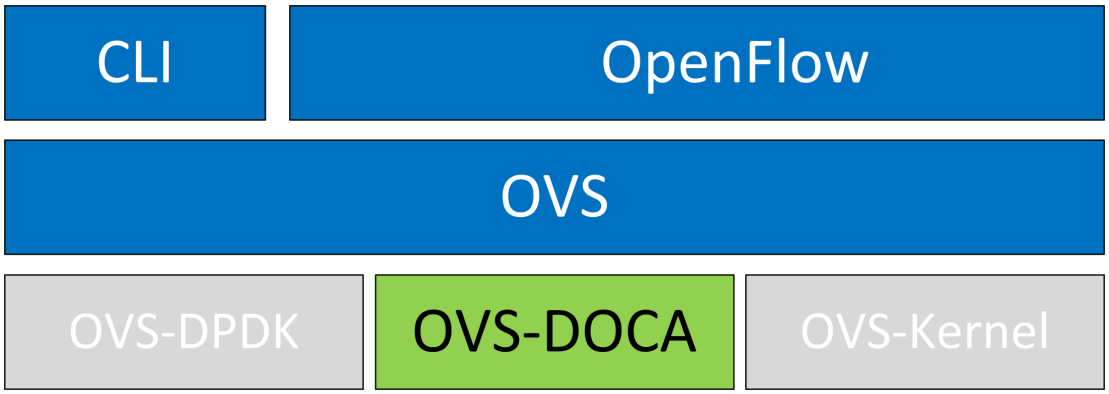
It is recommended to transition to OVS-DOCA Hardware Acceleration, that now fully replaces OVS-DPDK and provides significantly enhanced feature set along with improved performance.

OVS-DOCA is designed as a complete, modern replacement for OVS-DPDK, providing:

- A significantly enhanced feature set tailored for BlueField DPUs.
- Substantially improved hardware-accelerated network performance.
- Native integration with the DOCA platform.

8.16.5 OVS-DOCA Hardware Acceleration

OVS-DOCA is designed on top of NVIDIA's networking API to preserve the same OpenFlow, CLI, and data interfaces (e.g., vdpa, VF passthrough), as well as OVS-Kernel. While all OVS flavors make use of flow offloads for hardware acceleration, due to its architecture and use of DOCA libraries, the OVS-DOCA mode provides the most efficient performance and feature set among them, making the most out of NVIDA NICs and DPUs.



The following subsections provide the necessary steps to launch/deploy OVS DOCA.

8.16.5.1 1. Configuring OVS-DOCA

To configure OVS DOCA HW offloads:

1. Unbind the VFs:

```
echo 0000:04:00.2 > /sys/bus/pci/drivers/mlx5_core/unbind
echo 0000:04:00.3 > /sys/bus/pci/drivers/mlx5_core/unbind
```

⚠ VMs with attached VFs must be powered off to be able to unbind the VFs.

2. Change the e-switch mode from **legacy** to **switchdev** on the PF device (make sure all VFs are unbound):

```
echo switchdev > /sys/class/net/enp4s0f0/compat/devlink/mode
```

⚠ This command also creates the VF representor netdevices in the host OS.

To revert to SR-IOV **legacy** mode:

```
echo legacy > /sys/class/net/enp4s0f0/compat/devlink/mode
```

3. Bind the VFs:

```
echo 0000:04:00.2 > /sys/bus/pci/drivers/mlx5_core/bind
echo 0000:04:00.3 > /sys/bus/pci/drivers/mlx5_core/bind
```

4. Configure huge pages:

```
mkdir -p /hugepages
mount -t hugetlbfs hugetlbfs /hugepages
echo 4096 > /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
```

5. Run the Open vSwitch service:

```
systemctl start openvswitch
```

6. Enable DOCA mode and hardware offload (disabled by default):

```
ovs-vsctl --no-wait set Open_vSwitch . other_config:doca-init=true
ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
```

7. For hardware offload changes to take effect, restart the Open vSwitch service:
   - For Debian-based systems:

     ```
     systemctl restart openvswitch-switch
     ```

   - For RPM-based systems:

     ```
     systemctl restart openvswitch
     ```

   -
8. Create OVS-DOCA bridge:

```
ovs-vsctl --no-wait add-br br0-ovs -- set bridge br0-ovs datapath_type=doca
```

9. Add PF to OVS:

```
ovs-vsctl add-port br0-ovs enp4s0f0 -- set Interface enp4s0f0 type=doca
```

10. Add representor to OVS:

```
ovs-vsctl add-port br0-ovs enp4s0f0_0 -- set Interface enp4s0f0_0 type=doca
```

11. Optional configuration:
    a. To set port MTU, run:

    ```
    ovs-vsctl set interface enp4s0f0 mtu_request=9000
    ```

    Representors inherit their configuration from the ESW manager.

    b. To set VF/SF MAC, run:

    ```
    ovs-vsctl add-port br0-ovs enp4s0f0 -- set Interface enp4s0f0 type=doca options:dpdk-vf-
    mac=00:11:22:33:44:55
    ```

    Unbinding and rebinding the VFs/SFs is required for the change to take effect.

8.16.5.1.1 1.1. Setting Default Datapath

OVS commands can be simplified by configuring a default datapath type, which minimizes repetitive configurations and streamlines the OVS setup process for hardware-accelerated deployments.

To set a default datapath type, use the following command:

```
ovs-vsctl set Open_vSwitch . other_config:default-datapath-type=<type>
```

For example, to set the default datapath type to doca:

```
ovs-vsctl set Open_vSwitch . other_config:default-datapath-type=doca
```

This configuration allows bridges and interfaces to be created without specifying the datapath type explicitly for each command. For example:

```
ovs-vsctl --no-wait add-br br0-ovs
ovs-vsctl add-port br0-ovs enp4s0f0
```

This is equivalent to the following commands where the datapath type is explicitly set:

```
ovs-vsctl --no-wait add-br br0-ovs -- set bridge br0-ovs datapath_type=doca
ovs-vsctl add-port br0-ovs enp4s0f0 -- set Interface enp4s0f0 type=doca
```

If a non-supported datapath type is specified, OVS will automatically fall back to the default "system" type.

8.16.5.2 2. OVS-DOCA Design Considerations

OVS-DOCA is engineered to maximize the benefits of the DOCA offload architecture. To achieve this, specific behaviors of the userland datapath and ports have been modified from legacy OVS designs.

8.16.5.2.1 2.1. Eswitch Dependency

When configured in **switchdev** mode, the physical port and all supported functions share a single general domain (i.e., the eswitch) to execute offloaded flows.

Because all ports on the same eswitch are intrinsically linked to its main Physical Function (PF), their operational states are coupled. If the main PF is deactivated (e.g., administratively removed from OVS or its link state goes down), all dependent ports on that eswitch are automatically disabled as well.

8.16.5.2.2 2.2. Pre-allocated Offload Tables

To guarantee maximum flow insertion speeds, DOCA offloads utilize pre-allocated offload structures (entries and containers).

When the vSwitch daemon starts, these offloads are initialized with sensible, performance-optimized defaults. If your specific environment requires a different scale or number of offloads, you must adjust the OVS-DOCA specific configuration entries.

These configuration parameters are detailed in the next section.

8.16.5.2.3 2.3. Unsupported CT-CT-NAT

The specialized `ct-ct-nat` (Connection Tracking to Connection Tracking with Network Address Translation) mode, which is configurable in the standard OVS-kernel datapath, is not supported by OVS-DOCA.

8.16.5.3 3. OVS-DOCA Specific vSwitch Configuration

The following configuration is particularly useful or specific to OVS-DOCA mode.

> The full list of OVS vSwitch configuration is documented in `man ovs-vswitchd.conf.db`.

8.16.5.3.1 3.1. other_config

The following table provides `other_config` configurations which are global to the vSwitch (non-exhaustive list, check manpage for more):

Configuration	Description
`other_config:doca-init`	• Optional string, either true or false • Set this value to true to enable DOCA Flow HW offload • The default value is false. Changing this value requires restarting the daemon. • This is only relevant for userspace datapath
`other_config:hw-offload-ct-size`	• Optional string, containing an integer, at least 0 • Only for the DOCA offload provider on doca datapath • Configure the usable amount of connection tracking (CT) offload entries • The default value is 250000. Changing this value requires restarting the daemon. • Setting a value of 0 disables CT offload • Changing this configuration affects the OVS memory usage as CT tables are allocated on OVS start • Maximum number of supported connections is 2M Setting this parameter to more than 2M might result in failures. Do not exceed CT size of 1M for best performance.
`other_config:hw-offload-ct-ipv6-enabled`	• Optional string, either true or false • Only for the DOCA offload provider on doca datapath • Set this value to true to enable IPv6 CT offload • The default value is false. Changing this value requires restarting the daemon. • Changing this configuration affects the OVS memory usage as CT tables are allocated on OVS start

Configuration	Description
`other_config:doca-congestion-threshold`	• Optional string, containing an integer, in range 30 to 90 • The occupancy rate of DOCA offload structures that triggers a resize, as a percentage • Default to 80, but only relevant if `other_config:doca-init` is true. Changing this value requires restarting the daemon.
`other_config:ctl-pipe-size`	• Optional string, containing an integer • The initial size of DOCA control pipes • Default to 0, which is DOCA's internal default value
`other_config:ctl-pipe-infra-size`	• Optional string, containing an integer • The initial size of infrastructure DOCA control pipes: root, post-hash, post-ct, post-meter, split, miss. • Default to 0, which fallbacks to `other_config:ctl-pipe-size`
`other_config:pmd-quiet-idle`	• Optional string, either true or false • Allow the PMD threads to go into quiescent mode when idling. If no packets are received or waiting to be processed and sent, enter a continuous quiescent period. End this period as soon as a packet is received. • This option is disabled by default
`other_config:pmd-sleep-max`	• Optional string, containing an integer, in range 0 to 10,000 • Specifies the maximum sleep time in microseconds per iteration for a PMD thread which has received zero or a small amount of packets from the Rx queues it is polling. • The actual sleep time requested is based on the load of the Rx queues that the PMD polls and may be less than the maximum value • The default value is 0 microseconds, which means that the PMD does not sleep regardless of the load from the Rx queues that it polls • To avoid requesting very small sleeps (e.g., less than 10 µs) the value is rounded up to the nearest 10 µs • The maximum value is 10000 microseconds. • `other_config:pmd-maxsleep` is deprecated but still functional
`other_config:dpdk-max-memzones`	• Optional string, containing an integer • Specifies the maximum number of memzones that can be created in DPDK • The default is empty, keeping DPDK's default. Changing this value requires restarting the daemon.
`other_config:pmd-cpu-mask`	With PMD multi-threading support, OVS creates one PMD thread for each NUMA node by default if there is at least one DPDK interface added to OVS from that NUMA node. However, in cases where there are multiple ports/rxqs producing traffic, performance can be improved by creating multiple PMD threads running on separate cores. These PMD threads can share the workload by each being responsible for different ports/rxqs. Assignment of ports/rxqs to PMD threads is done automatically. A set bit in the mask means a PMD thread is created and pinned to the corresponding CPU core. For example, to run PMD threads on cores 1 and 2, run: `$ ovs-vsctl set Open_vSwitch . other_config:pmd-cpu-mask=0x6`
`other_config:hw-offload-ct-unidir-udp-enabled`	• Optional string, either true or false (default). Changing this value requires restarting the daemon. • Set this value to true to enable unidirectional UDP CT offload • Only for the DOCA offload provider on doca datapath

8.16.5.4 4. Offloading VXLAN Encapsulation/Decapsulation Actions

vSwitch in userspace rather than kernel-based Open vSwitch requires an additional bridge. The purpose of this bridge is to allow use of the kernel network stack for routing and ARP resolution.

The datapath must look up the routing table and ARP table to prepare the tunnel header and transmit data to the output port.

VXLAN encapsulation/decapsulation offload configuration is done with:

- PF on **0000:03:00.0** PCIe
- Local IP **56.56.67.1** – the **br-phy** interface is configured to this IP
- Remote IP **56.56.68.1**

To configure OVS DOCA VXLAN:

1. Create a **br-phy** bridge:

```
ovs-vsctl add-br br-phy -- set Bridge br-phy datapath_type=doca -- br-set-external-id br-phy bridge-id br-
phy -- set bridge br-phy fail-mode=standalone
```

2. Attach PF interface to **br-phy** bridge:

```
ovs-vsctl add-port br-phy enp4s0f0 -- set Interface enp4s0f0 type=doca
```

3. Configure IP to the bridge:

```
ip addr add 56.56.67.1/24 dev br-phy
```

4. Create a **br-ovs** bridge:

```
ovs-vsctl add-br br-ovs -- set Bridge br-ovs datapath_type=doca -- br-set-external-id br-ovs bridge-id br-
ovs -- set bridge br-ovs fail-mode=standalone
```

5. Attach representor to **br-ovs** :

```
ovs-vsctl add-port br-ovs enp4s0f0_0 -- set Interface enp4s0f0_0 type=doca
```

6. Add a port for the VXLAN tunnel:

```
ovs-vsctl add-port br-ovs vxlan0 -- set interface vxlan0 type=vxlan options:local_ip=56.56.67.1
options:remote_ip=56.56.68.1 options:key=45 options:dst_port=4789
```

8.16.5.4.1 4.1. VXLAN GBP Extension

The VXLAN group-based policy (GBP) model outlines an application-focused policy framework that specifies connectivity requirements for applications, independent of the network's physical layout.

Setting GBP extension for a VXLAN port allows for matching on and setting a GBP ID per flow. To enable GBP extension when port **vxlan0** is first added:

```
ovs-vsctl add-port br-int vxlan0 -- set interface vxlan0 type=vxlan options:key=30 options:remote_ip=10.0.30.1
options:exts=gbp
```

It is also possible to enable GBP extension for an existing VXLAN port:

```
ovs-vsctl set interface vxlan1 options:exts=gbp
```

This approach has a limitation that it does not take effect until after the OVS **vswitchd** service is restarted. In cases where there are multiple VXLAN ports, they must all share the same GBP extension configuration in their port options. A mixed configuration with some VXLAN ports having the GBP extension enabled and others disabled is not supported.

When GBP extension is enabled, the following OpenFlow rules which match on a GBP ID 32 or set a GBP ID 64 in the actions, can be offloaded:

```
ovs-ofctl add-flow br-int table=0,priority=100,in_port=vxlan0,tun_gbp_id=32 actions=output:pf0vf0
ovs-ofctl add-flow br-int table=0,priority=100,in_port=pf0vf0 actions=load:64->NXM_NX_TUN_GBP_ID[],output:vxlan0
```

8.16.5.4.2 4.2. VF-Tunnel Configuration

To offload underlay traffic effectively, configuring the underlay IP directly on the bridge port is insufficient. Instead, a dedicated VF or SF should be allocated, and its representor added to the **br-phy** bridge. This setup allows for proper offloading of underlay traffic.

To add the representor to the bridge, use the following command:

```
ovs-vsctl add-port br-phy <REP> -- set interface <REP> type=doca
```

Configure the underlay IP address directly on the VF or SF device.

Restrictions:

- **<REP>** refers to the Linux interface name of the representor
- The VF or SF must be bound to its driver before attaching the representor to OVS
- The VF or SF must reside in the same namespace as OVS
- The underlay IP address should be configured after the representor is attached to OVS. It is acceptable to restart OVS while the underlay IP is configured.

8.16.5.5 5. Offloading Connection Tracking

Connection tracking enables stateful packet processing by keeping a record of currently open connections.

OVS flows utilizing connection tracking can be accelerated using advanced NICs by offloading established connections.

To view offload statistics, run:

```
ovs-appctl dpctl/offload-stats-show
```

8.16.5.6 6. SR-IOV VF LAG

To configure OVS-DOCA SR-IOV VF LAG:

1. Enable SR-IOV on the NICs:

```
// It is recommended to query the parameters first to determine if a change is needed, to save potentially
unnecessary reboot.
mst start
mlxconfig -d <mst device> -y set PF_NUM_OF_VF_VALID=0   SRIOV_EN=1 NUM_OF_VFS=8
```

> ⚠ If configuration did change, perform a BlueField system reboot for the **mlxconfig** settings to take effect.

2. To be able to move to VF LAG mode while VFs/SFs exist, set the nvconig parameter **LAG_RESOURCE_ALLOCATION=1** in the BlueField Arm OS or on the host for ConnectX:

```
mst start
mlxconfig -d /dev/mst/mt*conf0 -y s LAG_RESOURCE_ALLOCATION=1
```

3. Allocate the desired number of VFs per port:

```
echo $n > /sys/class/net/<net name>/device/sriov_numvfs
```

4. Unbind all VFs:

```
echo <VF PCI> >/sys/bus/pci/drivers/mlx5_core/unbind
```

5. Change both NICs' mode to SwitchDev:

```
devlink dev eswitch set pci/<PCI> mode switchdev
```

6. Create Linux bonding using kernel modules:

```
modprobe bonding mode=<desired mode>
```

> ⚠ Other bonding parameters can be added here. The supported bond modes are Active-Backup, XOR, and LACP.

7. Bring all PFs and VFs down:

```
ip link set <PF/VF> down
```

8. Attach both PFs to the bond:

```
ip link set <PF> master bond0
```

9. Bring PFs and bond link up:

```
ip link set <PF0> up
ip link set <PF1> up
```

```
ip link set bond0 up
```

10. Add the bond interface to the bridge as **type=doca**:

```
ovs-vsctl add-port br-phy bond0 -- set Interface bond0 type=doca options:dpdk-lsc-interrupt=true
```

11. Add the VF representors of PF0 or PF1 to a bridge:

```
ovs-vsctl add-port br-phy enp4s0f0_0 -- set Interface enp4s0f0_0 type=doca
```

Or:

```
ovs-vsctl add-port br-phy enp4s0f1_0 -- set Interface enp4s0f1_0 type=doca
```

8.16.5.7 7. Multiport eSwitch Mode

In multiport eswitch mode, all uplinks and VFs/SFs representors of all physical ports are managed by the same hardware switch. This allows forwarding from the physical port entity to the physical port two entity.

1. To configure multiport eswitch mode, the nvconig parameter **LAG_RESOURCE_ALLOCATION=1** must be set in the BlueField Arm OS, according to the following instructions:

```
mst start
mlxconfig -d /dev/mst/mt*conf0 -y s  LAG_RESOURCE_ALLOCATION=1
```

2. Perform a BlueField system reboot for the **mlxconfig** settings to take effect.
3. After the driver loads, and after moving to switchdev mode, configure multiport eswitch for each PF where p0 and p1 represent the netdevices for the PFs:

```
devlink dev param set pci/0000:03:00.0 name esw_multiport value 1 cmode runtime
devlink dev param set pci/0000:03:00.1 name esw_multiport value 1 cmode runtime
```

> The mode becomes operational after entering switchdev mode on both PFs.

4. This mode can be activated by default in BlueField by adding the following line into **/etc/mellanox/mlnx-bf.conf**:

```
ENABLE_ESWITCH_MULTIPORT="yes"
```

While in this mode, the second port is not an eswitch manager, and should be add to OVS using this command:

```
ovs-vsctl add-port br-phy enp4s0f1 -- set interface enp4s0f1 type=doca
```

VFs for the second port can be added using this command:

```
ovs-vsctl add-port br-phy enp4s0f1_0 -- set interface enp4s0f1_0 type=doca
```

8.16.5.8 8. Offloading Geneve Encapsulation/Decapsulation

Geneve tunneling offload support includes matching on extension header.

OVS-DOCA Geneve option limitations:

- Only 1 Geneve option is supported
- Max option len is 7
- To change the Geneve option currently being matched and encapsulated, users must remove all ports or restart OVS and configure the new option
- Matching on Geneve options can work with **FLEX_PARSER** profile 0 (the default profile). Working with **FLEX_PARSER** profile 8 is also supported as well. To configure it, run:

```
mst start
mlxconfig -d <mst device> s FLEX_PARSER_PROFILE_ENABLE=8
```

Perform a BlueField system reboot for the **mlxconfig** settings to take effect.

To configure OVS-DOCA Geneve encapsulation/decapsulation:

1. Create a **br-phy** bridge:

```
ovs-vsctl --may-exist add-br br-phy -- set Bridge br-phy datapath_type=doca -- br-set-external-id br-phy
bridge-id br-phy -- set bridge br-phy fail-mode=standalone
```

2. Attach a PF interface to **br-phy** bridge:

```
ovs-vsctl add-port br-phy enp4s0f0 -- set Interface enp4s0f0 type=doca
```

3. Configure an IP to the bridge:

```
ifconfig br-phy <$local_ip_1> up
```

4. Create a **br-int** bridge:

```
ovs-vsctl --may-exist add-br br-int -- set Bridge br-int datapath_type=doca -- br-set-external-id br-int
bridge-id br-int -- set bridge br-int fail-mode=standalone
```

5. Attach a representor to **br-int** :

```
ovs-vsctl add-port br-int rep$x -- set Interface rep$x type=doca
```

6. Add a port for the Geneve tunnel:

```
ovs-vsctl add-port br-int geneve0 -- set interface geneve0 type=geneve options:key=<VNI>
options:remote_ip=<$remote_ip_1> options:local_ip=<$local_ip_1>
```

8.16.5.9 9. GRE Tunnel Offloads

To configure OVS-DOCA GRE encapsulation/decapsulation:

1. Create a **br-phy** bridge:

```
ovs-vsctl --may-exist add-br br-phy -- set Bridge br-phy datapath_type=doca -- br-set-external-id br-phy
bridge-id br-phy -- set bridge br-phy fail-mode=standalone
```

2. Attach a PF interface to **br-phy** bridge:

```
ovs-vsctl add-port br-phy enp4s0f0 -- set Interface enp4s0f0 type=doca
```

3. Configure an IP to the bridge:

```
ifconfig br-phy <$local_ip_1> up
```

4. Create a **br-int** bridge:

```
ovs-vsctl --may-exist add-br br-int -- set Bridge br-int datapath_type=doca -- br-set-external-id br-int
bridge-id br-int -- set bridge br-int fail-mode=standalone
```

5. Attach a representor to **br-int**:

```
ovs-vsctl add-port br-int enp4s0f0_0 -- set Interface enp4s0f0_0 type=doca
```

6. Add a port for the GRE tunnel:

```
ovs-vsctl add-port br-int gre0 -- set interface gre0 type=gre options:key=<VNI>
options:remote_ip=<$remote_ip_1> options:local_ip=<$local_ip_1>
```

8.16.5.10 10. Slow Path Rate Limiting/SW-Meter

Slow path rate limiting allows controlling the rate of traffic that bypasses hardware offload rules and is subsequently processed by software.

To configure slow path rate limiting:

1. Create a **br-phy** bridge:

```
ovs-vsctl --may-exist add-br br-phy -- set Bridge br-phy datapath_type=doca -- br-set-external-id br-phy
bridge-id br-phy -- set bridge br-phy fail-mode=standalone
```

2. Attach a PF interface to **br-phy** bridge:

```
ovs-vsctl add-port br-phy enp4s0f0 -- set Interface enp4s0f0 type=doca
```

3. Rate limit **enp4s0f0** to 10Kpps with 6K burst size:

```
ovs-vsctl set interface enp4s0f0 options:sw-meter=pps:10k:6k
```

A dry-run option is also supported to allow testing different software meter configurations in a production environment. This allows gathering statistics without impacting the actual traffic flow.

These statistics can then be analyzed to determine appropriate rate limiting thresholds. When the dry-run option is enabled, traffic is not dropped or rate-limited, allowing normal operations to continue without disruption. However, the system simulates the rate limiting process and increment counters as though packets are being dropped.

To enable slow path rate limiting dry-run:

1. Create a **br-phy** bridge:

```
ovs-vsctl --may-exist add-br br-phy -- set Bridge br-phy datapath_type=doca -- br-set-external-id br-phy
bridge-id br-phy -- set bridge br-phy fail-mode=standalone
```

2. Attach a PF interface to **br-phy** bridge:

```
ovs-vsctl add-port br-phy enp4s0f0 -- set Interface enp4s0f0 type=doca
```

3. Rate limit **enp4s0f0** to 10Kpps with 6K burst size:

```
ovs-vsctl set interface enp4s0f0 options:sw-meter=pps:10k:6k
```

4. Set the **sw-meter-dry-run** option:

```
ovs-vsctl set interface enp4s0f0 options:sw-meter-dry-run=true
```

8.16.5.11 11. Hairpin

Hairpin allows forwarding packets from wire to wire.

To configure hairpin :

1. Create a **br-phy** bridge:

```
ovs-vsctl --may-exist add-br br-phy -- set Bridge br-phy datapath_type=doca -- br-set-external-id br-phy
bridge-id br-phy -- set bridge br-phy fail-mode=standalone
```

2. Attach a PF interface to **br-phy** bridge:

```
ovs-vsctl add-port br-phy enp4s0f0 -- set Interface enp4s0f0 type=doca
```

3. Add hairpin OpenFlow rule:

```
ovs-ofctl add-flow br-phy"in_port=enp4s0f0,ip,actions=in_port"
```

8.16.5.12 12. OpenFlow Meters

OVS-DOCA supports OpenFlow meter action as covered in this document in section "OpenFlow Meters". In addition, OVS-DOCA supports chaining multiple meter actions together in a single datapth rule.

The following is an example configuration of such OpenFlow rules:

```
ovs-ofctl add-flow br-phy -O OpenFlow13 "table=0,priority=1,in_port=enp4s0f0_0,ip actions=meter=1,resubmit(,1)"
ovs-ofctl add-flow br-phy -O OpenFlow13 "table=1,priority=1,in_port=enp4s0f0_0,ip actions=meter=2,normal"
```

Meter actions are applied sequentially, first using meter ID 1 and then using meter ID 2.

Use case examples for such a configuration:

- Rate limiting the same logical flow with different meter types—bytes per second and packets per second
- Metering a group of flows. As meter IDs can be used by multiple flows, it is possible to re-use meter ID 2 from this example with other logical flows; thus, making sure that their cumulative bandwidth is limited by the meter.

8.16.5.13 13. DP-HASH Offloads

OVS supports group configuration. The "select" type executes one bucket in the group, balancing across the buckets according to their weights. To select a bucket, for each live bucket, OVS hashes flow data with the bucket ID and multiplies that by the bucket weight to obtain a "score". The bucket with the highest score is selected.

> For more details, refer to the ovs-ofctl man.

For example:

- **`ovs-ofctl add-group br-int 'group_id=1,type=select,bucket=<port1>'`**
- **`ovs-ofctl add-flow br-int in_port=<port0>,actions=group=1`**

Limitations:

- Offloads are supported on IP traffic only (IPv4 or IPv6)

8.16.5.14 14. sFlow

The sFlow standard outlines a method for capturing traffic data in switched or routed networks. It employs sampling technology to gather statistics from the device, making it suitable for high-speed networks.

With a predetermined sampling rate, one out of every N packets is captured. While this sampling method does not yield completely accurate results, it does offer acceptable accuracy.

To activate sampling for 0.2% of all traffic traversing an OVS bridge named **`br-int`**, run:

```
ovs-vsctl -- --id=@sflow create sflow agent=lo target=127.0.0.1:6343 header=96 sampling=512 -- set bridge br-int
sflow=@sflow
```

With this sFlow configuration on the bridge, captured packets are mirrored to an sFlow collector application that listens on the default sFlow port, 6343, on localhost.

> sFlow collector applications fall outside the scope of this guide.

It is possible to set the sampling rate to 1 while configuring sFlow on a bridge, which effectively mirrors all traffic to the sFlow collector.

8.16.5.15 15. Mirroring

Mirroring can be used to duplicate packets from one port to another besides the original packet destination. This can be done using either an OpenFlow output action or an **ovs-vsctl create mirror** command.

For example, to configure mirror all traffic from port **enp4s0f0_0** to port **enp4s0f0_1** on OVS bridge **br-int**, run:

```
ovs-vsctl -- --id=@p1 get port enp4s0f0_0 -- --id=@p2 get port enp4s0f0_1 -- --id=@m create mirror name=m1
select_dst_port=@p1 select_src_port=@p1 output-port=@p2 -- set bridge br-int mirrors=@m
```

This produces datapath rules with multiple output ports. Each output port permutation requires a different mirror configuration. By default, only 128 different such configurations can be supported. To change this number, use the **doca-mirror-max** **other_config**. For example, set **other_config:doca-mirror-max** to 2048 by running the following:

```
ovs-vsctl set Open_vSwitch . other_config:doca-mirror-max=2048
```

8.16.5.16 16. Guaranteed Packet Rate

Guaranteed Packet Rate (GPR) is a traffic control feature designed to mitigate noisy neighbor scenarios, in which one port transmits at a significantly higher rate than others—potentially causing starvation for other ports sharing the same queue.

GPR helps ensure fair access to resources by exposing per-core and per-port metering. These meters allow administrators to configure limits on:

- The number of packets per second (PPS) each port can transmit
- The total number of packets reaching the software layer

Benefits of GPR:

- Prevents traffic starvation caused by aggressive ports
- Enables fair scheduling across representors and PFs
- Can be dynamically configured—no service restart required

Enabling and configuring GPR:

- To set the desired GPR mode:

  ```
  ovs-vsctl set o . other_config:gpr-mode=<mode>
  ```

  Where **<mode>** can be one of the following:

Mode	Description
disabled	Default. GPR is turned off
rep-only	GPR applies only to representor ports
all-ports	GPR applies to all ports, including PFs

- To set metering rates:

```
ovs-vsctl set o . other_config:per-core-meter-rate=<rate-in-pps>
ovs-vsctl set o . other_config:per-port-meter-rate=<rate-in-pps>
```

  - **per-core-meter-rate** – Sets the maximum PPS allowed per core
  - **per-port-meter-rate** – Sets the maximum PPS allowed per port

    If **per-port-meter-rate** is not explicitly set, it is calculated automatically using the formula:

    ```
    per-port-meter-rate = (per-core-meter-rate * number_of_cores) / number_of_attached_ports
    ```

8.16.5.17 17. Configuring OVS-DOCA Pre-Miss Rules (Kernel Punting)

The OVS-DOCA pre-miss feature allows administrators to match specific Ethernet types (EtherTypes) and route them directly to the OS kernel. By punting these specific protocols to the kernel, they bypass the Software OVS datapath entirely, saving processing overhead for control-plane or specific L2 traffic.

8.16.5.17.1 17.1. Constraints and Default Behavior

- Port Restriction: This configuration applies strictly to Physical Function (PF) ports.
- Rule Limit: You can configure a maximum of 16 distinct EtherTypes to be sent to the kernel.
- Default Rules: If no **doca-pre-miss-rules** are explicitly configured, OVS-DOCA defaults to sending LACP, LLDP, and 802.1X packets to the kernel.

8.16.5.17.2 17.2. Configuration Commands

1. Set custom pre-miss rules: To punt specific EtherTypes to the kernel, provide a comma-separated list of hexadecimal EtherType values.

   ```
   # Syntax
   ovs-vsctl set interface <pf_port> options:doca-pre-miss-rules="<comma-separated-list-of-ethertypes>"

   # Example: Punt IPv4 (0x800) and custom EtherType (0x8899) on port p0
   ovs-vsctl set interface p0 options:doca-pre-miss-rules="0x800,0x8899"
   ```

   Packets matching these EtherTypes will be sent to the kernel and will not be processed by OVS.

2. Clear all pre-miss rules (drop defaults): To clear the default rules and prevent *any* pre-miss punting to the kernel, you must set the list to an empty value containing at least one whitespace character.

   ```
   # Clear default rules by passing a whitespace string
   ovs-vsctl set interface <pf_port> options:doca-pre-miss-rules=" "
   ```

3. Restore default pre-miss rules: To restore the default behavior (punting LACP, LLDP, and 802.1X to the kernel), completely remove the **doca-pre-miss-rules** option from the interface configuration.

```
# Remove the configuration key to restore defaults
ovs-vsctl remove interface <pf_port> options doca-pre-miss-rules
```

8.16.5.18 18. OVS-DOCA Known Limitations

- When using two PFs with 127 VFs each and adding their representors to OVS bridge, the user must configure **dpdk-memzones**:

```
ovs-vsctl set o . other_config:dpdk-max-memzones=6500
restart ovs
```

- In an OVS topology that includes both physical and internal bridges, sFlow offloads are only supported on the internal bridge when employing a VXLAN tunnel. Utilizing sFlow on the physical bridge leads to only partial offload of flows in this scenario.

8.16.5.19 19. OVS-DOCA Debugging

Additional debugging information can be enabled in the vSwitch log file using the **dbg** log level:

```
(
    topics='netdev|ofproto|ofp|odp|doca'
    IFS=$'\n'; for topic in $(ovs-appctl vlog/list | grep -E "$topics" | cut -d' ' -f1)
    do
        printf "$topic:file:dbg "
    done
) | xargs ovs-appctl vlog/set
```

The listed topics are relevant to DOCA offload operations.

Coverage counters specific to the DOCA offload provider have been added. The following command should be used to check them:

```
ovs-appctl coverage/show # Print the current non-zero coverage counters
```

The following table provides the meaning behind these DOCA-specific counters:

Counter	Description
doca_async_queue_full	The asynchronous offload insertion queue was full while the daemon attempted to insert a new offload. The queue will have been flushed and insertion attempted again. This is not a fatal error but is the sign of a slowed down hardware.
doca_async_queue_blocked	The asynchronous offload insertion queue has remained full even after several attempts to flush its currently enqueued requests. While not a fatal error, it should never happen during normal offload operations and should be considered a bug.
doca_async_add_failed	An asynchronous insertion failed specifically due to its asynchronous nature. This is not expected to happen and should be considered a bug.
doca_pipe_resize	The number of time a DOCA pipe has been resized. This is normal and expected as DOCA pipes receives more entries.

Counter	Description
doca_pipe_resize_over_10_ms	A DOCA pipe resize took longer than 10ms to complete. It can happen infrequently. If a sudden drop in insertion rate is measured, this counter could help identify the root cause.

8.16.5.20 20. Scaling Megaflows

Megaflows aggregate multiple microflows into a single flow entry, reduce the load on the flow table, and improve packet processing efficiency. Scaling megaflows in OVS is crucial for optimizing network performance and ensuring efficient handling of high traffic volumes. By default, OVS-DOCA can handle up to 200k megaflows.

To effectively manage and scale megaflows, several key configurations in the **other_config** section of OVS can be adjusted:

- The **flow-limit** parameter sets the maximum number of flows that can be stored in the flow table, helping to control memory usage and prevent overflow.
- The **max-revalidator** parameter defines the longest duration (in milliseconds) that re-validator threads will wait before initiating flow revalidation. It is crucial to understand that this represents the upper limit, and the actual timeout employed by OVS is the lesser of the **max-idle** and **max-revalidator** values. Modifying this parameter is generally not recommended without a thorough understanding of its effects. For systems with less powerful CPUs, setting a higher **max-revalidator** value is suggested to compensate for reduced computational capacity and ensure revalidation completes.

Fine-tuning these settings can improve the scalability and performance of an OVS deployment, allowing it to manage a greater number of megaflows efficiently.

- To set **flow-limit** (default is 200k):

```
$ ovs-vsctl set o . other_config:flow-limit=<desired_value>
```

- To set **max-revalidator** (default is 250ms).

```
$ ovs-vsctl set o . other_config:max-revalidator=<desired_value>
```

8.16.5.21 21. Software Datapath Packet Capture (ovs-doca-tcpdump)

Troubleshooting Use Only

This tool is intended exclusively for debugging and flow verification, and is not recommended for production environments. Enabling packet capture incurs a performance penalty and affects datapath throughput. Additionally, only one instance of **ovs-doca-tcpdump** can run at a time; attempting to launch a concurrent instance will result in an error.

The **ovs-doca-tcpdump** utility captures and displays packets processed by the OVS software datapath. It utilizes the Scapy Python library for packet parsing and display, providing enhanced packet analysis capabilities with metadata detailing interface and hook information.

8.16.5.21.1 21.1. Prerequisites

ovs-doca-tcpdump requires the Scapy Python library. If Scapy is not already installed on your system, install it using one of the following methods:

```
# Using pip
pip3 install scapy

# Debian/Ubuntu
apt-get install python3-scapy

# RHEL/Fedora/CentOS
dnf install python3-scapy
```

8.16.5.21.2 21.2. Tool Capabilities

Capability	Description
Configurable capture hooks	Determines when the packet is captured in the datapath: • **rx** (default): captures after initial processing • **tx** : captures before packets exit the datapath • **rx_pre_restore** :Captures before hardware metadata is stripped Use **--list-hooks** to view all available hooks.
Target scope	Capture traffic on specific interfaces or globally using **-i any** . Use **--list-interfaces** to discover available OVS port names.
Filter support	Supports standard Berkeley Packet Filter (BPF) expressions (e.g., **tcp port 80** , **host 192.168.1.1** , **icmp**).
Display options	• **-v** – Detailed Scapy packet dissection. • **-x** – Hex dump. • **-t** – Timestamps. • **-c** – Limit packet count. • **-s** – Set snapshot length.
PCAP export	Use **-w <file>** to write captured packets to a standard PCAP file for offline analysis with tools like Wireshark.
Control	Managed internally via UnixCtl commands: • **dpif-doca/tcpdump-set** – enable packet capture • **dpif-doca/tcpdump-unset** – disable capture and clear configs • **dpif-doca/tcpdump-show** – display active configuration

8.16.5.21.3 21.3. Example Usage

```
# Capture all software datapath traffic in verbose mode
ovs-doca-tcpdump -v

# Capture from a specific interface with TX hook
ovs-doca-tcpdump -i pf0vf0:tx
```

```
# Capture from multiple interfaces with different hooks
ovs-doca-tcpdump -i pf0vf0:rx_pre_restore,rx,tx+pf0vf1:tx

# Save capture to a PCAP file
ovs-doca-tcpdump -w capture.pcap

# Capture only ICMP packets
ovs-doca-tcpdump icmp

# Capture 100 packets with timestamps
ovs-doca-tcpdump -c 100 -t

# Verbose output with hex dump
ovs-doca-tcpdump -v -x

# Filter for HTTP traffic on a specific interface and save to file
ovs-doca-tcpdump -i pf0vf0 -w http_traffic.pcap "tcp port 80"

# List available interfaces
ovs-doca-tcpdump --list-interfaces

# List available hooks
ovs-doca-tcpdump --list-hooks
```

8.16.5.22 Multi-PMD Configuration

Poll Mode Driver (PMD) threads are responsible for performing datapath processing of packets that traverse the software datapath (e.g., slow path) and for handling connection tracking (CT) offloads.

Scaling PMD threads can improve those on systems with multiple cores. The following configuration steps enable you to control how PMD threads are scheduled and how ports are assigned to RX queues.

8.16.5.22.1 Configuring PMD CPU Mask

Set the **pmd-cpu-mask** in **other_config** to specify which CPU cores are used by PMD threads. By default, Open vSwitch (OVS) assigns one core per NUMA node.

> ⚠ Changing the PMD CPU mask requires restarting the OVS daemon for the changes to take effect.

8.16.5.22.2 Configuring Multiple RX Queues

For ESW manager ports, configure multiple RX queues to allow parallel packet processing.

The following is an example command to configure **N** RX queues for port **P**:

```
ovs-vsctl set interface <P> options:n_rxq=<N>
```

> ⚠ This change does not require an OVS restart. However, the port and its representors will be restarted, which flushes related flows and offloads. Representors inherit their configuration from the ESW manager.

8.16.5.22.3 Scheduling Queues on PMD Threads

OVS automatically schedules port-queue tuples across available PMD threads, using only cores from the card's NUMA node by default.

You can customize this scheduling by setting the **pmd-rxq-affinity** on the interface, using **<QUEUE>:<CORE>** tuples:

```
ovs-vsctl set interface <P> other_config:pmd-rxq-affinity="0:4,1:17"
```

In this example:

- Queue 0 is scheduled on core 4
- Queue 1 is scheduled on core 17

8.16.5.22.4 Verifying PMD Thread Scheduling

To view the current PMD thread-to-queue scheduling, run:

```
ovs-appctl dpif-doca/pmd-rxq-show
```

8.16.5.22.5 Limitations

CT offload size is configured by hw-offload-ct-size (500,000 by default). This size is distributed among the number of PMDs, configured by **pmd-cpu-mask**) (default is by the number of NUMAs in the system).

Each port accesses to CT according to number of RX queue it use (n_rxq).

For example if n_rxq is set to 2 but pmd-cpu-mask has 3 PMDs, only 2/3 of the doca-ct size can be utilized.

To overcome it, either align n_rxq to the number of PMDs if applicable (single NUMA systems) or increase hw-offload-ct-size by this factor (3/2 in this case).

For more information, refer to the Open vSwitch DPDK Installation Guide.

8.16.6 OVS Metrics

Open vSwitch (OVS) with DOCA provides Prometheus-compatible metrics to monitor health and performance.

8.16.6.1 Introduction

NVIDIA DOCA-accelerated Open vSwitch (OVS) offers comprehensive observability through real-time performance and health metrics. These metrics provide deep visibility into packet processing paths (both software and hardware) and are critical for monitoring, debugging, and optimizing virtual network functions (VNFs) and cloud-native workloads.

Metrics are exported in Prometheus format, making them easy to integrate with popular observability stacks such as Prometheus and Grafana. DOCA Telemetry Services (DTS) can also aggregate and forward these metrics as part of a broader telemetry pipeline.

This page outlines the available tools for accessing OVS metrics, how to configure them, and how to integrate them with Prometheus and DTS for automated monitoring and alerting.

Python Dependencies

You must have Python 3.8 or later installed. Operating systems utilizing older versions of Python are strictly not supported.

8.16.6.2 Prometheus Metrics

8.16.6.2.1 Running ovs-exporter

The OVS metrics exporter (**ovs-exporter**) is included with the OVS-DOCA package. It runs an HTTP server that can be scraped by a Prometheus instance.

Use **systemd** to manage the exporter:

Managing the ovs-exporter

```
# Start and enable the service
systemctl start ovs-exporter
systemctl enable ovs-exporter

# Check service status
systemctl status ovs-exporter

# Stop or disable the service
systemctl stop ovs-exporter
systemctl disable ovs-exporter
```

The default configuration exposes metrics on TCP port **6103** .

To modify the configuration, edit **/etc/openvswitch/ovs-exporter.conf** :

Default ovs-exporter configuration

```
[exporter]
port = 6103
extended = no
debug = no
```

OVS metrics are divided into three pages:

- **basic** – always enabled; suitable for most users
- **extended** – includes additional software counters
- **debug** – includes experimental or low-level counters

Additional metric pages can be enabled by setting their respective options to **yes** in the configuration file.

- **Debug** metrics are experimental and may change without notice.
- **Basic** and **extended** metrics are stable across LTS releases.
- Metrics scheduled for removal are marked as **deprecated** in their Prometheus **HELP** description.

To retrieve metrics from the HTTP endpoint:

HTTP Metrics query

```
curl -s 0.0.0.0:6103
```

8.16.6.2.2 Integration with DOCA Telemetry Services

To integrate OVS metrics with DTS, configure the **prometheus_aggr** DTS provider to scrape the **ovs-exporter** endpoint.

For configuration details, refer to the Prometheus Aggregator section in the *DOCA Telemetry Service Guide* for more information.

8.16.6.3 OVS CLI Metrics

OVS also exposes metrics directly through its control socket, without needing the **ovs-exporter** service. Run:

```
ovs-appctl metrics/show [-x] [-d]
```

- Use **-x** to include extended metrics
- Use **-d** to include debug metrics

The output format is Prometheus-compatible and printed to the terminal.

8.16.6.4 OVS-metrics Dashboard

The **ovs-metrics** tool provides a real-time view of hardware and software counters:

```
ovs-metrics
```

Dependency

This tool requires the **python3-doca-openvswitch** package. To install it:

- On Ubuntu:

```
sudo apt install python3-doca-openvswitch
```

- On RHEL:

```
sudo yum install python3-doca-openvswitch
```

The tool reports the following metrics every second:

- **sw-pkts** – number of packets passed in software (total)
- **sw-pps** – last second packet per second in software
- **sw-conns** – number of CT connections in software
- **sw-cps** – last second new connections per second in software

- **hw-pkts** – number of packets passed in hardware (total)
- **hw-pps** – last second packet per second in hardware
- **hw-conns** – number of CT connections in hardware
- **hw-cps** – last second new connections per second in hardware
- **enqueued** – number of rules pending hardware offload
- **hw-rules** – number of offloaded rules in hardware (including infrastructure rules)
- **hw-rps** – last second new hardware rules offloaded per second

8.16.6.5 OVS-Per Flow Counters

OVS allows users to register human-readable names for flow cookies. This capability exposes per-flow counters via **ovs-metrics**, which can then be accessed and monitored through DTS.

8.16.6.5.1 Management Commands

The following **ovs-appctl** commands are used to manage flow metric registrations.

8.16.6.5.1.1 Register or Update Metric

Associates a name with a flow cookie. If the cookie is already registered, this command updates the associated name (renames it).

```
ovs-appctl ofproto/flow-metric-add <bridge> <cookie> <name>
```

8.16.6.5.1.2 Remove Metric

Deletes the mapping for a specific cookie.

```
ovs-appctl ofproto/flow-metric-del <bridge> <cookie>
```

8.16.6.5.1.3 List Metrics

Displays all currently registered flow metrics.

```
ovs-appctl ofproto/flow-metric-list
```

8.16.6.5.2 Parameters

Parameter	Description	Format / Constraints
bridge	The name of the existing OVS bridge.	Example: **br0**
cookie	The flow identifier.	Hexadecimal value prefixed with **0x** (e.g., **0x1234**).
name	The human-readable identifier for the flow.	Must match the pattern **[A-Za-z][_A-Za-z0-9]*** (e.g., **failover_counter**).

8.16.6.5.3 Operational Behavior

- Metric registration is local to the specified bridge. A name is associated with a cookie only within that bridge's context.
- Counters are aggregated (summed) across all OpenFlow rules in the bridge that share the same registered cookie.
- A metric entry is automatically removed from the system only when the last OpenFlow entry utilizing that specific cookie is removed from the bridge.

8.16.7 OVS Inside BlueField

8.16.7.1 Verifying Host Connection on Linux

When the DPU is connected to another DPU on another machine, manually assign IP addresses with the same subnet to both ends of the connection.

1. Assuming the link is connected to **p3p1** on the other host, run:

```
$ ifconfig p3p1 192.168.200.1/24 up
```

2. On the host which the DPU is connected to, run:

```
$ ifconfig p4p2 192.168.200.2/24 up
```

3. Have one ping the other. This is an example of the DPU pinging the host:

```
$ ping 192.168.200.1
```

8.16.7.2 Verifying Connection from Host to BlueField

There are two SFs configured on the BlueField device, **enp3s0f0s0** and **enp3s0f1s0**, and their representors are part of the built-in bridge. These interfaces will get IP addresses from the DHCP server if it is present. Otherwise it is possible to configure IP address from the host. It is possible to access BlueField via the SF netdev interfaces.

For example:

1. Verify the default OVS configuration. Run:

```
# ovs-vsctl show
5668f9a6-6b93-49cf-a72a-14fd64b4c82b
    Bridge ovsbr1
        Port pf0hpf
            Interface pf0hpf
        Port ovsbr1
            Interface ovsbr1
                type: internal
        Port p0
            Interface p0
        Port en3f0pf0sf0
            Interface en3f0pf0sf0
    Bridge ovsbr2
        Port en3f1pf1sf0
            Interface en3f1pf1sf0
        Port ovsbr2
            Interface ovsbr2
                type: internal
        Port pf1hpf
            Interface pf1hpf
        Port p1
            Interface p1
```

```
    ovs_version: "2.14.1"
```

2. Verify whether the SF netdev received an IP address from the DHCP server. If not, assign a static IP. Run:

```
# ifconfig enp3s0f0s0
enp3s0f0s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.200.125  netmask 255.255.255.0  broadcast 192.168.200.255
        inet6 fe80::8e:bcff:fe36:19bc  prefixlen 64  scopeid 0x20<link>
        ether 02:8e:bc:36:19:bc  txqueuelen 1000  (Ethernet)
        RX packets 3730  bytes 1217558 (1.1 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 22  bytes 2220 (2.1 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

3. Verify the connection of the configured IP address. Run:

```
# ping 192.168.200.25 -c 5
PING 192.168.200.25 (192.168.200.25) 56(84) bytes of data.
64 bytes from 192.168.200.25: icmp_seq=1 ttl=64 time=0.228 ms
64 bytes from 192.168.200.25: icmp_seq=2 ttl=64 time=0.175 ms
64 bytes from 192.168.200.25: icmp_seq=3 ttl=64 time=0.232 ms
64 bytes from 192.168.200.25: icmp_seq=4 ttl=64 time=0.174 ms
64 bytes from 192.168.200.25: icmp_seq=5 ttl=64 time=0.168 ms

--- 192.168.200.25 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 91ms
rtt min/avg/max/mdev = 0.168/0.195/0.232/0.031 ms
```

8.16.7.3 Verifying Host Connection on Windows

Set IP address on the Windows side for the RShim or Physical network adapter, please run the following command in Command Prompt:

```
PS C:\Users\Administrator> New-NetIPAddress -InterfaceAlias "Ethernet 16" -IPAddress "192.168.100.1" -PrefixLength
22
```

To get the interface name, please run the following command in Command Prompt:

```
PS C:\Users\Administrator> Get-NetAdapter
```

Output should give us the interface name that matches the description (e.g. NVIDIA BlueField Management Network Adapter).

```
Ethernet 2          NVIDIA ConnectX-4 Lx Ethernet Adapter        6 Not Present  24-8A-07-0D-E8-1D
Ethernet 6          NVIDIA ConnectX-4 Lx Ethernet Ad...#2       23 Not Present  24-8A-07-0D-E8-1C
Ethernet 16         NVIDIA BlueField Management Netw...#2       15 Up           CA-FE-01-CA-FE-02
```

Once IP address is set, Have one ping the other.

```
C:\Windows\system32>ping 192.168.100.2

Pinging 192.168.100.2 with 32 bytes of data:
Reply from 192.168.100.2: bytes=32 time=148ms TTL=64
Reply from 192.168.100.2: bytes=32 time=152ms TTL=64
Reply from 192.168.100.2: bytes=32 time=158ms TTL=64
Reply from 192.168.100.2: bytes=32 time=158ms TTL=64
```

8.16.8 Virtual Switch on BlueField

For general information on OVS offload using ASAP² direct, please refer to the MLNX_OFED documentation under OVS Offload Using ASAP² Direct.

⚠ ASAP^2 is only supported in Embedded (DPU) mode.

NVIDIA® BlueField® networking platforms (DPUs or SuperNICs) support ASAP^2 technology. It utilizes the representors mentioned in the previous section. The BlueField software package includes OVS installation which already supports ASAP^2. The virtual switch running on the Arm cores allows us to pass all the traffic to and from the host functions through the Arm cores while performing all the operations supported by OVS. ASAP^2 allows us to offload the datapath by programming the NIC embedded switch and avoiding the need to pass every packet through the Arm cores. The control plane remains the same as working with standard OVS.

OVS bridges are created by default upon first boot of the BlueField after BFB installation.

If manual configuration of the default settings for the OVS bridge is desired, run:

```
systemctl start openvswitch-switch.service
ovs-vsctl add-port ovsbr1 p0
ovs-vsctl add-port ovsbr1 pf0hpf
ovs-vsctl add-port ovsbr2 p1
ovs-vsctl add-port ovsbr2 pf1hpf
```

To verify successful bridging:

```
$ ovs-vsctl show
9f635bd1-a9fd-4f30-9bdc-b3fa21f8940a
    Bridge ovsbr2
        Port ovsbr2
            Interface ovsbr2
                type: internal
        Port p1
            Interface p1
        Port pf1sf0
            Interface en3f1pf1sf0
        Port pf1hpf
            Interface pf1hpf
    Bridge ovsbr1
        Port pf0hpf
            Interface pf0hpf
        Port p0
            Interface p0
        Port ovsbr1
            Interface ovsbr1
                type: internal
        Port pf0sf0
            Interface en3f0pf0sf0
    ovs_version: "2.14.1"
```

The host is now connected to the network.

⚠ TC-offload is not supported for IPv6 fragment packets. To make IPv6 fragment packets pass through OVS, the MTU of a specific port must be set to equal to or larger than the fragmented packet size. IPv4 fragment packets can be TC-offloaded as their packet size is not checked by OVS.

8.16.8.1 Verifying Host Connection on Linux

When BlueField is connected to another BlueField on another machine, manually assign IP addresses with the same subnet to both ends of the connection.

1. Assuming the link is connected to p3p1 on the other host, run:

   ```
   $ ifconfig p3p1 192.168.200.1/24 up
   ```

2. On the host to which BlueField is connected, run:

```
$ ifconfig p4p2 192.168.200.2/24 up
```

3. Have one ping the other. This is an example of the BlueField pinging the host:

```
$ ping 192.168.200.1
```

8.16.8.2 Verifying Connection from Host to BlueField

There are two SFs configured on the BlueFIeld-2 device, **enp3s0f0s0** and **enp3s0f1s0**, and their representors are part of the built-in bridge. These interfaces will get IP addresses from the DHCP server if it is present. Otherwise it is possible to configure IP address from the host. It is possible to access BlueField via the SF netdev interfaces.

For example:

1. Verify the default OVS configuration. Run:

```
# ovs-vsctl show
5668f9a6-6b93-49cf-a72a-14fd64b4c82b
    Bridge ovsbr1
        Port pf0hpf
            Interface pf0hpf
        Port ovsbr1
            Interface ovsbr1
                type: internal
        Port p0
            Interface p0
        Port en3f0pf0sf0
            Interface en3f0pf0sf0
    Bridge ovsbr2
        Port en3f1pf1sf0
            Interface en3f1pf1sf0
        Port ovsbr2
            Interface ovsbr2
                type: internal
        Port pf1hpf
            Interface pf1hpf
        Port p1
            Interface p1
    ovs_version: "2.14.1"
```

2. Verify whether the SF netdev received an IP address from the DHCP server. If not, assign a static IP. Run:

```
# ifconfig enp3s0f0s0
enp3s0f0s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.200.125  netmask 255.255.255.0  broadcast 192.168.200.255
        inet6 fe80::8e:bcff:fe36:19bc  prefixlen 64  scopeid 0x20<link>
        ether 02:8e:bc:36:19:bc  txqueuelen 1000  (Ethernet)
        RX packets 3730  bytes 1217558 (1.1 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 22  bytes 2220 (2.1 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

3. Verify the connection of the configured IP address. Run:

```
# ping 192.168.200.25 -c 5
PING 192.168.200.25 (192.168.200.25) 56(84) bytes of data.
64 bytes from 192.168.200.25: icmp_seq=1 ttl=64 time=0.228 ms
64 bytes from 192.168.200.25: icmp_seq=2 ttl=64 time=0.175 ms
64 bytes from 192.168.200.25: icmp_seq=3 ttl=64 time=0.232 ms
64 bytes from 192.168.200.25: icmp_seq=4 ttl=64 time=0.174 ms
64 bytes from 192.168.200.25: icmp_seq=5 ttl=64 time=0.168 ms

--- 192.168.200.25 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 91ms
rtt min/avg/max/mdev = 0.168/0.195/0.232/0.031 ms
```

8.16.8.3 Verifying Host Connection on Windows

Set IP address on the Windows side for the RShim or Physical network adapter, please run the following command in Command Prompt:

```
PS C:\Users\Administrator> New-NetIPAddress -InterfaceAlias "Ethernet 16" -IPAddress "192.168.100.1" -PrefixLength
22
```

To get the interface name, please run the following command in Command Prompt:

```
PS C:\Users\Administrator> Get-NetAdapter
```

Output should give us the interface name that matches the description (e.g. NVIDIA BlueField Management Network Adapter).

```
Ethernet 2          NVIDIA ConnectX-4 Lx Ethernet Adapter        6 Not Present  24-8A-07-0D-E8-1D
Ethernet 6          NVIDIA ConnectX-4 Lx Ethernet Ad...#2       23 Not Present  24-8A-07-0D-E8-1C
Ethernet 16         NVIDIA BlueField Management Netw...#2       15 Up           CA-FE-01-CA-FE-02
```

Once IP address is set, Have one ping the other.

```
C:\Windows\system32>ping 192.168.100.2


Pinging 192.168.100.2 with 32 bytes of data:
Reply from 192.168.100.2: bytes=32 time=148ms TTL=64
Reply from 192.168.100.2: bytes=32 time=152ms TTL=64
Reply from 192.168.100.2: bytes=32 time=158ms TTL=64
Reply from 192.168.100.2: bytes=32 time=158ms TTL=64
```

8.16.8.4 Enabling OVS HW Offloading

OVS HW offloading is set by default by the **/sbin/mlnx_bf_configure** script upon first boot after installation.

1. Enable TC offload on the relevant interfaces. Run:

   ```
   $ ethtool -K <PF> hw-tc-offload on
   ```

2. Enable the HW offload: run the following commands (after enabling the HW offload):

   ```
   $ ovs-vsctl set Open_vSwitch . Other_config:hw-offload=true
   ```

3. Restarting OVS is required for the configuration to apply:
   - For Ubuntu:

     ```
     $ systemctl restart openvswitch-switch
     ```

   - For CentOS/RHEL:

     ```
     $ systemctl restart openvswitch
     ```

To show OVS configuration:

```
$ ovs-dpctl show
system@ovs-system:
  lookups: hit:0 missed:0 lost:0
  flows: 0
  masks: hit:0 total:0 hit/pkt:0.00
  port 0: ovs-system (internal)
  port 1: armbr1 (internal)
  port 2: p0
  port 3: pf0hpf
  port 4: pf0vf0
  port 5: pf0vf1
  port 6: pf0vf2
```

At this point OVS would automatically try to offload all the rules.

To see all the rules that are added to the OVS datapath:

```
$ ovs-appctl dpctl/dump-flows
```

To see the rules that are offloaded to the HW:

```
$ ovs-appctl dpctl/dump-flows type=offloaded
```

8.16.8.5 Enabling OVS-DPDK Hardware Offload

1. Remove previously configured OVS bridges. Run:

   ```
   ovs-vsctl del-br <bridge-name>
   ```

   Issue the command **ovs-vsctl show** to see already configured OVS bridges.
2. Enable the Open vSwitch service. Run:

   ```
   systemctl start openvswitch
   ```
3. Configure huge pages:

   ```
   echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
   ```
4. Configure DPDK socket memory and limit. Run:

   ```
   # ovs-vsctl set Open_vSwitch . other_config:dpdk-socket-limit=2048
   # ovs-vsctl set Open_vSwitch . other_config:dpdk-socket-mem=2048
   ```
5. Enable hardware offload (disabled by default). Run:

   ```
   ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-init=true
   ovs-vsctl --no-wait set Open_vSwitch . other_config:hw-offload=true
   ```
6. Configure the DPDK whitelist. Run:

   ```
   ovs-vsctl set Open_vSwitch . other_config:dpdk-extra="-a
   0000:03:00.0,representor=[0,65535],dv_flow_en=1,dv_xmeta_en=1,sys_mem_en=1"
   ```
7. Create OVS-DPDK bridge. Run:

   ```
   ovs-vsctl add-br br0-ovs -- set Bridge br0-ovs datapath_type=netdev -- br-set-external-id br0-ovs bridge-id
   br0-ovs -- set bridge br0-ovs fail-mode=standalone
   ```
8. Add PF to OVS. Run:

```
ovs-vsctl add-port br0-ovs p0 -- set Interface p0  type=dpdk options:dpdk-devargs=0000:03:00.0
```

9. Add representor to OVS. Run:

```
ovs-vsctl add-port br0-ovs pf0vf0 -- set Interface pf0vf0 type=dpdk options:dpdk-
devargs=0000:03:00.0,representor=[0]
ovs-vsctl add-port br0-ovs pf0hpf -- set Interface pf0hpf type=dpdk options:dpdk-
devargs=0000:03:00.0,representor=[65535]
```

10. Restart the Open vSwitch service. This step is required for HW offload changes to take effect.
    - For CentOS, run:

      ```
      systemctl restart openvswitch
      ```

    - For Debian/Ubuntu, run:

      ```
      systemctl restart openvswitch-switch
      ```

8.16.8.6 Configuring DPDK and Running TestPMD

1. Configure hugepages. Run:

```
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

2. Run testpmd.
    - For Ubuntu/Debian:

      ```
      env LD_LIBRARY_PATH=/opt/mellanox/dpdk/lib/aarch64-linux-gnu /opt/mellanox/dpdk/bin/dpdk-testpmd -a
      03:00.0,representor=[0,65535] --socket-mem=1024 -- --total-num-mbufs=131000 -i
      ```

    - For CentOS:

      ```
      env LD_LIBRARY_PATH=/opt/mellanox/dpdk/lib64/ /opt/mellanox/dpdk/bin/dpdk-testpmd -a
      03:00.0,representor=[0,65535] --socket-mem=1024 -- --total-num-mbufs=131000 -i
      ```

For a detailed procedure with port display, refer to the article "Configuring DPDK and Running testpmd on BlueField-2".

8.16.8.7 Flow Statistics and Aging

The aging timeout of OVS is given in milliseconds and can be configured by running the following command:

```
$ ovs-vsctl set Open_vSwitch . other_config:max-idle=30000
```

8.16.8.8 Connection Tracking Offload

This feature enables tracking connections and storing information about the state of these connections. When used with OVS, BlueField can offload connection tracking, so that traffic of established connections bypasses the kernel and goes directly to hardware.

Both source NAT (SNAT) and destination NAT (DNAT) are supported with connection tracking offload.

8.16.8.8.1 Configuring Connection Tracking Offload

This section provides an example of configuring OVS to offload all IP connections of host PF0.

1. Enable OVS HW offloading.
2. Create OVS connection tracking bridge. Run:

```
$ ovs-vsctl add-br ctBr
```

3. Add p0 and pf0hpf to the bridge. Run:

```
$ ovs-vsctl add-port ctBr p0
$ ovs-vsctl add-port ctBr pf0hpf
```

4. Configure ARP packets to behave normally. Packets which do not comply are routed to table 1. Run:

```
$ ovs-ofctl add-flow ctBr "table=0,arp,action=normal"
$ ovs-ofctl add-flow ctBr "table=0,ip,ct_state=-trk,action=ct(table=1)"
```

5. Configure RoCEv2 packets to behave normally. RoCEv2 packets follow UDP port 4791 and a different source port in each direction of the connection. RoCE traffic is not supported by CT. In order to run RoCE from the host add the following line before **`ovs-ofctl add-flow ctBr "table=0,ip,ct_state=-trk,action=ct(table=1)"`** :

```
$ ovs-ofctl add-flow ctBr table=0,udp,tp_dst=4791,action=normal
```

   This rule allows RoCEv2 UDP packets to skip connection tracking rules.
6. Configure the new established flows to be admitted to the connection tracking bridge and to then behave normally. Run:

```
$ ovs-ofctl add-flow ctBr "table=1,priority=1,ip,ct_state=+trk+new,action=ct(commit),normal"
```

7. Set already established flows to behave normally. Run:

```
$ ovs-ofctl add-flow ctBr "table=1,priority=1,ip,ct_state=+trk+est,action=normal"
```

8.16.8.8.2 Connection Tracking With NAT

This section provides an example of configuring OVS to offload all IP connections of host PF0, and performing source network address translation (SNAT). The server host sends traffic via source IP from 2.2.2.1 to 1.1.1.2 on another host. Arm performs SNAT and changes the source IP to 1.1.1.16. Note that static ARP or route table must be configured to find that route.

1. Configure untracked IP packets to do nat. Run:

```
ovs-ofctl add-flow ctBr "table=0,ip,ct_state=-trk,action=ct(table=1,nat)"
```

2. Configure new established flows to do SNAT, and change source IP to 1.1.1.16. Run:

```
ovs-ofctl add-flow ctBr "table=1,in_port=pf0hpf,ip,ct_state=+trk+new,action=ct(commit,nat(src=1.1.1.16)),
p0"
```

3. Configure already established flows act normal. Run:

```
ovs-ofctl add-flow ctBr "table=1,ip,ct_state=+trk+est,action=normal"
```

Conntrack shows the connection with SNAT applied. Run **conntrack -L** for Ubuntu 22.04 kernel or **cat /proc/net/nf_conntrack** for older kernel versions. Example output:

```
ipv4     2 tcp      6 src=2.2.2.1 dst=1.1.1.2 sport=34541 dport=5001 src=1.1.1.2 dst=1.1.1.16 sport=5001
dport=34541 [OFFLOAD] mark=0 zone=1 use=3
```

8.16.8.8.3 Querying Connection Tracking Offload Status

Start traffic on PF0 from the server host (e.g., iperf) with an external network. Note that only established connections can be offloaded. TCP should have already finished the handshake, UDP should have gotten the reply.

> ICMP is not currently supported.

To check if specific connections are offloaded from Arm, run **conntrack -L** for Ubuntu 22.04 kernel or **cat /proc/net/nf_conntrack** for older kernel versions.

The following is example output of offloaded TCP connection:

```
ipv4     2 tcp      6 src=1.1.1.2 dst=1.1.1.3 sport=51888 dport=5001 src=1.1.1.3 dst=1.1.1.2 sport=5001 dport=51888
[HW_OFFLOAD] mark=0 zone=0 use=3
```

8.16.8.8.4 Performance Tune Based on Traffic Pattern

Offloaded flows (including connection tracking) are added to virtual switch FDB flow tables. FDB tables have a set of flow groups. Each flow group saves the same traffic pattern flows. For example, for connection tracking offloaded flow, TCP and UDP are different traffic patterns which end up in two different flow groups.

A flow group has a limited size to save flow entries. By default, the driver has 4 big FDB flow groups. Each of these big flow groups can save at most 4000000/(4+1)=800k different 5-tuple flow entries. For scenarios with more than 4 traffic patterns, the driver provides a module parameter (**num_of_groups**) to allow customization and performance tune.

> The size of each big flow groups can be calculated according to formula: size = 4000000/(num_of_groups+1)

To change the number of big FDB flow groups, run:

```
$ echo <num_of_groups> > /sys/module/mlx5_core/parameters/num_of_groups
```

The change takes effect immediately if there is no flow inside the FDB table (no traffic running and all offloaded flows are aged out), and it can be dynamically changed without reloading the driver.

If there are residual offloaded flows when changing this parameter, then the new configuration only takes effect after all flows age out.

8.16.8.8.5 Connection Tracking Aging

Aside from the aging of OVS, connection tracking offload has its own aging mechanism with a default aging time of 30 seconds.

8.16.8.8.6 Maximum Tracked Connections

⚠ The maximum number for tracked offloaded connections is limited to 1M by default.

The OS has a default setting of maximum tracked connections which may be configured by running:

```
$ /sbin/sysctl -w net.netfilter.nf_conntrack_max=1000000
```

This changes the maximum tracked connections (both offloaded and non-offloaded) setting to 1 million.

The following option specifies the limit on the number of offloaded connections. For example:

```
# devlink dev param set pci/${pci_dev} name ct_max_offloaded_conns value $max cmode runtime
```

This value is set to 1 million by default from BlueFiled. Users may choose a different number by using the **devlink** command.

⚠ Make sure **net.netfilter.nf_conntrack_tcp_be_liberal=1** when using connection tracking.

8.16.8.9 Offloading VLANs

OVS enables VF traffic to be tagged by the virtual switch.

For BlueField, the OVS can add VLAN tag (VLAN push) to all the packets sent by a network interface running on the host (either PF or VF) and strip the VLAN tag (VLAN pop) from the traffic going from the wire to that interface. Here we operate in Virtual Switch Tagging (VST) mode. This means that the host/VM interface is unaware of the VLAN tagging. Those rules can also be offloaded to the HW embedded switch.

To configure OVS to push/pop VLAN you need to add the tag=$TAG section for the OVS command line that adds the representor ports. So if you want to tag all the traffic of VF0 with VLAN ID 52, you should use the following command when adding its representor to the bridge:

```
$ ovs-vsctl add-port armbr1 pf0vf0 tag=52
```

If the virtual port is already connected to the bridge prior to configuring VLAN, you would need to remove it first:

```
$ ovs-vsctl del-port pf0vf0
```

In this scenario all the traffic being sent by VF 0 will have the same VLAN tag. We could set a VLAN tag by flow when using the TC interface, this is explained in section "Using TC Interface to Configure Offload Rules".

8.16.8.10 VXLAN Tunneling Offload

VXLAN tunnels are created on the Arm side and attached to the OVS. VXLAN decapsulation/encapsulation behavior is similar to normal VXLAN behavior, including over **hw_offload=true**.

To allow VXLAN encapsulation, the uplink representor (**p0**) should have an MTU value at least 50 bytes greater than that of the host PF/VF. Please refer to "Configuring Uplink MTU" for more information.

8.16.8.10.1 Configuring VXLAN Tunnel

1. Consider **p0** to be the local VXLAN tunnel interface (or VTEP).

   To be consistent with the examples below, it is assumed that **p0** is configured with a 1.1.1.1 IPv4 address.

2. Remove **p0** from any OVS bridge.
3. Build a VXLAN tunnel over OVS arm-ovs. Run:

```
ovs-vsctl add-br arm-ovs -- add-port arm-ovs vxlan11 -- set interface vxlan11 type=vxlan
options:local_ip=1.1.1.1 options:remote_ip=1.1.1.2 options:key=100
options:dst_port=4789
```

4. Connect any host representor (e.g., **pf0hpf**) for which VXLAN is desired to the same arm-ovs bridge.
5. Configure the MTU of the VTEP (**p0**) used by VXLAN to at least 50 bytes larger than the host representor's MTU.

At this point, the host is unaware of any VXLAN operations done by the BlueField's OVS. If the remote end of the VXLAN tunnel is properly set, any network traffic traversing arm-ovs undergoes VXLAN encap/decap.

8.16.8.10.2 Querying OVS VXLAN hw_offload Rules

Run the following:

```
ovs-appctl dpctl/dump-flows type=offloaded
in_port(2),eth(src=ae:fd:f3:31:7e:7b,dst=a2:fb:09:85:84:48),eth_type(0x0800), packets:1, bytes:98, used:0.900s,
actions:set(tunnel(tun_id=0x64,src=1.1.1.1,dst=1.1.1.2,tp_dst=4789,flags(key))),3
tunnel(tun_id=0x64,src=1.1.1.2,dst=1.1.1.1,tp_dst=4789,flags(+key)),in_port(3),eth(src=a2:fb:09:85:84:48,dst=ae:fd:
f3:31:7e:7b),eth_type(0x0800), packets:75, bytes:7350, used:0.900s, actions:2
```

For the host PF, in order for VXLAN to work properly with the default 1500 MTU, follow these steps.

1. Disable host PF as the port owner from Arm (see section "Zero-trust Mode"). Run:

```
$ mlxprivhost -d /dev/mst/mt41682_pciconf0 --disable_port_owner r
```

2. The MTU of the end points (**pf0hpf** in the example above) of the VXLAN tunnel must be smaller than the MTU of the tunnel interfaces (**p0**) to account for the size of the VXLAN headers. For example, you can set the MTU of P0 to 2000.

8.16.8.11 GRE Tunneling Offload

GRE tunnels are created on the Arm side and attached to the OVS. GRE decapsulation/ encapsulation behavior is similar to normal GRE behavior, including over **hw_offload=true** .

To allow GRE encapsulation, the uplink representor (**p0**) should have an MTU value at least 50 bytes greater than that of the host PF/VF.

Please refer to "Configuring Uplink MTU" for more information.

8.16.8.11.1 Configuring GRE Tunnel

1. Consider **p0** to be the local GRE tunnel interface. **p0** should not be attached to any OVS bridge.

   To be consistent with the examples below, it is assumed that **p0** is configured with a 1.1.1.1 IPv4 address and that the remote end of the tunnel is 1.1.1.2.

2. Create an OVS bridge, **br0** , with a GRE tunnel interface, **gre0** . Run:

```
ovs-vsctl add-port br0 gre0 -- set interface gre0 type=gre options:local_ip=1.1.1.1
options:remote_ip=1.1.1.2 options:key=100
```

3. Add **pf0hpf** to **br0** .

```
ovs-vsctl add-port br0 pf0hpf
```

4. At this point, any network traffic sent or received by the host's PF0 undergoes GRE processing inside the BlueField OS.

8.16.8.11.2 Querying OVS GRE hw_offload Rules

Run the following:

```
ovs-appctl dpctl/dump-flows type=offloaded
recirc_id(0),in_port(3),eth(src=50:6b:4b:2f:0b:74,dst=de:d0:a3:63:0b:30),eth_type(0x0800),ipv4(frag=no),
packets:878, bytes:122802, used:0.440s,
actions:set(tunnel(tun_id=0x64,src=1.1.1.1,dst=1.1.1.2,ttl=64,flags(key))),2
tunnel(tun_id=0x64,src=1.1.1.1,dst=1.1.1.2,flags(+key)),recirc_id(0),in_port(2),eth(src=de:d0:a3:63:0b:30,dst=50:6b
:4b:2f:0b:74),eth_type(0x0800),ipv4(frag=no), packets:995, bytes:97510, used:0.440s, actions:3
```

For the host PF, in order for GRE to work properly with the default 1500 MTU, follow these steps.

1. Disable host PF as the port owner from Arm (see section "Zero-trust Mode"). Run:

```
$ mlxprivhost -d /dev/mst/mt41682_pciconf0 --disable_port_owner r
```

2. The MTU of the end points (**pf0hpf** in the example above) of the GRE tunnel must be smaller than the MTU of the tunnel interfaces (**p0**) to account for the size of the GRE headers. For example, you can set the MTU of P0 to 2000.

8.16.8.12 GENEVE Tunneling Offload

GENEVE tunnels are created on the Arm side and attached to the OVS. GENEVE decapsulation/ encapsulation behavior is similar to normal GENEVE behavior, including over **hw_offload=true** .

To allow GENEVE encapsulation, the uplink representor (**p0**) must have an MTU value at least 50 bytes greater than that of the host PF/VF.

Please refer to "Configuring Uplink MTU" for more information.

8.16.8.12.1 Configuring GENEVE Tunnel

1. Consider **p0** to be the local GENEVE tunnel interface. **p0** should not be attached to any OVS bridge.
2. Create an OVS bridge, **br0** , with a GENEVE tunnel interface, **gnv0** . Run:

```
ovs-vsctl add-port br0 gnv0 -- set interface gnv0 type=geneve options:local_ip=1.1.1.1
options:remote_ip=1.1.1.2 options:key=100
```

3. Add **pf0hpf** to **br0** .

```
ovs-vsctl add-port br0 pf0hpf
```

4. At this point, any network traffic sent or received by the host's PF0 undergoes GENEVE processing inside the BlueField OS.

Options are supported for GENEVE. For example, you may add option **0xea55** to tunnel metadata, run:

```
ovs-ofctl add-tlv-map geneve_br "{class=0xffff,type=0x0,len=4}->tun_metadata0"
ovs-ofctl add-flow geneve_br ip,actions="set_field:0xea55->tun_metadata0",normal
```

For the host PF, in order for GENEVE to work properly with the default 1500 MTU, follow these steps.

1. Disable host PF as the port owner from Arm (see section "Zero-trust Mode"). Run:

```
$ mlxprivhost -d /dev/mst/mt41682_pciconf0 --disable_port_owner r
```

2. The MTU of the end points (**pf0hpf** in the example above) of the GENEVE tunnel must be smaller than the MTU of the tunnel interfaces (**p0**) to account for the size of the GENEVE headers. For example, you can set the MTU of P0 to 2000.

8.16.8.13 Using TC Interface to Configure Offload Rules

Offloading rules can also be added directly, and not just through OVS, using the tc utility. To enable TC ingress on all the representors (i.e., uplink, PF, and VF).

```
$ tc qdisc add dev p0 ingress
$ tc qdisc add dev pf0hpf ingress
$ tc qdisc add dev pf0vf0 ingress
```

8.16.8.13.1 L2 Rules Example

The rule below drops all packets matching the given source and destination MAC addresses.

```
$ tc filter add dev pf0hpf protocol ip parent ffff: \
                                flower \
                                            skip_sw \
                                            dst_mac e4:11:22:11:4a:51 \
                                            src_mac e4:11:22:11:4a:50 \
                                 action drop
```

8.16.8.13.2 VLAN Rules Example

The following rules push VLAN ID 100 to packets sent from VF0 to the wire (and forward it through the uplink representor) and strip the VLAN when sending the packet to the VF.

```
$ tc filter add dev pf0vf0 protocol 802.1Q parent ffff: \
                                    flower \
                                              skip_sw \
                                              dst_mac e4:11:22:11:4a:51 \
                                              src_mac e4:11:22:11:4a:50 \
                                    action vlan push id 100 \
                                    action mirred egress redirect dev p0

$ tc filter add dev p0 protocol 802.1Q parent ffff: \
                                    flower \
                                             skip_sw \
                                             dst_mac e4:11:22:11:4a:51 \
                                             src_mac e4:11:22:11:4a:50 \
                                             vlan_ethtype 0x800 \
                                             vlan_id 100 \
                                             vlan_prio 0 \
                                  action vlan pop \
                                  action mirred egress redirect dev pf0vf0
```

8.16.8.13.3 VXLAN Encap/Decap Example

```
$ tc filter add dev pf0vf0 protocol 0x806 parent ffff: \
                                    flower \
                                              skip_sw \
                                              dst_mac e4:11:22:11:4a:51 \
                                              src_mac e4:11:22:11:4a:50 \
                                    action tunnel_key set \
                                    src_ip 20.1.12.1 \
                                    dst_ip 20.1.11.1 \
                                    id 100 \
                                    action mirred egress redirect dev vxlan100

$ tc filter add dev vxlan100 protocol 0x806 parent ffff: \
                                    flower \
                                             skip_sw \
                                             dst_mac e4:11:22:11:4a:51 \
                                             src_mac e4:11:22:11:4a:50 \
                                             enc_src_ip 20.1.11.1 \
                                             enc_dst_ip 20.1.12.1 \
```

```
                                enc_key_id 100 \
                                enc_dst_port 4789 \
        action tunnel_key unset \
        action mirred egress redirect dev pf0vf0
```

8.16.8.14 Virtio Acceleration Through Hardware vDPA

For configuration procedure, please refer to the MLNX_OFED documentation under OVS Offload Using ASAP² Direct > Virtio Acceleration through Hardware vDPA.

8.16.9 OVS Live Upgrade

This document describes the procedure for performing a live upgrade of Open vSwitch (OVS) with NVIDIA DOCA datapath support, enabling seamless transitions with minimal or no network traffic disruption.

8.16.9.1 Introduction

Non-disruptive upgrade (NDU), also known as a live or hot upgrade, replaces the OVS user-space process without disrupting active network traffic. It achieves this by synchronizing runtime state between the running process and a newly launched one, enabling a seamless handoff. NDU supports high-availability requirements and can also be used to restart OVS after configuration changes, ensuring continuous operation without downtime or packet loss.

8.16.9.2 NDU Operational Model

The NDU procedure involves two cooperating OVS processes:

- Server – the currently running OVS process
- Client – the new OVS process launched for the upgrade

The upgrade sequence consists of the following steps:

1. Client initialization – The new OVS process is launched in client mode.
2. State Synchronization – The client connects to the server and synchronizes essential runtime state, including:
    - Datapath and OpenFlow ports
    - Tunnel neighbor entries
    - OpenFlow tables, rules, meters, groups, and TLV mappings
    - Connection tracking (CT) state
    - MAC learning tables (FDB)
    - Routing entries

    > ⚠ Datapath rules (e.g., megaflows) are not transferred; the new process relearns them upon takeover.
3. Control plane handover – Upon successful synchronization, the client assumes control, and the server process is gracefully terminated.

Throughout the process, packet forwarding continues with negligible disruption.

8.16.9.3 Prerequisites

To support live upgrade:

- Both the existing and new OVS processes must support NDU.
- The datapath type must be set to `doca`.

8.16.9.4 Upgrade Procedure

8.16.9.4.1 Initiating the Upgrade

No manual intervention is required during the upgrade process. To initiate an NDU, reload the OVS systemd service:

```
initiate live upgrade
```

```
systemctl reload ovs-vswitchd.service
```

This command launches the client process, triggers synchronization, and transitions control to the new instance. The old process exits cleanly after handover.

Progress and any anomalies are logged to the OVS log files.

8.16.9.4.2 Post-Upgrade Validation

After the upgrade, confirm that:

- The new OVS process is active and managing the datapath
- All ports, flow rules, and dynamic state have been retained
- No traffic interruption occurred during the upgrade

Use standard OVS utilities to verify status:

```
ovs-vsctl show
ovs-ofctl dump-flows <bridge>
ovs-appctl dpctl/offload-stats-show
ovs-appctl dpctl/dump-flows
```

8.16.9.4.3 Error Handling and Rollback

If an error occurs during the synchronization or switchover phase, OVS will automatically revert to the original process to maintain service continuity. All rollback operations and failures are recorded in the log.

> ⚠ On Debian-based distributions (including Ubuntu), the `dpkg` package manager currently restarts OVS during `.deb` package upgrades. This behavior conflicts with the NDU mechanism. Once the installer behavior is updated, this limitation will be removed.

8.17 DOCA XPlane Service Guide

This guide provides instructions on how to use the DOCA XPlane Service on top of hosts equipped with NVIDIA® ConnectX-8 SuperNICs.

8.17.1 1. Introduction

DOCA XPlane is a background daemon designed to manage local and remote plane failover within OVS DOCA. It actively consumes route-availability updates from an NVIDIA® Spectrum-X fabric, dynamically programs OpenFlow rules to facilitate traffic failover, and exposes comprehensive gRPC telemetry for monitoring service health and topology state.

8.17.2 2. Requirements

To utilize the DOCA XPlane service, your environment must meet the following hardware and firmware prerequisites:

- Hardware: A single-port NVIDIA® ConnectX®-8 SuperNIC.
- Firmware version: Version **40.48.1000** or higher.

8.17.2.1 2.1. Configuring Firmware Settings on the SuperNIC

Before deploying the service, you must enable multi-plane mode, configure the number of planes, and set the number of Physical Functions (PFs) to match that number on each SuperNIC.

1. Execute the following **mlxconfig** commands to configure the device.

```
mlxconfig -d <mst-device> -y set SRIOV_EN=1
mlxconfig -d <mst-device> -y set MODULE_SPLIT_M0[0..7]=1 MODULE_SPLIT_M0[8..15]=FF
mlxconfig -d <mst-device> -y set LAG_RESOURCE_ALLOCATION=1
mlxconfig -d <mst-device> -y set LINK_TYPE_P1=2
mlxconfig -d <mst-device> -y set LOAD_BALANCE_MODE_P1=2
mlxconfig -d <mst-device> -y set NUM_OF_PF=4 NUM_OF_VFS=4 NUM_OF_PLANES_P1=4
```

2. Apply the configuration by performing a graceful shutdown and power cycle of the host.

8.17.2.2 2.2. Setting devlink Parameters

While configuring **devlink** parameters, the OVS-DOCA service must be stopped.

For each SuperNIC managed by DOCA XPlane:

1. Get PCI address of each PF. One easy way to do that is by using **lshw**:

```
root@server:~# lshw -class net -businfo
Bus info          Device          Class          Description
========================================================
...
pci@0000:9f:00.0  enp159s0f0np0   network        CX8 Family [ConnectX-8]
pci@0000:9f:00.1  enp159s0f1np1   network        CX8 Family [ConnectX-8]
pci@0000:9f:00.2  enp159s0f2np2   network        CX8 Family [ConnectX-8]
pci@0000:9f:00.3  enp159s0f3np3   network        CX8 Family [ConnectX-8]
...
```

2. For each PF in that SuperNIC, set **flow_steering_mode** to **hmfs**:

```
devlink dev param set pci/0000:9f:00.0 name flow_steering_mode cmode runtime value hmfs
devlink dev param set pci/0000:9f:00.1 name flow_steering_mode cmode runtime value hmfs
devlink dev param set pci/0000:9f:00.2 name flow_steering_mode cmode runtime value hmfs
devlink dev param set pci/0000:9f:00.3 name flow_steering_mode cmode runtime value hmfs
```

3. For each PF in that SuperNIC, set **eswitch** **mode** to **switchdev**:

```
devlink dev eswitch set pci/0000:9f:00.0 mode switchdev
devlink dev eswitch set pci/0000:9f:00.1 mode switchdev
devlink dev eswitch set pci/0000:9f:00.2 mode switchdev
devlink dev eswitch set pci/0000:9f:00.3 mode switchdev
```

4. For each PF of that SuperNIC, set **esw_multiport** to **true**:

```
sudo devlink dev param set pci/0000:9f:00.0 name esw_multiport value true cmode runtime
sudo devlink dev param set pci/0000:9f:00.1 name esw_multiport value true cmode runtime
sudo devlink dev param set pci/0000:9f:00.2 name esw_multiport value true cmode runtime
sudo devlink dev param set pci/0000:9f:00.3 name esw_multiport value true cmode runtime
```

8.17.2.3 2.3. Creating SR-IOV Virtual Functions

For each SuperNIC managed by DOCA XPlane, create one VF on the first PF of the SuperNIC.

```
echo 1 > /sys/class/net/enp159s0f0np0/device/sriov_numvfs
```

For configuration details, refer to the community guide HowTo Configure SR-IOV for ConnectX-4/ ConnectX- 5/ConnectX-6 with KVM (Ethernet).

8.17.2.4 2.4. Configuring hugepages

1. Use the doca-hugepages tool to provision at least 150 hugepages of 2048KB per SuperNIC managed by DOCA XPlane.
   For instance, for a server equipped with 4 NVIDIA® ConnectX-8 SuperNICs, run:

```
doca-hugepages config --app doca-xplane --size 2048 --num 600
doca-hugepages reload
```

2. Restart OVS-DOCA for it to use the newly provisioned hugepages:

```
# Debian and derivatives
systemctl restart openvswitch-switch

# RHEL and derivatives
systemctl restart openvswitch
```

8.17.3 3. Service Deployment

8.17.3.1 3.1. XPlane service

To deploy the DOCA XPlane service, follow these steps:

1. Download and install the appropriate package for your system: Ubuntu (deb) / Enterprise Linux (RPM)
2. Once installed, populate the configuration file as detailed in the "Configuration File" section.

3. After saving your configuration, start the XPlane daemon by running the following command:

```
systemctl start doca-xplane
```

8.17.3.2 3.2. CLI client

DOCA XPlane ships with a dedicated CLI client (**doca-xplane-client**) that retrieves topology and observability data via gRPC.
To use the client, download and install the appropriate package for your system:

- Ubuntu amd64 / arm64
- Enterprise Linux x86_64 / aarch64

8.17.4 4. Configuration File

Prior to starting the service, you must populate the configuration file located at **/opt/mellanox/doca/services/xplane/xplane.toml** with the list of interfaces to be used.

The table below details the user-configurable settings and their purposes:

Setting Name	Description	Example Value
capabilities.spcx	Reserved for future use. This has no effect in the current release.	`true`
capabilities.remote_failures	Enables dynamic installation of remote failure rules based on LLDP route availability updates.	`true`
capabilities.mp_pipeline_init	Enables the setup of the pipeline and static OpenFlow rules for plane-aware traffic steering and gRPC metrics.	`true`
br_xplane_interfaces.mtu	The MTU to be applied to all PFs managed by XPlane. ℹ The MTU reported by **ip link** will be equal to the configured value plus 50 bytes.	`9216`
br_xplane_interfaces.groups[].group	The group identifier for a set of PFs and an optional VF. Each group maps related interfaces that XPlane should manage together.	`"rail0"`
br_xplane_interfaces.groups[].pfs	A list of PF netdev names to attach to the **br-xplane** bridge for this specific group.	`["eth_p0_rail0", "eth_p1_rail0", "eth_p2_rail0", "eth_p3_rail0"]`
br_xplane_interfaces.groups[].vf_rep	The name of the VF representor interface to be used for the group.	`"eth_vf0_rail0"`

Setting Name	Description	Example Value
`logging.level`	The service and setup logging level. Supported values are `debug`, `info`, `warn`, and `error`.	`"info"`

8.17.5 5. Maintenance mode

To enable or disable maintenance mode on PFs, use DOCA XPlane's CLI client.

> ⚠ Maintenance mode leverages DOCA XPlane's gRPC API, which listens on `127.0.0.1` but does not support neither TLS nor client authentication.
> Use with caution and apply appropriate mitigations to prevent unauthorized access.

To enable maintenance mode on a PF:

```
doca-xplane-client maintenance set-link-maintenance --pf_name <pf> --enabled true
```

To disable maintenance mode on a PF:

```
doca-xplane-client maintenance set-link-maintenance --pf_name <pf> --enabled false
```

To get the maintenance status of a PF, run:

```
doca-xplane-client maintenance get-link-maintenance --pf_name <pf>
```

JSON response:

```
{
  "enabled": false
}
```

8.17.6 6. Troubleshooting

8.17.6.1 6.1. Checking System Status

Verify the current operational status of the DOCA XPlane daemon by running the following command:

```
systemctl status doca-xplane
```

Example output:

```
● doca-xplane.service - DOCA XPlane
     Loaded: loaded (/lib/systemd/system/doca-xplane.service; enabled; vendor preset: enabled)
     Active: active (running) since Thu 2026-04-23 11:40:08 IDT; 12h ago
   Main PID: 71852 (docker)
...
```

8.17.6.2 6.2. Service Logs

DOCA XPlane service logs are permanently stored and can be reviewed at `/var/log/xplane/`.

8.17.6.3 6.3. Using `doca-xplane-client`

The following are examples of how to query using the client (outputs have been trimmed for brevity).

8.17.6.3.1 6.3.1. Retrieve Overall System Status

Command:

```
$ doca-xplane-client get-status
```

JSON response:

```
{
  "serviceName": "xplane",
  "version": "0.1.0",
  "process": {
    "state": "RUNNING",
    "uptimeSeconds": "86400"
  },
  "topologySummary": {
    "groups": 2,
    "planes": 2
  },
  "health": {
    "planesUp": [1],
    "planesDegraded": [0],
    "overallState": "DEGRADED",
    "failures": {
      "localTotal": 1,
      "remoteTotal": 2
    }
  }
}
```

8.17.6.3.2 6.3.2. Print Topology as Read by DOCA XPlane Agent

Command:

```
$ doca-xplane-client get-topology
```

JSON response:

```
{
  "pfs": [
    {
      "plane": 0,
      "group": "rail0nic0",
      "pfName": "eth2",
      "pciAddress": "0000:08:00.0",
      "speedGbps": 200
    }
  ],
  "vfs": [
    {
      "vfRep": "eth2",
      "group": "rail0"
    }
  ],
  "patchPorts": []
}
```

8.17.6.3.3 6.3.3. Get Summary of Configured Planes

Command:

```
$ doca-xplane-client get-planes-summary
```

JSON response:

```
{
  "planes": [
    {
      "plane": 0,
      "state": "DEGRADED",
      "failures": {
        "localTotal": 1,
        "remoteTotal": 2
      },
      "traffic": {
        "packets": "1250000",
        "bytes": "900000000"
      }
    }
  ]
}
```

8.17.6.3.4 6.3.4. Get Local Failures (e.g., link down) for Specific Plane

Command:

```
$ doca-xplane-client get-plane-failures-local --plane_id 0
```

JSON response:

```
{
  "localFailures": [
    {
      "group": "rail0-nic0",
      "pfName": "eth2"
    }
  ]
}
```

8.17.6.3.5 6.3.5. Get Remote Failures for Specific Plane

Command:

```
$ doca-xplane-client get-plane-failures-remote --plane_id 0
```

JSON response:

```
{
  "remoteFailures": [
     {
      "cidr": "10.10.0.0/24",
      "groups": [
      ]
    },
    {
      "cidr": "2001:db8:10::/64",
      "groups": [
      ]
    },
     {
      "cidr": "0.0.0.0/0",
      "groups": [
        "2",
        "3"
      ]
    },
```

```
  ]
}
```

8.17.6.3.6 6.3.6. Get Traffic Summary for Specific Plane

Command:

```
$ doca-xplane-client get-plane-traffic-summary --plane_id 0
```

JSON response:

```
{
  "nonDiverted": {
    "packets": "1200000",
    "bytes": "864000000"
  },
  "divertedTo": {
    "packets": "50000",
    "bytes": "36000000"
  },
  "dropped": {
    "packets": "0",
    "bytes": "0"
  }
}
```

8.17.6.3.7 6.3.7. Get Statistics for Traffic Diverted from One Plane to Another

To get a complete picture of failover routing, you can query traffic diverted both from and to a specific plane:

Command for traffic diverted from the plane:

```
$ doca-xplane-client get-plane-traffic-diverted-from --plane_id 0
```

JSON response:

```
{
  "aggregated": {
    "packets": "75000",
    "bytes": "54000000"
  },
  "perPlane": {
    "1": {
      "packets": "75000",
      "bytes": "54000000",
      "percentage": 100
    }
  }
}
```

Command for traffic diverted to the plane:

```
$ doca-xplane-client get-plane-traffic-diverted-to --plane_id 0
```

JSON response:

```
{
  "aggregated": {
    "packets": "50000",
    "bytes": "36000000"
  },
  "perPlane": {
    "1": {
      "packets": "50000",
      "bytes": "36000000",
      "percentage": 100
```

```
        }
    }
}
```

9 API References

For DOCA SDK API documentation, please refer to the API References for DOCA 3.4.0.

10 DOCA Troubleshooting

DOCA troubleshooting topics are available in the DOCA page of the *NVIDIA BlueField Platform Software Troubleshooting Guide*.

10.1 NVIDIA BlueField Reset and Reboot Procedures

10.1.1 BlueField System Reboot

This section describes the necessary operations to load new NIC firmware, following NVIDIA® BlueField® NIC firmware update. This procedure deprecates the need for full server power cycle.

The following steps are executed in the BlueField OS:

1. Issue a query command to ascertain whether BlueField system reboot is supported by your environment:

   ```
   mlxfwreset -d 03:00.0 q
   ```

   If the output includes the following lines, proceed to step 2:

   ```
   3: Driver restart and PCI reset                                -Supported (default)
   ...
   1: Driver is the owner                                         -Supported (default)
   ```

   > ⚠ If it says **Not Supported** instead, then proceed to the instructions under section "BlueField System-level Reset".

2. Issue a BlueField system reboot:

   ```
   mlxfwreset -d 03:00.0 -y -l 3 --sync 1 r
   ```

10.1.2 BlueField System-level Reset

This section describes the way to perform system-level reset (SLR) which is necessary for firmware configuration changes to take effect.

- SLR for BlueField running in DPU mode
- SLR for BlueField running in NIC mode
- SLR for BlueField running in DPU mode on hosts with separate power control (special use case)

10.1.2.1 System-level Reset for BlueField in DPU Mode

The following is the high-level flow of the procedure:

1. Graceful shutdown of BlueField Arm cores.
2. Query BlueField state to affirm shutdown reached.

> In systems with multiple BlueField networking platforms, repeat steps 1 and 2 for all devices before proceeding.

3. Warm reboot the server.

Step by step process:

> Some of the following steps can be performed using different methods, depending on resource availability and support in the user's environment.

1. Graceful shutdown of BlueField Arm cores.

   > This operation is expected to finish within 15 seconds.

   Possible methods:

   - From the BlueField OS:

     ```
     shutdown -h now
     ```

     Or:

     ```
     mlxfwreset -d /dev/mst/mt*pciconf0 -l 1 -t 4 r
     ```

   - From the host OS:

     > Not relevant when the BlueField is operating in Zero-Trust Mode.

     ```
     mlxfwreset -d <mst-device> -l 1 -t 4 r
     ```

   - Using the BlueField BMC:

     ```
     ipmitool -C 17 -I lanplus -H <bmc_ip> -U root -P <password> power soft
     ```

     Or using Redfish (BlueField-3 and above):

     ```
     curl -k -u root:<password> -H "Content-Type: application/json" -X POST https://<bmc_ip>/redfish/v1/
     Systems/Bluefield/Actions/ComputerSystem.Reset -d '{"ResetType": "GracefulShutdown"}'
     ```

2. Query BlueField state.

   Possible methods:

   - From the host OS:

     > Not relevant when the BlueField is operating in Zero-Trust Mode.

     ```
     echo DISPLAY_LEVEL 2 > /dev/rshim0/misc
     cat /dev/rshim0/misc
     ```

     Expected output:

```
INFO[BL31]: System Off
```

- Utilizing the BlueField BMC:

```
ipmitool -C 17 -I lanplus -H <bmc_ip> -U root -P <password> raw 0x32 0xA3
```

Expected output: **06**.

3. Warm reboot the server from the host OS:

```
mlxfwreset -d <mst-device> -l 4 r
```

> ⚠ If multiple BlueField devices are present in the host, this command must run only once. In this case, the MST device can be of any of the BlueFields for which the reset is necessary and participated in step 1.

Or:

```
reboot
```

> ⚠ For external hosts which do not toggle PERST# in their standard reboot command, use the **mlxfwreset** option.

10.1.2.2 System-level Reset for BlueField in NIC Mode

Perform warm reboot of the host OS:

```
mlxfwreset -d <mst-device> -l 4 r
```

Or:

```
reboot
```

> ⚠ For external hosts which do not toggle PERST# in their standard reboot command, use the **mlxfwreset** option.

10.1.2.3 System-level Reset for Host with Separate Power Control

This procedure is a special use case relevant only to host platforms with separate power control for the PCIe slot and CPUs, in which the BlueField (running in DPU mode) is provided power while host OS/CPUs may be in shutdown or similar standby state (this allows the BlueField device to be operational while the host CPU is in shutdown/standby state).

The following is the high-level flow of the procedure:

1. Graceful shutdown of host OS or similar CPU standby.
2. Graceful shutdown of BlueField Arm cores.

3. Query BlueField state to affirm shutdown reached.
4. Full BlueField Reset
5. Query BlueField state to affirm operational state reached

   > In systems with multiple BlueField networking platforms, repeat steps 1 through 5 for all devices before proceeding.

6. Power on the server.

Step by step process:

> Some of the following steps can be performed using different methods, depending on resource availability and support in the user's environment.

1. Graceful shutdown of host OS by any means preferable.
2. Graceful shutdown of BlueField Arm cores.

   > This step normally takes up to 15 seconds to complete.

   - From the BlueField OS:

     ```
     shutdown -h now
     ```

   - Utilizing the BlueField BMC:
     - Using IPMI:

       ```
       ipmitool -C 17 -I lanplus -H <bmc_ip> -U root -P <password> power soft
       ```

     - Using Redfish (for BlueField-3 and above):

       ```
       curl -k -u root:<password> -H "Content-Type: application/json" -X POST https://<bmc_ip>/
       redfish/v1/Systems/Bluefield/Actions/ComputerSystem.Reset -d '{"ResetType":
       "GracefulShutdown"}'
       ```

3. Query the BlueField's state utilizing the BlueField BMC:

   ```
   ipmitool -C 17 -I lanplus -H <bmc_ip> -U root -P <password> raw 0x32 0xA3
   ```

   Expected output: **06**.
4. Perform BlueField hard reset utilizing the BlueField BMC:

   > This step takes up to 2 minutes to complete.

   - Using IPMI:

     ```
     ipmitool -C 17 -I lanplus -H <bmc_ip> -U root -P <password> power cycle
     ```

   - Using Redfish (for BlueField-3 and above):

     ```
     curl -k -u root:<password> -H "Content-Type: application/json" -X POST https://<bmc_ip>/redfish/v1/
     Systems/Bluefield/Actions/ComputerSystem.Reset -d '{"ResetType" : "PowerCycle"}'
     ```

5. Query BlueField operational state utilizing the BlueField BMC:

   > At this point, the BlueField is expected to be operational.

   ```
   ipmitool -C 17 -I lanplus -H <bmc_ip> -U root -P <password> raw 0x32 0xA3
   ```

   Expected output: **05**.
6. Power on/boot up the host OS.

10.1.3 BlueField DPU Reset Using a BMC Platform

The BlueField DPU can also be reset from a BMC platform using NC-SI command over I2C. This option is more common in DPU BMC absence, when the BlueField DPU is running in NIC mode or when it is used as a controller.

The reset is performed using the *Reset BlueField DPU (Command=0x12, Parameter=0xB)*, which allows a BMC platform to reset the NVIDIA BlueField DPU device. This command is only applicable to BlueField-2 devices.

The Reset BlueField-3 DPU command is addressed to the package only. When the internal reset is complete, the BMC platform should reconfigure the device.

> The *Reset BlueField DPU* command is supported on BlueField-2 and later devices.

Reset BlueField DPU Format

Bytes/Bits	31:24	23:16	15:8	7:0	
0:15	NC-SI Header (OEM Command)				
16:19	NVIDIA Manufacture ID (IANA) = 0x8119				
20:23	Command rev=0x00	MLNX Cmd ID=0x12	Parameter=0x0B	NICR	Mode
24:27	Checksum 31:0				

The parameter descriptions for Reset BlueField DPU command are provided below.

Reset BlueField DPU Parameters

Field	Description
NICR	• 0 - NIC does not reset. Only the embedded CPU will reset. • 1 - Reset the embedded CPU and the NIC
Mode	This field defines the type of conditions to use before performing the internal reset • 0 - The internal reset will start after sending the response to this command • 1 - The internal reset will start only when all the hosts asserts their PERST# signals low • 2 - The internal reset will start only when all the hosts disabled their PCIe links. This may or may not include assertion of their respective PERST# signals low. • Other - Reserved

Reset BlueField DPU Response

The ConnectX adapter responds to a Reset BlueField DPU command when the package ID matches, and with no checksum error.

Reset BlueField DPU Response Format

Bytes/Bits	31:24	23:16	15:8	7:0	
0:15	NC-SI Header (OEM Command)				
16:19	Response Code		Reason Code		
20:23	NVIDIA Manufacture ID (IANA) = 0x8119				
24:27	Command rev=0x00	MLNX Cmd ID=0x12	Parameter=0x0B	NICR	Mode
28:31	Checksum 31:0				

10.1.4 Shutdown BlueField DPU OS (Command=0x12, Parameter=0x1A)

The Shutdown BlueField DPU OS command allows trusted platform bmc to send an OS Shutdown request to the embedded CPU on NVIDIA BlueField DPU Devices.

The Shutdown BlueField DPU OS command format is shown below.

Shutdown BlueField DPU OS Command Format

Bytes/Bits	31:24	23:16	15:8	7:0
0:15	NC-SI Header (OEM Command)			
16:19	NVIDIA Manufacture ID (IANA) = 0x8119			
20:23	Command rev=0x00	MLNX Cmd ID=0x12	Parameter=0x1A	Reserved
24:27	Reserved			
28:31	Checksum 31:0			

This command has no input parameters. This command is a package command.

10.1.4.1 Shutdown BlueField DPU OS Response

NVIDIA BlueField DPU always receive and respond to Shutdown BlueField DPU OS command on supporting devices when the package ID matches. When the Shutdown BlueField DPU OS command is sent to a non-supporting device, or the command is received from an untrusted platform bmc, the command will fail with reason code 0x7FFF (Unsupported command).

Shutdown BlueField DPU OS Response Format

Bytes/Bits	31:24	23:16	15:8	7:0
0:15	NC-SI Header (OEM Command)			
16:19	Response Code		Reason Code	
20:23	NVIDIA Manufacture ID (IANA) = 0x8119			
24:27	Command rev=0x00	MLNX Cmd ID=0x12	Parameter=0x1A	Reserved

Bytes/Bits	31:24	23:16	15:8	7:0
28:31	Reserved			
32:35	Checksum 31:0			

11 Miscellaneous (Runtime)

This section contains the following pages:

- DOCA Glossary
- DOCA Crypto Acceleration
- DOCA Services Fluent Logger
- DOCA DPU CLI
- DOCA Switching
- DOCA with OpenSSL
- BlueField Scalable Functions
- DOCA TLS Offload
- DOCA Virtual Functions
- BlueField SR-IOV
- fTPM over OP-TEE

11.1 DOCA Glossary

The table below provides an extensive glossary of terms, abbreviations, and acronyms used throughout the DOCA SDK documentation.

Term/ Acronym	Description/Definition
b	Lower-case b is used to indicate size in bits or multiples of bits (e.g., 1Kb = 1024 bits)
B	Upper-case B is used to indicate size in bytes or multiples of bytes (e.g., 1KB = 1024 bytes)
ACS	Access control services
ASAP 2	Accelerated Switching and Packet Processing
ASN	Autonomous system number
ATF	Arm-trusted firmware
BAR	Base address register
BDF address	Bus, device, function address. The device's PCIe bus address.
BFB	BlueField bootstream
BGP	Border gateway protocol
BMC	Board management controller
BSP	BlueField support package
BUF	Buffer
CBS	Committed burst size
CIR	Committed information rate
CMDQ	Command queue
CPDS	Control pipe dynamic size
CQE	Completion queue events

Term/ Acronym	Description/Definition
CTX	Context
DEK	Data encryption key
DMA	Direct memory access
DN	Data network
DOCA	DPU SDK (Data Processing Unit Software Development Kit)
DPA	Data path accelerator; an auxiliary processor designed to accelerate data-path operations.
DPCP	Direct packet control plane
DPDK	Data plane development kit
DPI	Deep packet inspection
DPU	Data processing unit, the third pillar of the data center with CPU and GPU.
DW	Dword
EBS	Excess burst size
ECE	Enhanced connection establishment
ECMP	Equal-cost multi-path
ECPF	Embedded CPU physical function
EIR	Excess information rate
EM	Exact match
eMMC	Embedded multi-media card
ESP	Encapsulating security payload (or EFI system partition)
EU	Execution unit. HW thread; a logical DPA processing unit.
FAR	Forwarding action rule
FIFO	First-in-first-out
FIPS	Federal Information Processing Standards
FLR	Function level reset
FPGA	Field-programmable gate arrays
FW	Firmware
GDAKIN	GPUDirect async kernel-initiated network
GDB	GNU debugger
gNB	Next Generation NodeB
HCA	Host-channel adapter – an InfiniBand (IB) device that terminates an IB link.
Host	When used alone, refers to the server host. Arm host refers to the AARCH64 Linux OS running on the BlueField Arm cores.
HW	Hardware
hwmon	Hardware monitoring
IB	InfiniBand

Term/ Acronym	Description/Definition
ICM	Interface configuration memory
ICV	Integrity check value
IDE	Integrated development environment
IKE	Internet key exchange
In-band	Administration activities traversing the IB connectivity only.
IPoIB	IP over InfiniBand
IR	Intermediate representation
IRQ	Interrupt request
iSER	iSCSI RDMA protocol
KPI	Key performance indicator
LFT	Unicast linear forwarding tables
LRO	Large receive offload
LSB	Least significant byte (or least significant bit as *lsb*)
LSO	Large send offload
LTO	Link-time optimization
MFT	Mellanox firmware tools
MLNX_OFED	Mellanox OpenFabrics Enterprise Distribution
MPI	Message passing interface
MSB	Most significant byte (or most significant bit as *msb*)
MSS	Maximum segment size (or Memory subsystem)
MST	Mellanox software tools
MTU	Maximum transmission unit
NAT	Network address translation
NIC	Network interface card
NIST	National Institute of Standards and Technology
NP	Notification point
NS	Namespace
NUMA	Non-uniform memory access
OOB	Out-of-band
OS	Operating system
OVS	Open vSwitch
PAT	Port address translation
PBA	Pending bit array
PBS	Peak burst size
PCIe	PCI Express
PDR	Packet detection rule

Term/ Acronym	Description/Definition
PE	Progress engine
PF	Physical function
PFC	Priority flow control
PHC	Physical hardware clock
PIR	Peak information rate
PK	Platform key
PKA	Public key accelerator
POC	Proof of concept
PR	Path record
PUD	Process under debug
QER	QoS enforcement rule
QoS	Quality of service
RAN	Radio access network
RAP	Reference application
RD	Route distinguisher
RDMA	Remote direct memory access
RDMA CM	RDMA connection manager
RegEx	Regular expression
RN	Request node (e.g., RN-F: Fully coherent, RN-D: IO coherent)
RNG	Random number generator/generation
RoCE	RDMA over converged Ethernet
RP	Reaction point
RQ	Receive queue
RShim	Random shim
RSP	Remote serial protocol
RT	Route target
RTOS	Real-time operating system
RTT	Round-trip time
RX	Receive
RXP	Regular expression processor
SA	Security association (or Subnet administrator)
SBSA	Server base system architecture
SDK	Software development kit
SF	Sub-function or Scalable function
SFC	Service function chaining (in HBN)
SG	Scatter-gather

Term/ Acronym	Description/Definition
SHA	Secure hash algorithm
SL	Service level
SM	Subnet Manager
SMF	Session management function
SNAP	Storage-defined network-accelerated processing
SNAT	Source NAT
SPDK	Storage performance development kit
SPI	Security parameters index
SQ	Send queue
SR-IOV	Single-root IO virtualization
SRP	SCSI RDMA protocol
SuperNIC	A configuration of a DPU specific for E-W (East-West) networking.
SVI	Switch virtual interface
SW	Software
Sync event	Synchronization event
TAI	International Atomic Time
TIR	Transport interface receive
TIS	Transport interface send
TLS	Transport layer security
TSO	TCP segmentation offload
TX	Transmit
uDAPL	User direct access programming library
UDS	Unix domain socket
UE	User equipment
UEFI	Unified extensible firmware interface
ULP	Upper layer protocol
UPF	User-plane function
URR	Usage reporting rule
UTC	Coordinated Universal Time
vDPA	Virtual data path acceleration
VF	Virtual function
VFE	Virtio full emulation
vHBA	Virtual SCSI host bus adapter
VM	Virtual machine
VMA	NVIDIA® Messaging Accelerator
VNI	Virtual network identifier (or VXLAN network identifier)

Term/ Acronym	Description/Definition
VPI	Virtual protocol interconnect
VRF	Virtual routing and forwarding
VTEP	VXLAN tunnel endpoint
WAN	Wide area network
WorkQ / workq	Work queue
WQE	Work queue elements
WR	Write
XLIO	NVIDIA® Accelerated IO

11.2 DOCA Crypto Acceleration

NVIDIA® BlueField® DPU incorporates several Public Key Acceleration (PKA) engines to offload the processor of the Arm host, providing high-performance computation of PK algorithms. BlueField's PKA is useful for a wide range of security applications. It can assist with SSL acceleration, or a secure high-performance PK signature generator/checker and certificate related operations.

BlueField's PKA software libraries implement a simple, complete framework for crypto public key infrastructure (PKI) acceleration. It provides direct access to hardware resources from the user space, and makes available a number of arithmetic operations—some basic (e.g., addition and multiplication), and some complex (e.g., modular exponentiation and modular inversion)—and high-level operations such as RSA, Diffie-Hallman, Elliptic Curve Cryptography, and the Federal Digital Signature Algorithm (DSA as documented in FIPS-186) public-private key systems.

Some of the use cases for the BlueField PKA involve integrating OpenSSL software applications with BlueField's PKA hardware. The BlueField PKA dynamic engine for OpenSSL allows applications integrated with OpenSSL (e.g., StrongSwan) to accomplish a variety of security-related goals and to accelerate the cryptographic processing with the BlueField PKA hardware. OpenSSL versions ≥1.0.0, ≤1.1.1, and 3.0.2 are supported.

> ⚠ With CentOS 7.6, only OpenSSL 1.1 (not 1.0) works with PKA engine and keygen. Use **`openssl11`** with PKA engine and keygen.

The engine supports the following operations:

- RSA
- DH
- DSA
- ECDSA
- ECDH
- Random number generation that is cryptographically secure.

Up to 4096-bit keys for RSA, DH, and DSA operations are supported. Elliptic Curve Cryptography support of (nist) prime curves for 160, 192, 224, 256, 384 and 521 bits.

For example:

To sign a file using BlueField's PKA engine:

```
$ openssl dgst -engine pka -sha256 -sign <privatekey> -out <signature> <filename>
```

To verify the signature, execute:

```
$ openssl dgst -engine pka -sha256 -verify <publickey> -signature <signature> <filename>
```

For further details on BlueField PKA, please refer to "PKA Driver Design and Implementation Architecture Document" and/or "PKA Programming Guide". Directions and instructions on how to integrate the BlueField PKA software libraries are provided in the README files on our PKA GitHub.

11.3 DOCA Services Fluent Logger

This guide provides instructions on how to use the logging infrastructure for DOCA services on top of NVIDIA® BlueField® DPU.

11.3.1 Introduction

Fluent Bit is a fast log collector that collects information from multiple sources and then forwards the data onward using Fluent.

On NVIDIA DPUs, the Fluent Bit logger can be easily configured to collect system data and the logs from the different DOCA services.

11.3.2 Deployment

The deployment is based on a recommended configuration template for the existing Fluent Bit container.

For information about the deployment of DOCA containers on top of the BlueField DPU, refer to NVIDIA DOCA Container Deployment Guide.

The following is an example YAML file for deploying the Fluent Bit pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: fluent-bit
spec:
  hostNetwork: true
  containers:
  - name: fluent-bit
    image: fluent/fluent-bit:latest
    imagePullPolicy: Always
    # Example resource definitions
    resources:
      requests:
        memory: "100Mi"
        cpu: "200m"
      limits:
        memory: "200Mi"
        cpu: "300m"
    volumeMounts:
    - name: varlog
      mountPath: /var/log
    - name: config-file
      mountPath: /fluent-bit/etc/fluent-bit.conf
  volumes:
  - name: varlog
    hostPath:
      path: /var/log
  - name: config-file
    hostPath:
```

```
      path: /opt/mellanox/doca/services/fluent-bit.conf
      type: File
```

As explained in the "Configuration" section, Fluent Bit uses a configuration file. As such, to ensure that the example YAML file is shared from the DPU to the deployed Fluent Bit container, use the following:

```
path: /opt/mellanox/doca/services/fluent-bit.conf
```

The path below is just an example for where the user can place the **fluent-bit.conf** file. The file could be placed in a different directory on the DPU as long as the YAML file points to the updated location.

11.3.3 Configuration

The Fluent Bit configuration file should have the following sections:

- **[SERVICE]** – to define the service specifications
- **[INPU]** – to define folders to collect logs from (there could be multiple inputs)
- **[OUTPUT]** – IP and port to stream the data to

Example configuration file:

```
[SERVICE]
    Flush         2
    Log_Level     info
    Daemon        off
    Parsers_File  parsers.conf
    HTTP_Server   On
    HTTP_Listen   0.0.0.0
    HTTP_Port     2020

[INPUT]
    Name              tail
    Tag               kube.*
    Path              /var/log/containers/*.log
    Parser            docker
    Mem_Buf_Limit     5MB
    Skip_Long_Lines   On
    Refresh_Interval  10

[INPUT]
    Name              tail
    Tag               sys.*
    Path              /var/log/doca/*/*.log
    Mem_Buf_Limit     5MB
    Skip_Long_Lines   On
    Refresh_Interval  10

[OUTPUT]
    Name  es
    Match *
    Host 10.20.30.40
    Port 9201
    Index fluent_bit
    Type cpu_metrics
```

The most important field to pay attention to is **Path** for the **INPUT** section. DOCA services report their logs to a unique directory under **/var/log/doca/<service_name>/*.log** per the respective DOCA service. As such, the configuration above defines the **/var/log/doca/*/*.log** input definition.

More information about the full specifications can be found in the official Fluent Bit manual.

11.3.4 Troubleshooting

For container-related troubleshooting, refer to the "Troubleshooting" section in the NVIDIA DOCA Container Deployment Guide.

For general troubleshooting, refer to the DOCA Troubleshooting.

When copying the above YAML file, it is possible that the container infrastructure logs give an error related to RFC 1123". These errors are usually a result of a spacing error in the file, which sometimes occur when copying the file as is from this page. To fix this issue, make sure that only the space character (' ') is used as a spacer in the file and not other whitespace characters that might have been added during the copy operation.

11.4 DOCA DPU CLI

This guide provides quick access to a useful set of CLI commands and utilities on the NVIDIA® BlueField® DPU environment.

11.4.1 Introduction

This guide provides a concise guide on useful commands for DOCA deployment and configuration.

The tables in this guide provide two categories of commands:

- General commands for Linux/networking environment
- DOCA/DPU-specific commands

> ⚠ For more information about these commands, such as usage instructions, flag options, arguments and so on, use the `-h` option after the command or use the manual (e.g., `man lspci`).

11.4.2 General Commands

Command	Description
`ifconfig`	Used to configure kernel-resident network interfaces. It is used at boot time to set up interfaces as necessary. After that, it is usually only needed when debugging or when system tuning is needed. If no arguments are given, `ifconfig` displays the status of the currently active interfaces. If a single interface argument is given, it displays the status of the given interface only. If a single `-a` argument is given, it displays the status of all interfaces, even those that are down. Otherwise, it configures an interface.
`ethtool <devname>`	Used to query and control network device driver and hardware settings, particularly for wired Ethernet devices. `<devname>` is the name of the network device on which `ethtool` should operate. ⚠ This command shows the speed of the network card of the DPU.

Command	Description
`lspci`	Displays information about PCIe buses in the system and devices connected to them. By default, it shows a brief list of devices.
`tcpdump`	Dump traffic on a network. Usage: `tcpdump -i <interface>` where `<interface>` is any port interface (physical/SF rep/VF port rep).
`ovs-vsctl`	Utility for querying and configuring `ovs-vswitchd`. The `ovs-vsctl` program supports the model of a bridge implemented by Open vSwitch in which a single bridge supports ports on multiple VLANs.
`mount 10.0.0.10:/vol/myshare/myshare/`	Used for mounting a work directory on the DPU. ⚠ Must be used after creating a new directory named `myshare` under root (i.e., `mkdir /myshare`)
`scp`	Secure copy (remote file copy program). Useful for copying files from BlueField to the host and vice versa.
`iperf`	Used for server-client connection. Useful to check if the network connection achieves the speed of the network card on the DPU (line rate).

11.4.3 DPU/DOCA Commands

Command	Description
`ibdev2netdev`	Displays available `mlnx` interfaces
`mst`	Used to start MST service, to stop it, and for other operations with NVIDIA devices like reset and enabling remote access
`cat /etc/mlnx-release`	Displays the full BlueField image (bfb) version
`cat /etc/os-release`	Displays the details of the underlying OS installed on BlueField
`ibv_devinfo`	Displays the current InfiniBand connected devices and relevant information. Useful for checking current firmware version.
`ipmitool power cycle`	Power cycle ⚠ Prior to performing a power cycle, make sure to perform a graceful shutdown.

`echo '1024' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages`
`sudo mkdir /mnt/huge`
`sudo mount -t hugetlbfs -o pagesize=2M nodev /mnt/huge`
	DPDK setup. Allocates hugepages for DPDK environment abstraction layer (EAL).

Command	Description
`mlxdevm tool`	`mlxdevm tool` is found under `/opt/mellanox/iproute2/sbin/`. With this tool it is possible to create an SF and set its state to active, configure a HW address and set it to trusted, deploy the created SF and print info about it.
`/opt/mellanox/iproute2/sbin/ mlxdevm port add pci/ <pci_address> flavour pcisf pfnum <correspondig_physical_function_number> sfnum <unique_sf_number>`	Creates an SF in the flavor of the given PF with the given unique SF number. Example: `/opt/mellanox/iproute2/sbin/mlxdevm port add pci/0000:03`:00.0 flavour pcisf pfnum 0 sfnum 4`
`/opt/mellanox/iproute2/sbin/ mlxdevm port show`	Displays information about the available SFs
`/opt/mellanox/iproute2/sbin/ mlxdevm port function set pci/ 0000:03:00.0/<sf_index> hw_addr <HW_address> trust on state active`	Configures SF capabilities such as setting the HW address, making it "trusted", and setting its state to active. `<sf_index>` the SF. To obtain this index, you may run `mlxdevm port show`. Example: `/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/ 229377 hw_addr 02:25:f2:8d:a2:4c trust on state active`
`$ echo mlx5_core.sf.<next_serial> > /sys/bus/auxiliary/drivers/ mlx5_core.sf_cfg/unbind` `$ echo mlx5_core.sf. <next_serial> > /sys/bus/auxiliary/drivers/ mlx5_core.sf/bind`	These two commands deploy the created SF. The first command unbinds the SF from the default driver, while the second command binds the SF to the actual driver. The deployment phase should be done after the capabilities of the SF are configured. The SF is identified by `<next_serial>` which can be obtained by running the command below.
`ls /sys/bus/auxiliary/devices/ mlx5_core.sf.*`	Displays additional information about the created SFs and their "next serial numbers". For example, if `mlx5_core.sf.2` exists in the output of the command, then running `cat /sys/bus/auxiliary/devices/mlx5_core.sf.2/sfnum` would output the sfnum related to `mlx5_core.sf.2`.
`/opt/mellanox/iproute2/sbin/ mlxdevm port function set pci/ <pci_address>/<sf_index> state inactive` `/opt/mellanox/iproute2/sbin/ mlxdevm port del pci/ <pci_address>/<sf_index>`	These two commands must be executed to delete a given SF. First, users must set the state of the SF to inactive, and only then should it be deleted.
`/opt/mellanox/iproute2/sbin/ mlxdevm port help`	Displays additional information about operations that can be used on created SF ports
`crictl pods`	Displays currently active K8S pods, and their IDs (it might take up to 20-30 seconds for the pod to start)
`crictl ps`	Displays currently active containers and their IDs

Command	Description
`crictl ps -a`	Displays all containers, including containers that recently finished their execution
`crictl logs <container-id>`	Examines the logs of a given container
`crictl exec -it <container-id> /bin/bash`	Attaches a shell to a running container
`journalctl -u kubelet`	Examines the Kubelet logs. Useful when a pod/container fails to spawn.
`crictl stopp <pod-id>`	Stops a running K8S pod
`crictl stop <container-id>`	Stops a running container
`crictl rmi <image-id>`	Removes a container image from the local K8S registry

11.5 DOCA Switching

These pages provides a comprehensive guide to the hardware-accelerated switching capabilities of NVIDIA® BlueField® and NVIDIA® ConnectX® platforms, including configurations for switchdev mode, representor models, and network features such as vDPA, bridge offload, and link aggregation.

11.5.1 Introduction

BlueField and ConnectX networking platforms leverage hardware-based switching offloads to deliver high-performance network virtualization with minimal CPU overhead. By offloading complex switching logic to the E-Switch, these platforms ensure unparalleled scalability and efficiency for modern data center workloads.

This section details the switching capabilities enabled by DOCA, providing the foundational knowledge and configuration steps required to manage virtualized network resources. The following chapters guide users through the DPU kernel representor model, hardware-accelerated vDPA, and bridge offloading techniques. Additionally, it covers essential network management tasks, including the implementation of link aggregation (LAG) and the configuration of host PF/VF parameters and uplink MTU settings.

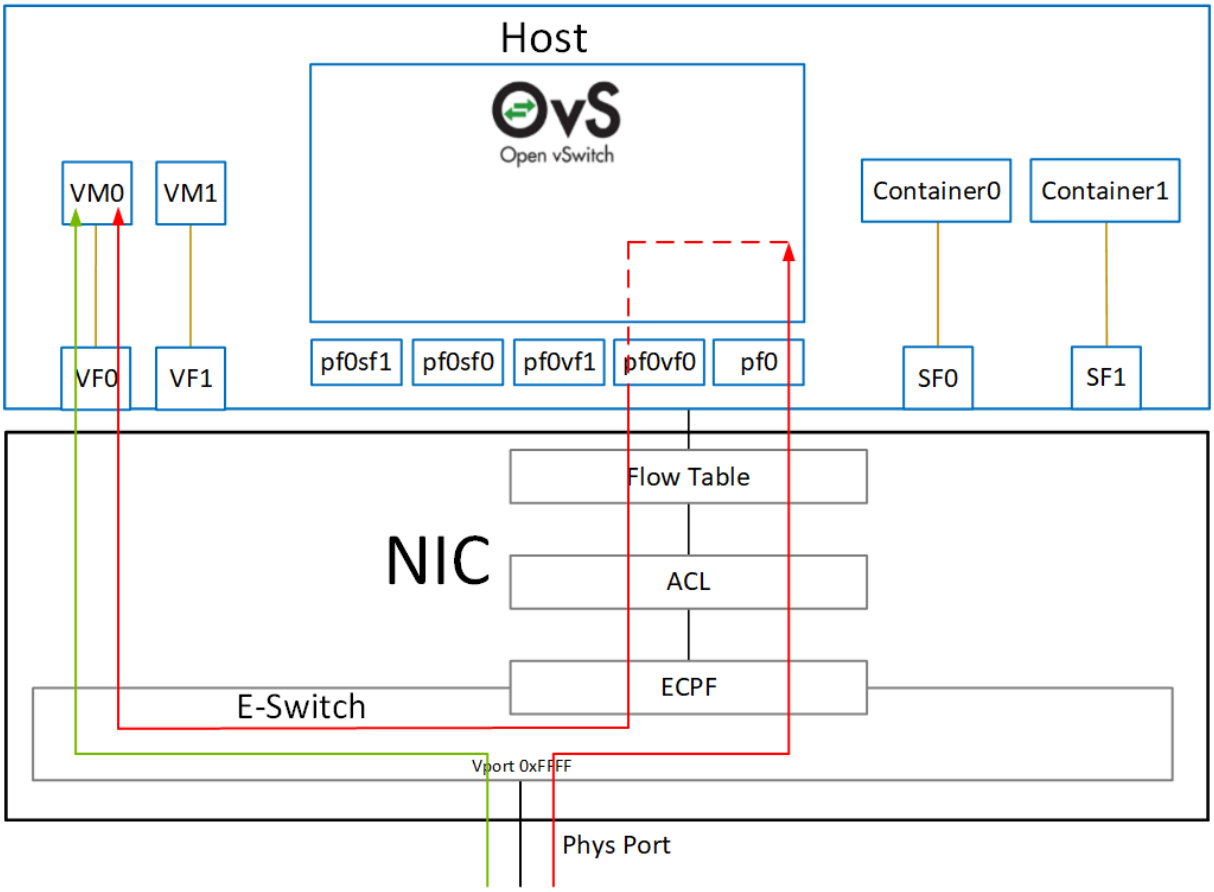
11.5.2 Switch Device Mode

Switchdev mode allows the physical function (PF) to operate as a virtual switch, granting software a means to control associated virtual function (VF) and scalable function (SF) traffic. In this mode, software can intercept incoming/outgoing VF/SF traffic and configure steering rules that offload traffic.

Once this mode is configured, the PF is considered an E-Switch, and additional ports are allocated for each associated VF and SF. The additional ports are referred to as port representors.

11.5.2.1 ConnectX and BlueField NIC Mode

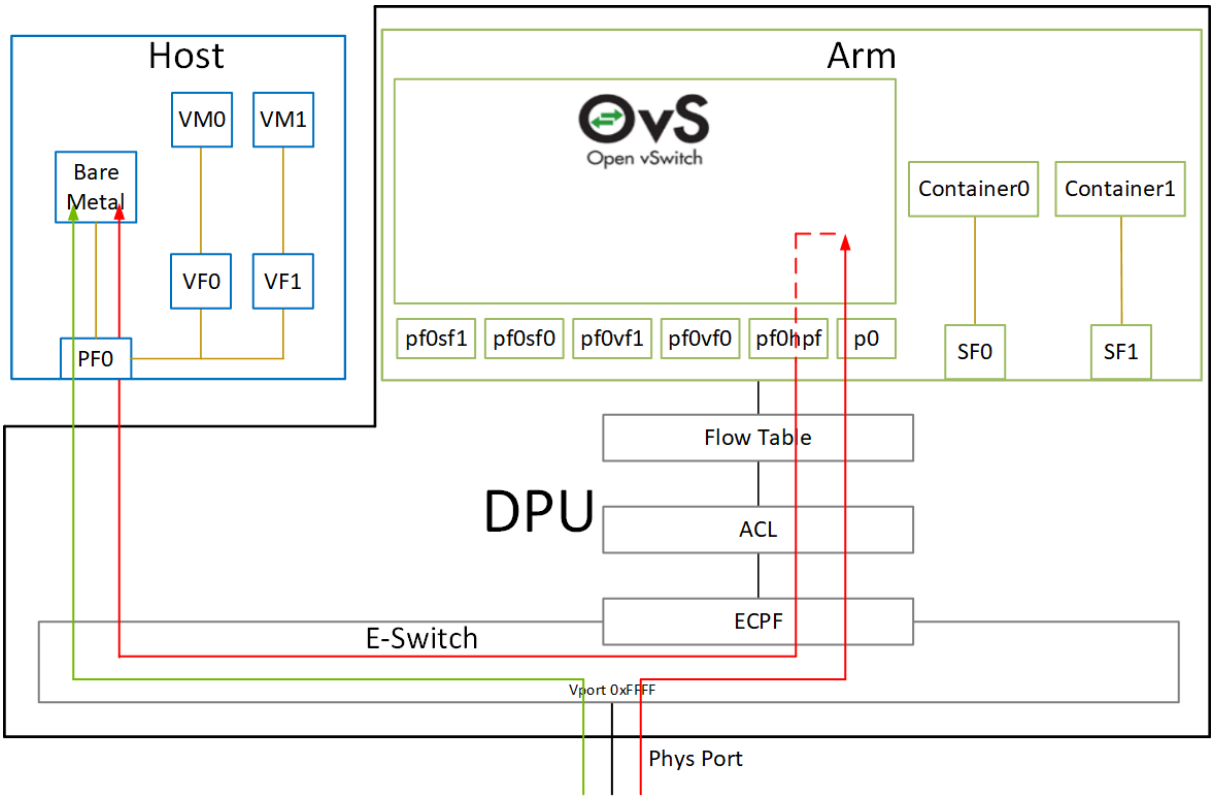
Switchdev mode may be configured for each PF as described in the "Configuring Switchdev Mode" section.



11.5.2.2 BlueField DPU Mode

When BlueField operates in DPU mode, switchdev mode is configured by default for all PFs on the Arm. That configuration must not be changed. In this mode, the host cannot configure PFs to operate in switchdev mode. Instead, host PFs operate in legacy mode only.

The following diagram shows the mapping of between the PCIe functions exposed on the host side and the representors. For the sake of simplicity, the diagram shows a single port model (duplicated for the second port).



The red arrow demonstrates packet flow through the representors, while the green arrow demonstrates the packet flow when steering rules are offloaded to the embedded switch.

11.5.3 E-Switch Port

Once switchdev mode is configured, the PF acts as an e-switch and manages all switch ports. The e-switch can be used to control VF and SF traffic. Software can then use the e-switch through the netdev and RDMA core interfaces.

11.5.3.1 Netdev Interface

The e-switch has a netdev interface (e.g., **pf0**), allowing it to be used with common Linux networking tools (e.g., **ifconfig**) and OVS. While the interface looks like a regular network port, in reality it is considered the upstream representor and it cannot host an IP server. That is, configuring an IP address to this interface is not useful (e.g., it cannot be pinged).

The netdev interface can be used mainly by OVS and the Linux bridge.

To overcome this limitation, it is possible to create SFs. For more details, see BlueField Scalable Functions.

11.5.3.2 RDMA Core Interface

The e-switch has an RDMA device instance (e.g., **mlx5_0**) which grants software access to the e-switch capabilities for offloading steering rules and accessing various RDMA capabilities.

The instance behaves like an RDMA core device with the following limitations:

- It cannot host an RDMA connection and cannot be used to connect to the remote RDMA target (e.g., cannot do RDMA send)
- It cannot have an RDMA GID

The RDMA device can mainly be used to configure the e-switch.

To overcome this limitation, it is possible to create SFs. For more details, see BlueField Scalable Functions.

11.5.4 Represntor Port

For each network function (i.e., VF/SF) a corresponding representor port is created. The representor port represents the actual port. The network function can then be used to access the network by a VM/container, while the representor can be used to manage switching by virtual switch software (e.g., bare metal host).

11.5.4.1 Netdev Interface

Each representor port would have a netdev interface (e.g., **pf0vf0**), allowing it to be used with common Linux networking tools (e.g., **ifconfig**) as well as OVS.

While the interface looks like a regular network port, in reality it is a VF/SF representor and it cannot host an IP server. That is, configuring an IP address to this interface is not useful (e.g., it cannot be pinged).

The representor netdev can be used to refer to a specific network function when configuring steering rules using OVS or Linux bridge.

11.5.4.2 RDMA Core Interface

The representor does not have a corresponding RDMA core instance (e.g., **mlx5_0**). Instead, it is managed by the RDMA core instance of the e-switch manager. Each representor is an RDMA port of the e-switch RDMA device.

11.5.5 Configuring Switchdev Mode

> ⚠ For BlueField in DPU mode, there is no need to follow these steps as the PFs are already configured to switchdev mode by default.

1. Unbind all VFs:

```
# echo 0000:3d:00.2 > /sys/bus/pci/drivers/mlx5_core/unbind
```

```
# echo 0000:3d:00.3 > /sys/bus/pci/drivers/mlx5_core/unbind
```

VMs with attached VFs must be powered off to be able to unbind the VFs.

2. Change the e-switch mode from legacy to switchdev on the PF device:

```
# devlink dev eswitch set pci/0000:3d:00.0 mode switchdev
```

This creates the VF/SF representor ports in the host OS.

Before changing the mode, make sure that all VFs are unbound.

To return to legacy mode, run:

```
# devlink dev eswitch set pci/0000:3d:00.0 mode legacy
```

This removes the VF/SF representor ports.

On OSes or kernels that do not support devlink, moving to switchdev mode can be done using sysfs:

```
# echo switchdev > /sys/class/net/pf0/compat/devlink/mode
```

3. At this stage, VF representors have been created. To map a representor to its VF, make sure to obtain the representor's **switchid** and **portname** by running:

```
# ip -d link show eth0
41: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group default qlen 1000
    link/ether ba:e6:21:37:bc:d4 brd ff:ff:ff:ff:ff:ff promiscuity 0 addrgenmode eui64 numtxqueues 10
 numrxqueues 10 gso_max_size 65536 gso_max_segs 65535 portname pf0vf0 switchid f4ab580003a1420c
```

Where:

- **switchid** – used to map representor to device, both device PFs have the same **switchid**
- **portname** – used to map representor to PF and VF. Value returned is **pf<X>vf<Y>**, where **X** is the PF number and **Y** is the number of VF.

4. Bind the VFs:

```
echo 0000:3d:00.2 > /sys/bus/pci/drivers/mlx5_core/bind
echo 0000:3d:00.3 > /sys/bus/pci/drivers/mlx5_core/bind
```

11.5.6 DPU Kernel Representors Model

This model is only applicable when the NVIDIA® BlueField® networking platform (DPUs or SuperNIC) is operating in DPU mode.

BlueField uses netdev representors to map each one of the host side physical and virtual functions.

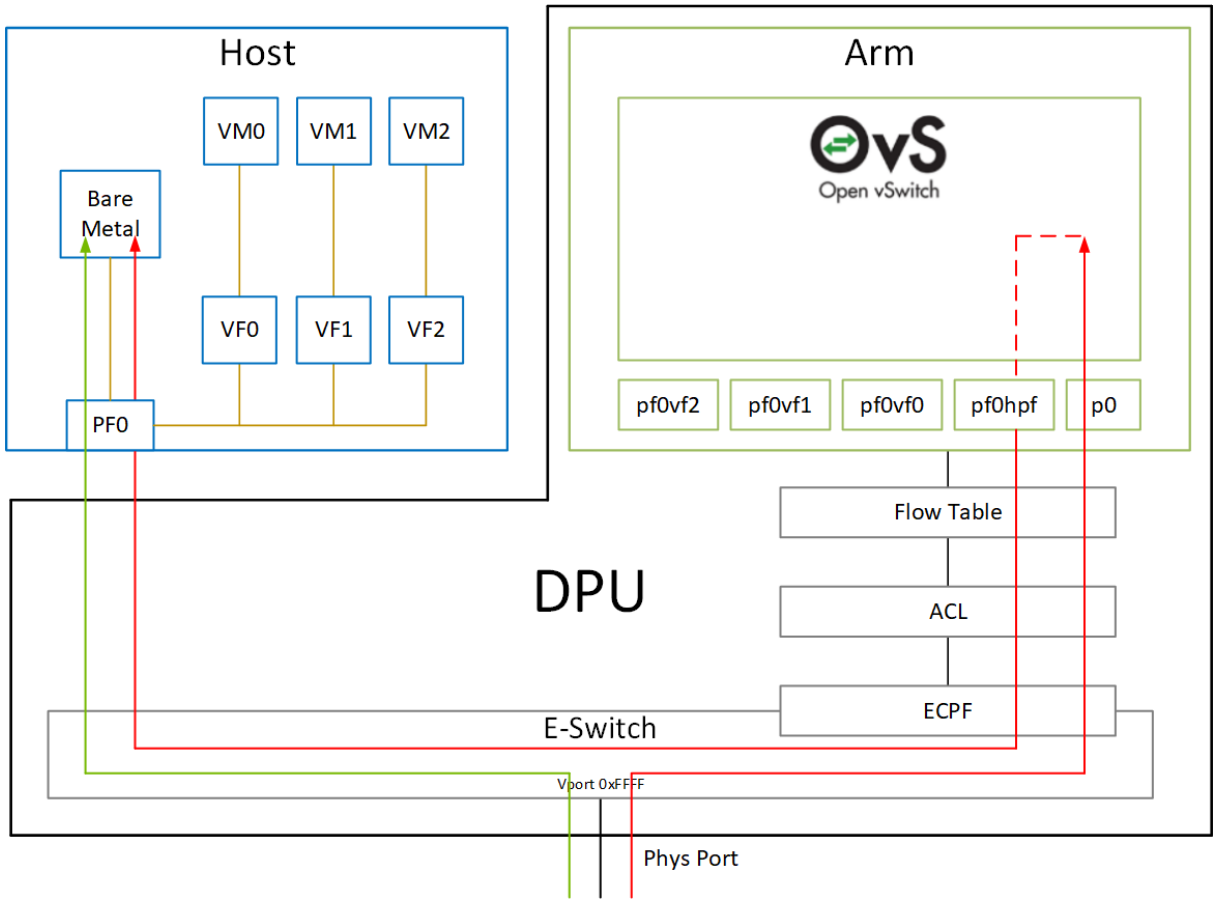
1. Serve as the tunnel to pass traffic for the virtual switch or application running on the Arm cores to the relevant PF or VF on the host side.
2. Serve as the channel to configure the embedded switch with rules to the corresponding represented function.

Those representors are used as the virtual ports being connected to OVS or any other virtual switch running on the Arm cores.

When operating in DPU mode, we see 2 representors for each one of the BlueField's network ports: one for the uplink, and another one for the host side PF (the PF representor created even if the PF is not probed on the host side). For each one of the VFs created on the host side a corresponding representor would be created on the Arm side. The naming convention for the representors is as follows:

- Uplink representors: **p<port_number>**
- PF representors: **pf<port_number>hpf**
- VF representors: **pf<port_number>vf<function_number>**

The following diagram shows the mapping of between the PCIe functions exposed on the host side and the representors. For the sake of simplicity, a single port model (duplicated for the second port) is shown.



The red arrow demonstrates a packet flow through the representors, while the green arrow demonstrates the packet flow when steering rules are offloaded to the embedded switch. More details on that are available in the switch offload section.

The MTU of host functions (PF/VF) must be smaller than the MTUs of both the uplink and corresponding PF/VF representor. For example, if the host PF MTU is set to 9000, both uplink and PF representor must be set to above 9000.

11.5.7 Virtio Acceleration through Hardware vDPA

11.5.7.1 Hardware vDPA Installation

Hardware vDPA requires QEMU v2.12 (or with upstream 6.1.0) and DPDK v20.11 as minimal versions.

To install QEMU:

1. Clone the sources:

```
git clone https://git.qemu.org/git/qemu.git
cd qemu
git checkout v2.12
```

2. Build QEMU:

```
mkdir bin
cd bin
../configure --target-list=x86_64-softmmu --enable-kvm
make -j24
```

To install DPDK:

1. Clone the sources:

```
git clone git://dpdk.org/dpdk
cd dpdk
git checkout v20.11
```

2. Install dependencies (if needed):

```
yum install cmake gcc libnl3-devel libudev-devel make pkgconfig valgrind-devel pandoc libibverbs libmlx5
libmnl-devel -y
```

3. Configure DPDK:

```
export RTE_SDK=$PWD
make config T=x86_64-native-linuxapp-gcc
cd build
sed -i 's/\(CONFIG_RTE_LIBRTE_MLX5_PMD=\)n/\1y/g' .config
sed -i 's/\(CONFIG_RTE_LIBRTE_MLX5_VDPA_PMD=\)n/\1y/g' .config
```

4. Build DPDK:

```
make -j
```

5. Build the vDPA application:

```
cd $RTE_SDK/examples/vdpa/
```

```
make -j
```

11.5.7.2 Hardware vDPA Configuration

To configure huge pages:

```
mkdir -p /hugepages
mount -t hugetlbfs hugetlbfs /hugepages
echo <more> > /sys/devices/system/node/node0/hugepages/hugepages-1048576kB/nr_hugepages
echo <more> > /sys/devices/system/node/node1/hugepages/hugepages-1048576kB/nr_hugepages
```

To configure a vDPA VirtIO interface in an existing VM's xml file (using **libvirt**):

1. Open the VM's configuration XML for editing:

   ```
   virsh edit <domain name>
   ```

2. Perform the following:

   a. Change the top line to:

   ```
   <domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
   ```

   b. Assign a memory amount and use 1GB page size for huge pages (size must be the same as that used for the vDPA application), so that the memory configuration looks as follows.

   ```
   <memory unit='KiB'>4194304</memory>
   <currentMemory unit='KiB'>4194304</currentMemory>
   <memoryBacking>
     <hugepages>
       <page size='1048576' unit='KiB'/>
     </hugepages>
   </memoryBacking>
   ```

   c. Assign an amount of CPUs for the VM CPU configuration, so that the **vcpu** and **cputune** configuration looks as follows:

   ```
   <vcpu placement='static'>5</vcpu>
   <cputune>
     <vcpupin vcpu='0' cpuset='14'/>
     <vcpupin vcpu='1' cpuset='16'/>
     <vcpupin vcpu='2' cpuset='18'/>
     <vcpupin vcpu='3' cpuset='20'/>
     <vcpupin vcpu='4' cpuset='22'/>
   </cputune>
   ```

   d. Set the memory access for the CPUs to be shared, so that the **cpu** configuration looks as follows:

   ```
   <cpu mode='custom' match='exact' check='partial'>
     <model fallback='allow'>Skylake-Server-IBRS</model>
     <numa>
       <cell id='0' cpus='0-4' memory='8388608' unit='KiB' memAccess='shared'/>
     </numa>
   </cpu>
   ```

   e. Set the emulator in use to be the one built in step 2, so that the emulator configuration looks as follows:

   ```
   <emulator><path to qemu executable></emulator>
   ```

f. Add a virtio interface using QEMU command line argument entries, so that the new interface snippet looks as follows:

```
<qemu:commandline>
  <qemu:arg value='-chardev'/>
  <qemu:arg value='socket,id=charnet1,path=/tmp/sock-virtio0'/>
  <qemu:arg value='-netdev'/>
  <qemu:arg value='vhost-user,chardev=charnet1,queues=16,id=hostnet1'/>
  <qemu:arg value='-device'/>
  <qemu:arg value='virtio-net-
pci,mq=on,vectors=6,netdev=hostnet1,id=net1,mac=e4:11:c6:d3:45:f2,bus=pci.0,addr=0x6,
   page-per-vq=on,rx_queue_size=1024,tx_queue_size=1024'/>
</qemu:commandline>
```

⚠ In this snippet, the vhostuser socket file path, the amount of queues, the MAC and the PCIe slot of the virtio device can be configured.

11.5.7.3 Running Hardware vDPA

⚠ Hardware vDPA supports switchdev mode only.

1. Create the ASAP^2 environment:
   a. Create the VFs.
   b. Enter switchdev mode.
   c. Set up OVS.
2. Run the vDPA application:

```
cd $RTE_SDK/examples/vdpa/build
./vdpa -w <VF PCI BDF>,class=vdpa --log-level=pmd,info -- -i
```

3. Create a vDPA port via the vDPA application CLI:

```
create /tmp/sock-virtio0 <PCI DEVICE BDF>
```

⚠ The vhostuser socket file path must be the one used when configuring the VM.

4. Start the VM:

```
virsh start <domain name>
```

For further information on the vDPA application, visit the Vdpa Sample Application DPDK documentation.

11.5.8 Bridge Offload

⚠ Bridge offload is supported switchdev mode only.

⚠ Bridge offload is supported from kernel version 5.15 onward.

A Linux bridge is an in-kernel software network switch (based on and implementing a subset of IEEE 802.1D standard) used to connect Ethernet segments together in a protocol-independent manner. Packets are forwarded based on L2 Ethernet header addresses.

mlx5 provides the ability to offload bridge dataplane unicast packet forwarding and VLAN management to hardware.

11.5.8.1 Basic Configuration

1. Initialize the ASAP^2 environment:
    a. Create the VFs.
    b. Enter switchdev mode.
2. Create a bridge and add mlx5 representors to bridge:

```
ip link add name bridge0 type bridge
ip link set enp8s0f0_0 master bridge0
```

11.5.8.2 Configuring VLAN

1. Enable VLAN filtering on the bridge:

```
ip link set bridge0 type bridge vlan_filtering 1
```

2. Configure port VLAN matching (trunk mode). In this configuration, only packets with specified VID are allowed.

```
bridge vlan add dev enp8s0f0_0 vid 2
```

3. Configure port VLAN tagging (access mode). In this configuration, VLAN header is pushed/popped upon reception/transmission on port.

```
bridge vlan add dev enp8s0f0_0 vid 2 pvid untagged
```

11.5.8.3 VF LAG Support

Bridge supports offloading on bond net device that is fully initialized with mlx5 uplink representors and is in single (shared) FDB LAG mode. Details about initialization of LAG are provided in section "SR-IOV VF LAG".

To add a bonding net device to bridge:

```
ip link set bond0 master bridge0
```

For further information on interacting with Linux bridge via iproute2 bridge tool, refer to man 8 bridge.

11.5.9 Link Aggregation

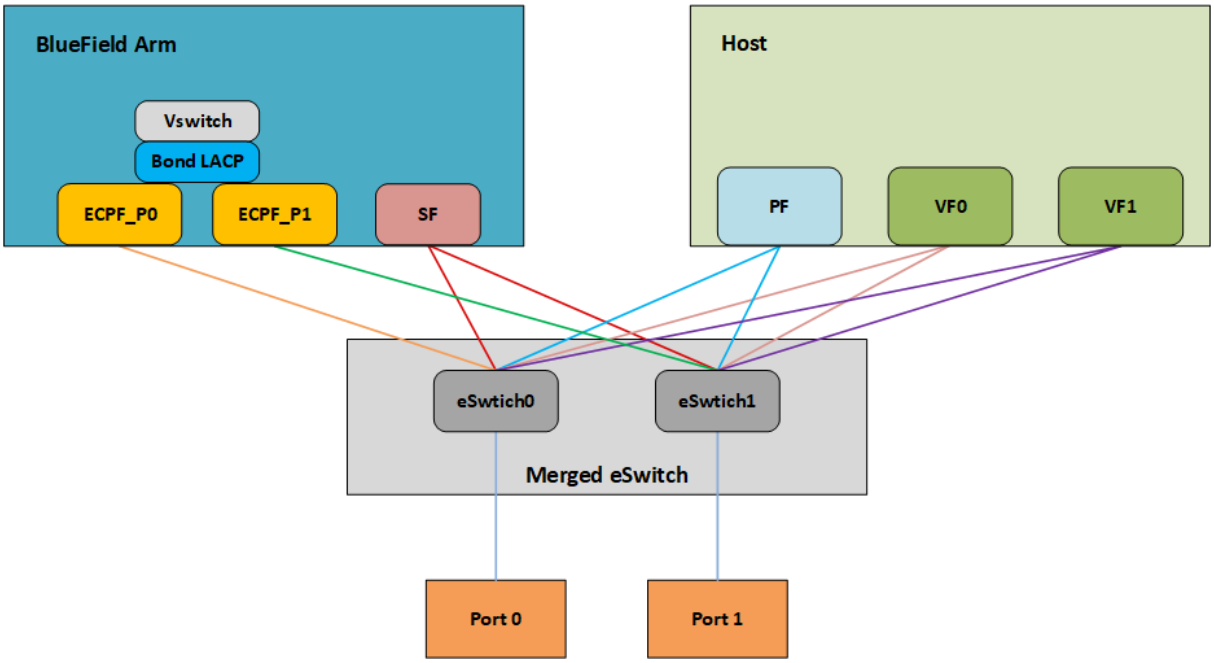
Network bonding enables combining two or more network interfaces into a single interface. It increases the network throughput, bandwidth and provides redundancy if one of the interfaces fails.

NVIDIA® BlueField® networking platforms (DPUs or SuperNICs) have an option to configure network bonding on the Arm side in a manner transparent to the host. Under such configuration, the host would only see a single PF.

> ⚠ This functionality is supported when BlueField is set in embedded function ownership mode for both ports.

> ⚠ While LAG is being configured (starting with step 2 under section "LAG Configuration"), traffic cannot pass through the physical ports.

The following diagram describes this configuration:



11.5.9.1 LAG Modes

Two LAG modes are supported on BlueField:

- Queue Affinity mode
- Hash mode

11.5.9.1.1 Queue Affinity Mode

In this mode, packets are distributed according to the QPs.

1. To enable this mode, run:

```
$ mlxconfig -d /dev/mst/<device-name> s LAG_RESOURCE_ALLOCATION=0
```

   Example device name: **mt41686_pciconf0**.

2. Add/edit the following field from **/etc/mellanox/mlnx-bf.conf** as follows:

```
LAG_HASH_MODE="no"
```

3. Perform BlueField system reboot for the **mlxconfig** settings to take effect.

11.5.9.1.2 Hash Mode

In this mode, packets are distributed to ports according to the hash on packet headers.

⚠ For this mode, prerequisite steps 3 and 4 are not required.

1. To enable this mode, run:

```
$ mlxconfig -d /dev/mst/<device-name> s LAG_RESOURCE_ALLOCATION=1
```

   Example device name: **mt41686_pciconf0**.

2. Add/edit the following field from **/etc/mellanox/mlnx-bf.conf** as follows:

```
LAG_HASH_MODE="yes"
```

3. Perform BlueField system reboot for the **mlxconfig** settings to take effect.

11.5.9.2 Prerequisites

1. Set the LAG mode to work with.
2. (Optional) Hide the second PF on the host. Run:

```
$ mlxconfig -d /dev/mst/<device-name> s HIDE_PORT2_PF=True NUM_OF_PF=1
```

   Example device name: **mt41686_pciconf0**.

   ⚠ Perform BlueField system reboot for the **mlxconfig** settings to take effect.

3. Delete any installed Scalable Functions (SFs) on the Arm side.
4. Stop the driver on the host side. Run:

```
$ systemctl stop openibd
```

5. The uplink interfaces (**p0** and **p1**) on the Arm side must be disconnected from any OVS bridge.

11.5.9.3 LAG Configuration

1. Create the bond interface. Run:

```
$ ip link add bond0 type bond
$ ip link set bond0 down
$ ip link set bond0 type bond miimon 100 mode 4 xmit_hash_policy layer3+4
```

> ⚠ While LAG is being configured (starting with the next step), traffic cannot pass through the physical ports.

2. Subordinate both the uplink representors to the bond interface. Run:

```
$ ip link set p0 down
$ ip link set p1 down
$ ip link set p0 master bond0
$ ip link set p1 master bond0
```

3. Bring the interfaces up. Run:

```
$ ip link set p0 up
$ ip link set p1 up
$ ip link set bond0 up
```

The following is an example of LAG configuration in Ubuntu:

```
# cat  /etc/netplan/70-bf-lag.yaml

network:
  renderer: networkd
  bonds:
    bond0:
      optional: true
      dhcp4: no
      interfaces: [p0, p1]
      parameters:
        mode: 802.3ad
        mii-monitor-interval: 100
  ethernets:
    p0:
      dhcp4: no
      optional: true
    p1:
      dhcp4: no
      optional: true
  version: 2
```

As a result, only the first PF of the BlueFields would be available to the host side for networking and SR-IOV.

> ⛔ When in shared RQ mode (enabled by default), the uplink interfaces (**p0** and **p1**) must always stay enabled. Disabling them will break LAG support and VF-to-VF communication on same host.

For OVS configuration, the bond interface is the one that needs to be added to the OVS bridge (interfaces **p0** and **p1** should not be added). The PF representor for the first port (**pf0hpf**) of the LAG must be added to the OVS bridge. The PF representor for the second port (**pf1hpf**) would still be visible, but it should not be added to OVS bridge. Consider the following examples:

```
ovs-vsctl add-br bf-lag
ovs-vsctl add-port bf-lag bond0
ovs-vsctl add-port bf-lag pf0hpf
```

Trying to change bonding configuration in Queue Affinity mode (including bringing the subordinated interface up/down) while the host driver is loaded would cause FW syndrome and failure of the operation. Make sure to unload the host driver before altering BlueField bonding configuration to avoid this.

When performing driver reload (**openibd restart**) or reboot, it is required to remove bond configuration and to reapply the configurations after the driver is fully up. Refer to steps 1-4 of "Removing LAG Configuration".

11.5.9.4 Removing LAG Configuration

1. If Queue Affinity mode LAG is configured (i.e., **LAG_RESOURCE_ALLOCATION=0**):
   a. Delete any installed Scalable Functions (SFs) on the Arm side.
   b. Stop driver (openibd) on the host side. Run:

   ```
   systemctl stop openibd
   ```
2. Delete the LAG OVS bridge on the Arm side. Run:

   ```
   ovs-vsctl del-br bf-lag
   ```

   This allows for later restoration of OVS configuration for non-LAG networking.
3. Stop OVS service. Run:

   ```
   systemctl stop openvswitch-switch.service
   ```
4. Run:

   ```
   ip link set bond0 down
   modprobe -rv bonding
   ```

   As a result, both of the BlueField's network interfaces would be available to the host side for networking and SR-IOV.
5. For the host to be able to use BlueField's ports, make sure to attach the ECPF and host representor in an OVS bridge on the Arm side. Refer to "Virtual Switch on BlueField" for instructions on how to perform this.
6. Revert from **HIDE_PORT2_PF** , on the Arm side. Run:

   ```
   mlxconfig -d /dev/mst/<device-name> s HIDE_PORT2_PF=False NUM_OF_PF=2
   ```
7. Restore default LAG settings in BlueField's firmware. Run:

   ```
   mlxconfig -d /dev/mst/<device-name> s LAG_RESOURCE_ALLOCATION=DEVICE_DEFAULT
   ```
8. Delete the following line from **/etc/mellanox/mlnx-bf.conf** on the Arm side:

```
LAG_HASH_MODE=...
```

9. Perform BlueField system reboot for the **mlxconfig** settings to take effect.

11.5.9.5 LAG on Multi-host

Only LAG hash mode is supported with BlueField multi-host.

11.5.9.5.1 LAG Multi-host Prerequisites

1. Enable LAG hash mode.
2. Hide the second PF on the host. Run:

```
$ mlxconfig -d /dev/mst/<device-name> s HIDE_PORT2_PF=True NUM_OF_PF=1
```

3. Make sure NVME emulation is disabled:

```
$ mlxconfig -d /dev/mst/<device-name> s NVME_EMULATION_ENABLE=0
```

   Example device name: **mt41686_pciconf0**.

4. The uplink interfaces (**p0** and **p4**) on the Arm side, representing port0 and port1, must be disconnected from any OVS bridge. As a result, only the first PF of BlueField would be available to the host side for networking and SR-IOV.

11.5.9.5.2 LAG Configuration on Multi-host

1. Create the bond interface. Run:

```
$ ip link add bond0 type bond
$ ip link set bond0 down
$ ip link set bond0 type bond miimon 100 mode 4 xmit_hash_policy layer3+4
```

2. Subordinate both the uplink representors to the bond interface. Run:

```
$ ip link set p0 down
$ ip link set p4 down
$ ip link set p0 master bond0
$ ip link set p4 master bond0
```

3. Bring the interfaces up. Run:

```
$ ip link set p0 up
$ ip link set p4 up
$ ip link set bond0 up
```

4. For OVS configuration, the bond interface is the one that must be added to the OVS bridge (interfaces p0 and p4 should not be added). The PF representor, **pf0hpf**, must be added to the OVS bridge with the bond interface. The rest of the uplink representors must be added to another OVS bridge along with their PF representors. Consider the following examples:

```
ovs-vsctl add-br br-lag
ovs-vsctl add-port br-lag bond0
ovs-vsctl add-port br-lag pf0hpf
ovs-vsctl add-br br1
ovs-vsctl add-port br1 p1
ovs-vsctl add-port br1 pf1hpf
ovs-vsctl add-br br2
ovs-vsctl add-port br2 p2
ovs-vsctl add-port br2 pf2hpf
```

```
ovs-vsctl add-br br3
ovs-vsctl add-port br3 p3
ovs-vsctl add-port br3 pf3hpf
```

When performing driver reload (**openibd restart**) or reboot, you must remove bond configuration from NetworkManager, and to reapply the configurations after the driver is fully up.

11.5.9.5.3 Removing LAG Configuration on Multi-host

Refer to section "Removing LAG Configuration".

11.5.10 Controlling Host PF and VF Parameters

NVIDIA® BlueField® networking platforms (DPUs or SuperNICs) allow control over some of the networking parameters of the PFs and VFs running on the host side.

11.5.10.1 Setting Host PF and VF Default MAC Address

From the Arm, users may configure the MAC address of the physical function in the host. After sending the command, users must reload the NVIDIA driver in the host to see the newly configured MAC address. The MAC address goes back to the default value in the firmware after system reboot.

Example:

```
$ echo "c4:8a:07:a5:29:59" > /sys/class/net/p0/smart_nic/pf/mac
$ echo "c4:8a:07:a5:29:61" > /sys/class/net/p0/smart_nic/vf0/mac
```

11.5.10.2 Setting Host PF and VF Link State

vPort state can be configured to Up, Down, or Follow. For example:

```
$ echo "Follow" > /sys/class/net/p0/smart_nic/pf/vport_state
```

11.5.10.3 Querying Configuration

To query the current configuration, run:

```
$ cat /sys/class/net/p0/smart_nic/pf/config
MAC        : e4:8b:01:a5:79:5e
MaxTxRate  : 0
State      : Follow
```

Zero signifies that the rate limit is unlimited.

11.5.10.4 Disabling Host Networking PFs

It is possible to not expose networking functions to the host for users interested in using storage or virtio functions only. When this feature is enabled, the host PF representors (i.e. **pf0hpf** and **pf1hpf**) will not be seen on the Arm.

- Without a PF on the host, it is not possible to enable SR-IOV, so VF representors will not be seen on the Arm either
- Without PFs on the host, there can be no SFs on it

To disable host networking PFs, run:

```
mlxconfig -d /dev/mst/mt41686_pciconf0 s NUM_OF_PF=0
```

To reactivate host networking PFs:

- For single-port BlueFields, run:

  ```
  mlxconfig -d /dev/mst/mt41686_pciconf0 s NUM_OF_PF=1
  ```

- For dual-port BlueFields, run:

  ```
  mlxconfig -d /dev/mst/mt41686_pciconf0 s NUM_OF_PF=2
  ```

⚠ When there are no networking functions exposed on the host, the reactivation command must be run from the Arm.

⚠ Perform BlueField system reboot for the **mlxconfig** settings to take effect.

11.5.10.5 Configuring Uplink MTU

To configure the port MTU while operating in DPU mode, users must restrict the external host port ownership by issuing the following command on the BlueField:

```
mlxprivhost -d /dev/mst/<pciconf0 device> r --disable_port_owner
```

Server cold reboot is required for this restriction to take effect.

Once the host is restricted, the port MTU is configured by changing the MTU of the uplink representor (**p0** or **p1**).

11.5.11 DPDK on BlueField

Please refer to "NVIDIA BlueField Board Support Package" in the DPDK documentation.

11.6 DOCA with OpenSSL

This guide provides instructions on using DOCA SHA for OpenSSL implementations.

11.6.1 Introduction

The **doca_sha_offload_engine** is an OpenSSL dynamic engine with the ability of offloading SHA calculation. It can offload the OpenSSL one-shot SHA-1, SHA-256, and SHA-512. It supports synchronous mode and asynchronous mode by leveraging the OpenSSL **async_jobs** library. For more information on the **async_jobs** library, please refer to official OpenSSL documentation.

This engine is based on the **doca_sha** library and the OpenSSL dynamic engine interface API. For more information on the OpenSSL dynamic engine, please refer to official OpenSSL documentation.

This engine can be called by an OpenSSL application through the OpenSSL high-level algorithm call interface, **EVP_Digest**. For more information on the **EVP_Digest**, please refer to official OpenSSL documentation.

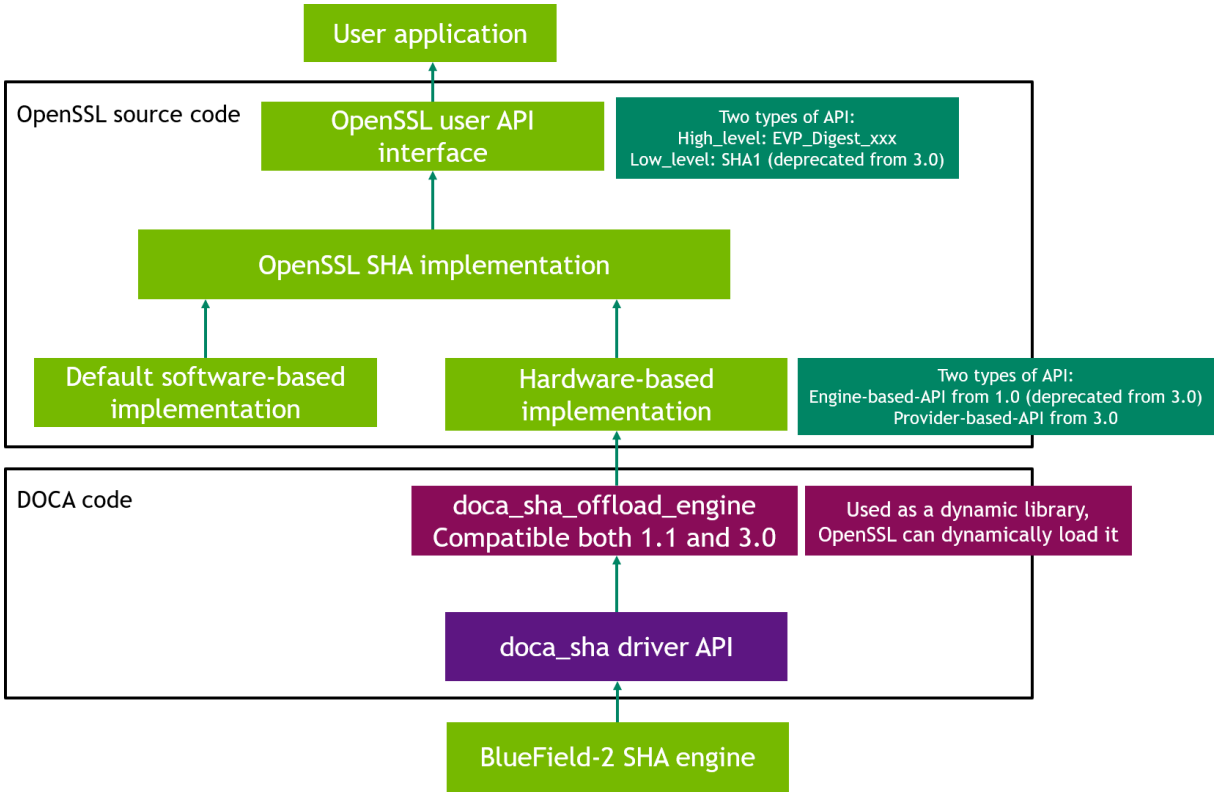
11.6.2 Prerequisites

- Hardware-based **doca_sha** engine which can be verified by calling **doca_sha_get_hardware_supported()**
- Installed OpenSSL version ≥ 1.1.1

11.6.3 Architecture

The following diagram shows the software hierarchy of **doca_sha_offload_engine** and its location in the whole DOCA repository.

From the perspective of OpenSSL, this engine is an instantiation of the OpenSSL dynamic engine interface API by leveraging the **doca_sha** library.



11.6.4 Capabilities and Limitations

- Only one-shot OpenSSL SHA is supported
- The maximum message length ≤ 2GB, the same as **doca_sha** library

11.6.5 OpenSSL Command Line Verification

Verify that the engine can be loaded:

```
$ openssl engine dynamic -pre NO_VCHECK:1 -pre SO_PATH:${DOCA_DIR}/infrastructure/doca_sha_offload_engine/
libdoca_sha_offload_engine.so -pre LOAD -vvv -t -c
(dynamic) Dynamic engine loading support
[Success]: SO_PATH:${DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_sha_offload_engine.so
[Success]: LOAD
Loaded: (doca_sha_offload_engine) Openssl SHA offloading engine based on doca_sha
 [SHA1, SHA256, SHA512]
     [ available ]
     set_pci_addr: set the pci address of the doca_sha_engine
          (input flags): STRING
```

- For SHA-1:

```
$ echo "hello world" | openssl dgst -sha1 -engine {DOCA_DIR}/infrastructure/doca_sha_offload_engine/
libdoca_sha_offload_engine.so -engine_impl
```

- For SHA-256:

```
$ echo "hello world" | openssl dgst -sha256 -engine {DOCA_DIR}/infrastructure/doca_sha_offload_engine/
libdoca_sha_offload_engine.so -engine_impl
```

- For SHA-512:

```
$ echo "hello world" | openssl dgst -sha512 -engine {DOCA_DIR}/infrastructure/doca_sha_offload_engine/
libdoca_sha_offload_engine.so -engine_impl
```

11.6.6 OpenSSL Throughput Test

openssl-speed is the OpenSSL throughput benchmark tool. For more information, consult official OpenSSL documentation. **doca_sha_offload_engine** throughput can also be measured using **openssl-speed**.

- SHA-1, each job 10000 bytes, using engine:

```
$ openssl speed -evp sha1 -bytes 10000 -elapsed --engine {DOCA_DIR}/infrastructure/doca_sha_offload_engine/
libdoca_sha_offload_engine.so
```

- SHA-256, each job 10000 bytes, using engine, **async_jobs=256**:

```
$ openssl speed -evp sha256 -bytes 10000 -elapsed --engine {DOCA_DIR}/infrastructure/
doca_sha_offload_engine/libdoca_sha_offload_engine.so -async_jobs 256
```

- SHA-512, each job 10000 bytes, using engine, **async_jobs=256**, **threads=8**:

```
$ openssl speed -evp sha512 -bytes 10000 -elapsed --engine {DOCA_DIR}/infrastructure/
doca_sha_offload_engine/libdoca_sha_offload_engine.so -async_jobs 256 -multi 8
```

11.6.7 Using DOCA SHA Offload Engine in OpenSSL Application

More information on the dynamic engine usage can be found in the official OpenSSL documentation.

1. To load the **doca_sha_offload_engine** (optionally, set engine PCIe address):

```
ENGINE *e;
const char *doca_engine_path = "${DOCA_DIR}/infrastructure/doca_sha_offload_engine/
libdoca_sha_offload_engine.so";
const char *default_doca_pci_addr = "03:00.0";
ENGINE_load_dynamic();
e = ENGINE_by_id(doca_engine_path);
ENGINE_ctrl_cmd_string(e, "set_pci_addr", doca_engine_pci_addr, 0);
ENGINE_init(e);
ENGINE_set_default_digests(e);
```

2. To perform SHA calculation by calling the OpenSSL high-level function EVP_XXX:

```
const EVP_MD *evp_md = EVP_sha1();
EVP_MD_CTX *mdctx = EVP_MD_CTX_create();
EVP_DigestInit_ex(mdctx, evp_md, e);
EVP_DigestUpdate(mdctx, msg, msg_len);
EVP_DigestFinal_ex(mdctx, digest, digest_len);
EVP_MD_CTX_destroy(mdctx);
```

3. To unload the engine:

```
ENGINE_unregister_digests(e);
ENGINE_finish(e);
ENGINE_free(e);
```

11.7 BlueField Scalable Functions

This page explains how to allocate resources for scalable functions (SFs), create and deploy an SF, use its functional network interface and representor, and remove the SF.

11.7.1 Overview

Scalable functions (SFs), also called subfunctions, are lightweight functions created from a parent PCI Express (PCIe) physical function (PF). Each SF has its own function capabilities, resources, and transmit and receive queues while sharing PCIe-level resources with its parent function and other SFs.

SFs can coexist with Single Root I/O Virtualization (SR-IOV) virtual functions (VFs), but SFs do not require PCIe SR-IOV to be enabled. SFs also support embedded switch (eSwitch) representors, allowing multiple services to use the networking device concurrently on the BlueField DPU.

Characteristic	SF	VF
Parent	PCIe PF	SR-IOV PF
Requires SR-IOV to be enabled	No	Yes
Has dedicated function resources and queues	Yes	Yes
Can have an eSwitch representor	Yes	Yes

The following diagram shows the relationship between SF management, the SF representor, and the functional SF network interface:

```
BlueField Arm system
+-- mlxdevm management interface
+-- SF representor --> eSwitch, OVS, or another forwarding configuration

Selected PCI controller
+-- Functional SF network interface --> workload or application
```

The SF representor is created on the BlueField Arm system. The selected controller determines where the functional SF network interface becomes available after the SF is activated and its driver is loaded.

11.7.2 Before You Begin

11.7.2.1 Requirements

Before configuring SFs:

- Install the required BlueField software. For installation instructions, refer to the DOCA Installation Guide for Linux.
- Use firmware version 20.30.1004 or later.
- Enable the following Linux kernel configuration options:
  - `MLX5_ESWITCH`
  - `MLX5_SF`

- Verify that `mlxdevm` is available at `/opt/mellanox/iproute2/sbin/mlxdevm`. The tool is supplied with the `iproute2` package.

> Upstream kernels: If `mlxdevm` is unavailable on an upstream-based kernel, use `devlink` instead. Refer to the Upstream Guide on Scalable Functions.

11.7.2.2 Choose the Target Controller

Unless otherwise specified, run the `mlxdevm` management commands on the BlueField Arm system. The optional `controller` parameter selects the PCI controller on which the functional SF device is instantiated.

In a standard single-host BlueField DPU system:

Functional SF netdev Location	`controller` Argument
BlueField Arm system	Omit the argument; `controller 0` is used
External x86 host	Specify `controller 1`

The SF representor is created on the BlueField Arm system in both cases. The `netdev` shown in `mlxdevm port add` output is the SF representor, not the functional SF network interface.

In a multi-host configuration, use `mlxdevm port show` to inspect controller assignments and consult the platform-specific topology documentation to identify the controller associated with the target host PF.

> Controller 1 scope: This page documents the BlueField Arm-side driver-binding procedure for an SF created on `controller 0`. It does not define any additional host-side driver-binding steps that might be required for `controller 1`. Use the procedure documented for the target software release before applying driver-binding commands on the external host.

11.7.2.3 Understand the SF Identifiers

The procedure uses several identifiers. Record them as you create and deploy the SF.

Identifier	Example	Purpose
PCI device address	`0000:03:00.0`	Identifies the parent PCIe function
SF number	`4`	User-selected value passed as `sfnum`
SF port index	`229409`	Returned by `mlxdevm`; used to configure, activate, and delete the SF
SF representor	`en3f0pf0sf4`	Used for eSwitch, OVS, or another forwarding configuration
Auxiliary-device name	`mlx5_core.sf.4`	Used for driver binding and application initialization

Some **mlxdevm** commands accept either **pci/<pci_address>/<sf_port_index>** or the SF representor name. This page uses the PCI address and SF port index so that the identifier is explicit.

11.7.3 Configure SF Resources

Run the following command separately for each PF or embedded CPU physical function (ECPF) that will create SFs. The **-d <pci_address>** argument selects the function to configure. The same configuration parameter names apply to PFs and ECPFs.

```
mlxconfig -d 0000:03:00.0 set \
    PF_BAR2_ENABLE=0 \
    PER_PF_NUM_SF=1 \
    PF_TOTAL_SF=236 \
    PF_SF_BAR_SIZE=10
```

⚠ These settings affect SF resource allocation for the BlueField DPU and host. Do not enable **PF_BAR2_ENABLE** and **PER_PF_NUM_SF** at the same time. The preceding example disables **PF_BAR2_ENABLE** and enables **PER_PF_NUM_SF**.

Parameter	Description
PF_BAR2_ENABLE	Uses the deprecated configuration in which PFs and ECPFs receive the same number of SFs. When enabled, **PF_TOTAL_SF** and **PF_SF_BAR_SIZE** do not apply.
PER_PF_NUM_SF	Enables separate SF resource allocation for each PF or ECPF.
PF_TOTAL_SF	Specifies the maximum number of SFs that can be created by the function selected with **-d <pci_address>**.
PF_SF_BAR_SIZE	Specifies the BAR2 allocation for each SF created by the function selected with **-d <pci_address>**. The value is expressed as a power of two in KB. For example, **10** allocates 1 MB per SF.

For example, **PF_SF_BAR_SIZE=10** and **PF_TOTAL_SF=14** allocate 14 MB of BAR2 in total.

Perform a BlueField system-level reset for the **mlxconfig** settings to take effect.

11.7.4 Create and Deploy an SF

The SF lifecycle consists of creating the SF port, configuring and activating the SF, and deploying the functional driver. The following procedure separates those lifecycle stages into discrete tasks so that each identifier and execution environment is clear.

```
Create SF port
    --> Configure and activate SF
        --> Identify auxiliary device
            --> Bind functional driver
```

```
                --> Verify and use SF
```

Unless otherwise specified, run the commands in this section on the BlueField Arm system.

11.7.4.1 Create the SF Port

Create an SF by adding a port with the **pcisf** flavor:

```
/opt/mellanox/iproute2/sbin/mlxdevm port add pci/<pci_address> flavour pcisf pfnum <pf_number> sfnum <sf_number>
```

Each SF must have a unique SF number. SF numbers 1000 and later are reserved for the virtio-net controller.

The following example creates SF 4 on PF 0 of the local BlueField Arm-side controller. Because the **controller** argument is omitted, **controller 0** is used:

```
/opt/mellanox/iproute2/sbin/mlxdevm port add pci/0000:03:00.0 flavour pcisf pfnum 0 sfnum 4
```

Illustrative output:

```
pci/0000:03:00.0/229409: type eth netdev en3f0pf0sf4 flavour pcisf controller 0 pfnum 0 sfnum 4
    function:
      hw_addr 00:00:00:00:00:00 state inactive opstate detached roce true max_uc_macs 128 trust off
```

Record the following values from the output:

- SF number: **4**
- SF port index: **229409**
- SF representor: **en3f0pf0sf4**
- Controller: **0**

To inspect all created SF ports and their current attributes, run:

```
/opt/mellanox/iproute2/sbin/mlxdevm port show
```

To create an SF on PF 0 of the external host controller, specify **controller 1** :

```
/opt/mellanox/iproute2/sbin/mlxdevm port add pci/0000:03:00.0 flavour pcisf pfnum 0 sfnum 88 controller 1
```

Example output:

```
pci/0000:03:00.0/32768: type eth netdev eth6 flavour pcisf controller 1 pfnum 0 sfnum 88 splittable false
    function:
      hw_addr 00:00:00:00:00:00 state inactive opstate detached
```

In this output, **eth6** is the SF representor on the BlueField Arm system. It is not the functional SF network interface on the external host.

11.7.4.2 Configure and Activate the SF

Configure the SF through its management interface before binding the functional driver. The following command sets the MAC address, enables trust mode, and activates the SF:

```
/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/<pci_address>/<sf_port_index> \
    hw_addr <mac_address> \
    trust on \
    state active
```

For example:

```
/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/229409 \
    hw_addr 02:25:f2:8d:a2:4c \
    trust on \
    state active
```

A trusted function has additional privileges, such as the ability to update the steering database.

> **Changing trust mode:** To change the trust mode after activation, first set the SF state to **inactive**, change the trust setting, and then reactivate the SF.

Set all required function attributes before binding the functional SF driver.

11.7.4.3 Identify the Auxiliary Device

After activating an SF created on **controller 0**, identify the auxiliary device that corresponds to the SF before binding its functional driver.

List the available devices:

```
devlink dev show
```

Example output after creating and activating the SF:

```
pci/0000:03:00.0
pci/0000:03:00.1
auxiliary/mlx5_core.sf.2
auxiliary/mlx5_core.sf.3
auxiliary/mlx5_core.sf.4
```

The numeric suffix in **auxiliary/mlx5_core.sf.<serial>** is the auxiliary-device serial. Do not assume that this serial is the same as the SF number. Match the auxiliary device to the SF by reading its **sfnum** file:

```
cat /sys/bus/auxiliary/devices/mlx5_core.sf.<serial>/sfnum
```

For example:

```
cat /sys/bus/auxiliary/devices/mlx5_core.sf.4/sfnum
```

Example output:

```
4
```

This output confirms that **mlx5_core.sf.4** corresponds to SF number 4.

11.7.4.4 Bind the SF Driver

> **Controller 0 only:** The following procedure applies to an SF created on the BlueField Arm-side controller (**controller 0**). Run these commands on the BlueField Arm system.

Unbind the SF from the default configuration driver and bind it to the functional SF driver:

```
echo mlx5_core.sf.<serial> > /sys/bus/auxiliary/drivers/mlx5_core.sf_cfg/unbind
echo mlx5_core.sf.<serial> > /sys/bus/auxiliary/drivers/mlx5_core.sf/bind
```

For example:

```
echo mlx5_core.sf.4 > /sys/bus/auxiliary/drivers/mlx5_core.sf_cfg/unbind
echo mlx5_core.sf.4 > /sys/bus/auxiliary/drivers/mlx5_core.sf/bind
```

11.7.4.5 Verify the SF

Run the following commands on the BlueField Arm system:

```
/opt/mellanox/iproute2/sbin/mlxdevm port show
devlink dev show
ip link show
```

Verify the following:

- **mlxdevm port show** lists the expected SF number, controller, representor, MAC address, trust mode, and state.
- **devlink dev show** lists the auxiliary device matched to the SF.
- The SF representor exists on the BlueField Arm system.
- For an SF created on **controller 0** , the functional SF network interface exists on the BlueField Arm system after driver binding.
- For an SF created on **controller 1** , verify the functional network interface on the external host after completing the host-side deployment procedure documented for the target software release.

11.7.5 Use an SF

The functional SF network interface and the SF representor serve different purposes.

Interface	Location	Purpose
Functional SF network interface	Selected controller	Used by the workload or application; configure the workload IP address here
SF representor	BlueField Arm system	Used for eSwitch forwarding configuration, such as OVS or a Linux bridge

Do not configure the workload IP address on the SF representor.

11.7.6 Example: Run an Application Using Two SFs and OVS

This example creates two SFs on **controller 0**, connects their representors to two OVS bridges, and initializes their auxiliary devices in a DOCA application. Run all commands in this example on the BlueField Arm system.

The example creates the following forwarding paths:

```
p0      <--> sf_bridge1 <--> en3f0pf0sf4
pf0hpf  <--> sf_bridge2 <--> en3f0pf0sf5
```

Two SFs associated with the same PCIe function are required for this example.

11.7.6.1 Prepare SFs 4 and 5

Create the SF ports:

```
/opt/mellanox/iproute2/sbin/mlxdevm port add pci/0000:03:00.0 flavour pcisf pfnum 0 sfnum 4
/opt/mellanox/iproute2/sbin/mlxdevm port add pci/0000:03:00.0 flavour pcisf pfnum 0 sfnum 5
```

Run the following command and record the SF port index and representor name returned for each SF:

```
/opt/mellanox/iproute2/sbin/mlxdevm port show
```

Configure and activate the two SFs by using the actual SF port indices returned by the command:

```
/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/<sf4_port_index> \
    hw_addr 02:25:f2:8d:a2:4c trust on state active

/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/<sf5_port_index> \
    hw_addr 02:25:f2:8d:a2:5c trust on state active
```

For each SF, complete the steps in Identify the Auxiliary Device and Bind the SF Driver. Record the auxiliary-device serial assigned to each SF.

11.7.6.2 Remove an Existing Bridge, if Necessary

This optional example removes an existing bridge named **ovsbr1**:

```
ovs-vsctl del-br ovsbr1
```

11.7.6.3 Create the OVS Bridges

Create **sf_bridge1** and **sf_bridge2**, and add the physical-port and host-PF representors:

```
ovs-vsctl add-br sf_bridge1
ovs-vsctl add-br sf_bridge2
ovs-vsctl add-port sf_bridge1 p0
```

```
ovs-vsctl add-port sf_bridge2 pf0hpf
```

11.7.6.4 Add the SF Representors

Add each SF representor to its bridge:

```
ovs-vsctl add-port sf_bridge1 en3f0pf0sf4
ovs-vsctl add-port sf_bridge2 en3f0pf0sf5
```

If either representor is down, bring it up:

```
ip link set en3f0pf0sf4 up
ip link set en3f0pf0sf5 up
```

11.7.6.5 Verify the OVS Configuration

Run:

```
ovs-vsctl show
```

Example output:

```
Bridge sf_bridge1
    Port p0
        Interface p0
    Port sf_bridge1
        Interface sf_bridge1
            type: internal
    Port en3f0pf0sf4
        Interface en3f0pf0sf4
Bridge sf_bridge2
    Port sf_bridge2
        Interface sf_bridge2
            type: internal
    Port en3f0pf0sf5
        Interface en3f0pf0sf5
    Port pf0hpf
        Interface pf0hpf
ovs_version: "2.14.1"
```

11.7.6.6 Run the Application

Initialize the two SF auxiliary devices when starting the application:

```
<executable_binary> \
    -a auxiliary:mlx5_core.sf.<sf4_serial> \
    -a auxiliary:mlx5_core.sf.<sf5_serial>
```

For example, if the assigned auxiliary-device serials are 4 and 5:

```
doca_<app_name> \
    -a auxiliary:mlx5_core.sf.4 \
    -a auxiliary:mlx5_core.sf.5 \
    -- [application_flags]
```

11.7.7 Manage SFs

Unless otherwise specified, run the commands in this section on the BlueField Arm system.

11.7.7.1 List and Inspect SFs

To list SF ports and function attributes, run:

```
/opt/mellanox/iproute2/sbin/mlxdevm port show
```

To list PCI and SF auxiliary devices, run:

```
devlink dev show
```

To identify which SF number corresponds to an auxiliary device, run:

```
cat /sys/bus/auxiliary/devices/mlx5_core.sf.<serial>/sfnum
```

11.7.7.2 Remove an SF

If the SF representor is connected to an OVS bridge, remove it from the bridge first:

```
ovs-vsctl del-port <bridge_name> <sf_representor>
```

Set the SF state to inactive:

```
/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/<pci_address>/<sf_port_index> state inactive
```

Delete the SF port:

```
/opt/mellanox/iproute2/sbin/mlxdevm port del pci/<pci_address>/<sf_port_index>
```

For example:

```
ovs-vsctl del-port sf_bridge1 en3f0pf0sf4
/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/229409 state inactive
/opt/mellanox/iproute2/sbin/mlxdevm port del pci/0000:03:00.0/229409
```

11.7.8 Use devlink on an Upstream Kernel

For an upstream-based kernel on which **mlxdevm** is unavailable, follow the Upstream Guide on Scalable Functions to perform the equivalent operations with **devlink**.

11.7.9 Related Documentation

- DOCA Installation Guide for Linux
- NVIDIA BlueField Reset and Reboot Procedures
- BlueField Virtio-net Controller
- Upstream Guide on Scalable Functions

11.8 DOCA TLS Offload

This guide provides an overview and configuration steps of TLS hardware offloading via kernel-TLS, using hardware capabilities of NVIDIA® BlueField® DPU.

11.8.1 1. Introduction

Transport layer security (TLS) is a cryptographic protocol designed to provide communications security over a computer network. The protocol is widely used in applications such as email, instant messaging, and voice over IP (VoIP), but its use in securing HTTPS remains the most publicly visible.

The TLS protocol aims primarily to provide cryptography, including privacy (confidentiality), integrity, and authenticity using certificates, between two or more communicating computer applications. It runs in the application layer and is itself composed of two layers: the TLS record and the TLS handshake protocols.

TLS works over TCP and consists of 3 phases:

1. Handshake – establishment of a connection
2. Application – sending and receiving encrypted packets
3. Termination – connection termination

11.8.1.1 1.1. TLS Handshake

In the handshake phase, the client and server decide on which cipher suites they will use, and exchange keys and certificates according to the following flow:

1. Client hello, provides the server at a minimum with the following:
    - A key exchange algorithm, to determine how symmetric keys are exchanged
    - An authentication or digital signature algorithm, which dictates how server authentication and client authentication (if required) are implemented
    - A bulk encryption cipher, which is used to encrypt the data
    - A hash/MAC (message authentication code) function, which determines how data integrity checks are carried out
    - The version of the protocol it understands
    - The cipher suites it is capable of working with
    - A unique random number, which is important to guard against replay attacks
2. Server hello:
    - Selects a cipher suite
    - Generates its own random number
    - Assigns a session ID to the TLS connection
    - Sends enough information to complete a key exchange—most often, this means sending a certificate including an RSA public key
3. Client:
    - Responsible for completing the key exchange using the information the server provided

At this point, the connection is secured, both sides have agreed on an encryption algorithm, a MAC algorithm, and respective keys.

11.8.1.2 1.2. kTLS

The Linux kernel provides TLS offload infrastructure. kTLS (kernel TLS) offloads TLS handling from the user-space to the kernel-space.

kTLS has 3 modes of operation:

- SW – all operation is handled in kernel (i.e., handshake, encryption, decryption)
- HW-offload (the focus of this guide) – handshake and error handling are performed in software. Packets are encrypted/decrypted in hardware. In this case, there is an additional offload from the kernel to the hardware.
- HW-record – all operations are handled by the hardware (driver and firmware) including the handshake. It also handles its own TCP session. This option is currently not supported.

It is important to understand that Rx (receiving) and Tx (sending) can have two separate modes. For example, Rx can be dealt in SW mode but Tx in HW-offload mode (i.e., the hardware will only encrypt but not decrypt).

11.8.1.3 1.3. HW-offloading kTLS

In general, the TLS HW-offload performs best and provides optimal value on longer lived sessions, with relatively large packets. Scaling in terms of concurrent connections and connections per second is use-case dependent (e.g., the amount of active concurrent connections from the overall open concurrent connections is material).

It is necessary to learn the following terms before proceeding:

- The transport interface send (TIS) object is responsible for performing all transport-related operations of the transmit side. Messages from Send Queues (SQs) get segmented and transmitted by the TIS including all transport required implications. For example, in the case of a large send offload, the TIS is responsible for the segmentation. The NVIDIA® ConnectX® hardware uses a TIS object to save and access the TLS crypto information and state of an offloaded Tx kTLS connection.
- The transport interface receive (TIR) object is responsible for performing all transport-related operations on the receive side. TIR performs the packet processing and reassembly and is also responsible for demultiplexing the packets into different receive queues (RQs).
- Both TIS and TIR hold the data encryption key (DEK).

11.8.1.3.1 1.3.1. kTLS Offload Flow in High Level

The following flow does not include resync and errors.

1. Establishes a TLS connection with remote host (server or client) by handling a TLS handshake by kernel on current host.
2. Initializes the following state for each connection, Rx and Tx:
    - Crypto secrets (e.g., public key)

- Crypto processing state
- Record metadata (e.g., record sequence number, offset)
- Expected TCP sequence number

Tx flow:

1. Packets belonging to device offloaded sockets arrive to the kernel and it does not encrypt them.
2. Kernel performs record framing and marks the packet with a connection identifier.
3. Kernel sends packets to the device driver for offloading.
4. Device checks that the sequence number matches the state in the TIS and performs encryption and authentication.

Rx flow:

1. When the connection is created, a HW steering rule is added to steer packets to their respective TIR.
2. Device receives the packet then validates and checks that sequence number of TCP matches the state in the TIR.
3. Performs decryption and authentication, and indicates in the CQE (completion queue entry).
4. Kernel understands that the packet is already decrypted so it does not decrypt it itself and passes it on to the user-space.

11.8.1.3.2 1.3.2. Resync and Error Handling

When the sequence number does not match expectations or if any other error occurs, the hardware gives control back to the SW which handles the problem.

See more about kTLS modes, resync, and error handling in the Linux Kernel documentation.

11.8.2 2. Prerequisites

All commands in this section should be performed on host (not on BlueField) unless stated otherwise.

11.8.2.1 2.1. Checking Hardware Support for Crypto Acceleration

To verify cryptographic acceleration support for your BlueField or ConnectX device, consult the hardware user manual to confirm if your specific model is crypto-enabled.

- Documentation for BlueField devices
- Documentation for ConnectX devices

> ⚠ ConnectX-8 devices and the ConnectX-9 devices used in NVIDIA Vera Rubin NVL-72 systems do not support TLS acceleration, even when using crypto-enabled OPNs. Other crypto-enabled devices starting from ConnectX-6 Dx support TLS acceleration.

11.8.2.2 2.2. Kernel Requirements

- Operating system must be either:
  - FreeBSD 13.0+.
  - A Linux distribution built on Linux kernel version 5.3 or later for Tx support and version 5.9 or later for Rx support. We recommend using the latest version when possible for the best available optimizations.

    > ⚠ TIS Pool optimization is added to Linux kernel version 6.0. Instead of creating TIS per new connection, unused TIS from previous connection, will be recycled. This will improve Tx connection rate. No further installations required beyond installing the kernel itself.

- Check the current kernel version on the host. Run:

```
host> uname -r
```

- The kernel must be configured to support TLS by setting the options **TLS_DEVICE** and **MLX5_TLS** to **y**. To check if TLS is configured, run:

```
host> cat /boot/config-$(uname -r) | grep TLS
```

Example output:

```
host> cat /boot/config-5.4.0-121-generic | grep TLS
...
CONFIG_TLS_DEVICE=y
CONFIG_MLX5_TLS=y
...
```

If the current kernel does not support one of the options, you can change the configurations and recompile, or build a new kernel.

> ⚠ Follow the build instructions provided with the kernel provider.

Schematic flow for building a Linux kernel:

a. Enter the Linux kernel directory downloaded (usually in **/usr/src/**):

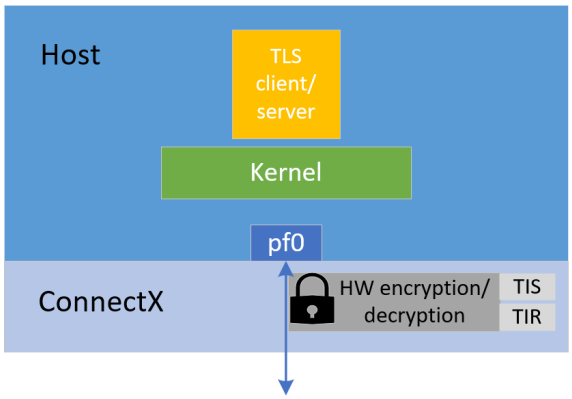
```
host> make menuconfig # Set TLS_DEVICE=y and MLX5_TLS=y in options. Setting location in the menu
can be found by pressing '/' and typing 'setting'.
host> make -j <num-of-cores> && make -j <num-of-cores> modules_install && make -j <num of cores>
install
```

b. Update the grub to the new configured kernel then reboot.

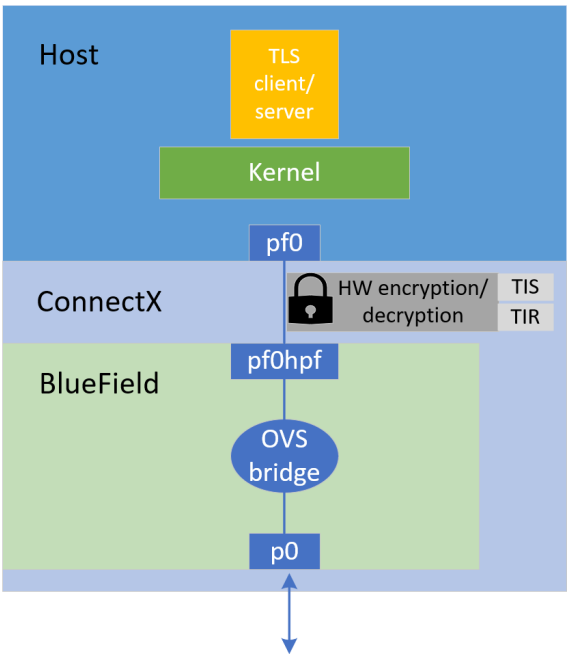
11.8.3 3. Configurations and Useful Commands

11.8.3.1 3.1. TLS Setup

Host + ConnectX Setup



Host + BlueField Setup



11.8.3.2 3.2. Finding NVIDIA Interfaces

```
host> mst start          # if mst driver is not loaded.
host> mst status -v
```

NVIDIA's netdev interfaces are found be under the **NET** column.

For example:

```
host> mst status -v
....
DEVICE_TYPE             MST                           PCI       RDMA            NET                     NUMA
BlueField2(rev:0)       /dev/mst/mt41686_pciconf0.1   b1:00.1   mlx5_1          net-ens5f1              1

BlueField2(rev:0)       /dev/mst/mt41686_pciconf0     b1:00.0   mlx5_0          net-ens5f0              1
```

In this example, the interfaces **ens5f1** and **ens5f0** are NVIDIA's netdev interfaces.

11.8.3.3 3.3. Configuring TLS Offload

- To check if the offload option is on or off, run:

```
host> ethtool -k $iface | grep tls
```

Example output:

```
tls-hw-tx-offload: on
tls-hw-rx-offload: off
tls-hw-record: off [fixed]
```

> ⚠ **tls-hw-record** is not required for the device as kTLS does not support "HW Record" mode.

- To turn Tx offload on or off:

```
host> ethtool -K $iface tls-hw-tx-offload <on | off>
```

- To turn Rx offload on or off:

```
host> ethtool -K $iface tls-hw-rx-offload <on | off>
```

11.8.3.4 3.4. Configuring OVS Bridge on BlueField

When the host is connected to a BlueField device, an OVS bridge must be configured on the BlueField so traffic passes bidirectionally from host to uplink. If no OVS bridge is configured, the host is isolated from the network (see diagram above).

> ⚠ On BlueField image version 3.7.0 or higher the default OVS configuration can be used without additional modifications.

To configure the OVS bridge on BlueField, run the following commands on BlueField:

```
dpu> for br in $(ovs-vsctl list-br); do ovs-vsctl del-br $br; done # erasing existing bridges
dpu> ovs-vsctl add-br ovs-br0 && ovs-vsctl add-port ovs-br0 p0 && ovs-vsctl add-port ovs-br0 pf0hpf
dpu> ovs-vsctl add-br ovs-br1 && ovs-vsctl add-port ovs-br1 p1 && ovs-vsctl add-port ovs-br1 pf1hpf
dpu> ovs-vsctl set Open_vSwitch . other_config:hw-offload=true && systemctl restart openvswitch-switch
```

Where **p0** / **p1** are the uplink interfaces and **pf0hpf** / **pf1hpf** are the interfaces facing the host.

11.8.4 4. Common Use Cases

11.8.4.1 4.1. OpenSSL

OpenSSL is an all-around cryptography library that offers open-source application of the TLS protocol. It is the main library for using kTLS and other applications since Nginx depends on it as their base library.

> ⚠ The kTLS and HW offloading do not depend on OpenSSL. Any program that can implement a TLS stack can be run instead. However, because of the vast use of OpenSSL, this guide addresses installation recommendations.

kTLS is supported only in OpenSSL version 3.2.0 or higher, and only on the supported kernel versions. The supported OpenSSL version is available for download from distro packages, or it can be downloaded and compiled from the OpenSSL GitHub.

Many modules depend on OpenSSL. Changing the default version may cause problems. Adding **--prefix=/var/tmp/ssl --openssldir=/var/tmp/ssl** in the **./Configure** command below may prevent the built OpenSSL from becoming the default one used by the system. Make sure the directory of the OpenSSL you build manually is not located in any paths listed in the PATH environment variable.

1. Check the version of the default OpenSSL:

```
host> openssl version
```

2. Follow OpenSSL installation instructions from OpenSSL's supplied guides. During the configuration process, make sure to set the **enable-ktls** option before building it by running it from within the OpenSSL directory (works in version 3.2.0 and higher). For example:

```
host> ./Configure linux-$(uname -p) enable-ktls --prefix=/var/tmp/ssl --openssldir=/var/tmp/ssl # Add
"threads" as well for multithread support
```

3. Check if kTLS is enabled in OpenSSL by running the following command from within the OpenSSL directory, and check whether **ktls** is listed under **Enabled features**:

```
host> perl configdata.pm --dump | less
```

If OpenSSL has been downloaded manually, the OpenSSL executable would be located in the **/<openssl-dir>/apps/** directory. For example, checking the version from within OpenSSL directory is done using the command **./apps/openssl version**.

Installing a new OpenSSL requires recompiling user tools that were configured over OpenSSL (e.g., Nginx).

In OpenSSL's master source code, there is a feature "Support for kTLS Zero-Copy sendfile() on Linux" (Zero-Copy commit). If the Zero-Copy option is set, **SSL_sendfile()** uses the Zero-Copy TX mode which means that the data itself is not copied from the user space to Kernel space. This gives a performance boost when used with kTLS hardware offload. Be aware that invalid TLS records may be transmitted if the file is changed while being sent.

11.8.4.2 4.2. Nginx

Nginx is a free and open-source software web server that can also be used as a reverse proxy, load balancer, mail proxy and HTTP cache. Nginx can be configured to depend on the OpenSSL library and therefore can use TLS hardware offload on supported crypto-enabled devices.

11.8.4.2.1 4.2.1. Prerequisites

Refer to the “OpenSSL” section for setting OpenSSL.

11.8.4.2.2 4.2.2. Configuration

1. Install dependencies. For Ubuntu distribution, for example:

```
host> apt install libpcre3 libpcre3-dev
```

2. Clone Nginx's repository and enter directory:

```
host> git clone https://github.com/nginx/nginx.git && cd nginx
```

3. Configure Nginx components to support kTLS:

```
host> ./auto/configure --with-openssl=/<insert_path_to_openssl_directory> --with-debug --with-
http_ssl_module --with-openssl-opt="enable-ktls -DOPENSSL_LINUX_TLS -g3"
```

4. Build Nginx:

```
host> make -j <num of cores> && sudo make -j <num-of-cores> install
```

> ⚠ If **make** fails with a **deprecated openssl functions** error, remove **-Werror** for **CFLAGS** in objs/Makefile and try again.

5. Add the following lines to the end of the **/usr/local/nginx/conf/nginx.conf** file (before the last closing bracket):

```
server {
    listen 443 ssl default_server reuseport;
    server_name localhost;
    root /tmp/nginx/docs/html/;

    include /etc/nginx/default.d/*.conf;
    ssl_certificate /usr/local/nginx/conf/cert.pem;
    ssl_certificate_key /usr/local/nginx/conf/key.pem;
    ssl_ciphers ECDHE-RSA-AES128-GCM-SHA256;
    ssl_conf_command Options KTLS;
    ssl_protocols TLSv1.2;

 location / {
    index index.html;
 }

 error_page 404 /404.html;
    location = /40x.html {
 }

 error_page 500 502 503 504 /50x.html;
    location = /50x.html {
 }
}
```

6. Notice that the key and certificate of the Nginx server should be located in **/usr/local/nginx/conf/** . Therefore, after creating a key and certificate (as mentioned in section "DOCA TLS Offload#Adding Certificate and Key") they should be copied to the aforementioned directory:

```
host> cp key.pem /usr/local/nginx/conf/ && cp cert.pem /usr/local/nginx/conf/
```

7. To run Nginx:

```
host> cd nginx && objs/nginx
```

This command starts Nginx Server in the background.

11.8.4.2.3 4.2.3. Stopping Nginx

```
host> pkill nginx
```

11.8.4.2.4 4.2.4. Wrk – Client

A simple client for requesting Nginx's server is "wrk". It can be installed by running the following:

```
host> git clone https://github.com/wg/wrk.git && cd wrk/ && make -j <num-of-cores>
```

11.8.4.2.5 4.2.5. Using Wrk

The following is an example of using the wrk client to request the page **index.html** from the Nginx server in address **4.4.4.4** (run within wrk's directory):

```
host> taskset -c 0 ./wrk -t1 -c10 -d30s https://4.4.4.4:443/index.html
```

⚠ Testing the kTLS offload (with or without hardware offload) is in the same manner as mentioned in section "Testing kTLS". TBD

11.8.5 5. Testing Offload via OpenSSL

This chapter demonstrates how to test the kTLS hardware offload.

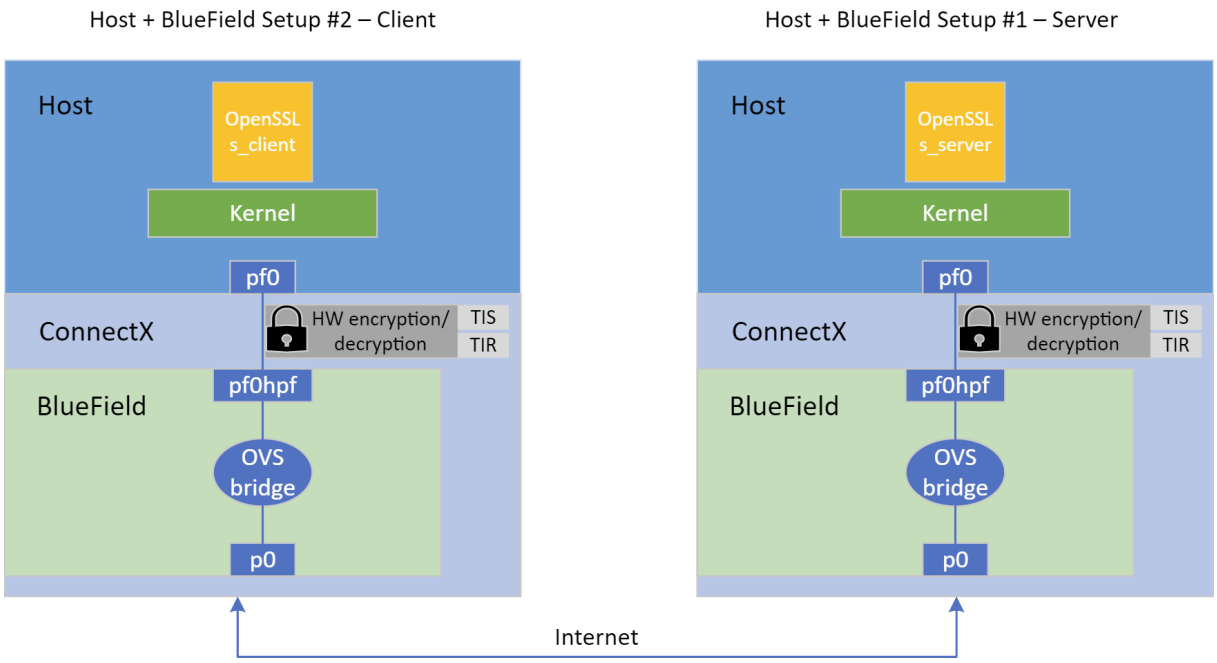
⚠ Make sure to refer to section "DOCA TLS Offload#OpenSSL" before proceeding.

11.8.5.1 5.1. TLS Testing Setup

For testing purposes, a server and a client are required. The testing section only tests a single setup of a host and BlueField-2 or a host ConnectX which will participate either as a server or as a client. Setting a back-to-back setup of the same kind and installing the same OpenSSL version can help avoid misconfigurations. Nevertheless, it is required to have the same OpenSSL version on both the client and server.

Make sure the desired kTLS is configured as detailed in section "DOCA TLS Offload#Configuring TLS Offload". To test hardware offload, make sure **tls-hw-tx-offload** and/or **tls-hw-rx-offload** are on. To test kTLS software mode, make sure to turn them off.

In addition, make sure both hosts (server and client) can communicate bidirectionally through ConnectX or BlueField. One can set the interface that supports the offload (on the host) with an IP, in same subnet. Make sure that when using BlueField, an OVS bridge is set on BlueField as shown in "Configuring OVS Bridge on BlueField".



11.8.5.2 5.2. Adding Certificate and Key

The server side should create a certificate and key. The client can also use a certificate, but it is not necessary for this test case. Run the following command in the installed OpenSSL directory and fill in all the requested details:

```
host> openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem -days 365 -nodes
```

The following files are created:

- **key.pem** – private-key file used to generate the CSR and, later, to secure and verify connections using the certificate
- **cert.pem** – certificate signing request (CSR) file used to order your SSL certificate and, later, to encrypt messages that only its corresponding private key can decrypt

The server side should be run before client side so that client's request are answered by server.

11.8.5.3 5.3. Running Server Side

The following example works on OpenSSL version 3.1.0:

```
host> openssl s_server -key key.pem -cert cert.pem -tls1_2 -cipher ECDHE-RSA-AES128-GCM-SHA256 -accept 443 -ktls
```

Notice the **-ktls** flag.

⚠ Refer to official OpenSSL documentation on **s_server** for more information.

In this example, the key and certificate are provided, the cipher suite and TLS version are configured, and the server listens to port 443 and is instructed to use kTLS.

11.8.5.4 5.4. Running Client Side

The following example works on OpenSSL version 3.1.0:

```
host> openssl s_client -connect 4.4.4.4:443 -tls1_2 -ktls
```

Where 4.4.4.4 is the IP of the port/interface under test (e.g., **enp177s0f0np0**).

⚠ Refer to official OpenSSL documentation on **s_client** for more information.

11.8.5.5 5.5. Testing kTLS

After the connection is established (handshake is done), a prompt will open and the user, both on the client and server side, can send a message to other side in a chat-like manner. Messages should appear on the other side once they are received.

The following example checks kTLS hardware offload on the tested setup by tracking Rx and Tx TLS on device counters:

```
host> ethtool -S $iface | grep -i 'tx_tls_encrypted\|rx_tls_decrypted' # ($iface is the interface that offloads)
```

To check kTLS over kernel counters:

```
host> cat /proc/net/tls_stat
```

Output example:

⚠ The comments are not part of the output and are added as explanation.

```
host> cat /proc/net/tls_stat
TlsCurrTxSw                             0               # Current Tx connections opened in SW mode
TlsCurrRxSw                             0               # Current Rx connections opened in SW mode
TlsCurrTxDevice                         0               # Current Tx connections opened in HW-offload mode
TlsCurrRxDevice                         0               # Current Rx connections opened in HW-offload mode
TlsTxSw                                 2323828         # Accumulated number of Tx connections opened in SW mode
TlsRxSw                                 1               # Accumulated number of Rx connections opened in SW mode
TlsTxDevice                             12203652        # Accumulated number of Tx connections opened in HW-offload
mode
TlsRxDevice                             0               # Accumulated number of Rx connections opened in HW-offload
mode
TlsDecryptError                         0               # Failed record decryption (e.g., due to incorrect
authentication tag)
TlsRxDeviceResync                       0               # Rx resyncs sent to HW's handling cryptography
TlsDecryptRetry                         0               # All Rx records re-decrypted due to TLS_RX_EXPECT_NO_PAD
misprediction
TlsRxNoPadViolation                     0               # Data Rx records re-decrypted due to TLS_RX_EXPECT_NO_PAD
misprediction
```

More information about the kernel counters can be found in the Statistics section of the Kernel TLS documentation.

11.8.6 6. Optimizations over kTLS

11.8.6.1 6.1. XLIO

The NVIDIA accelerated IO (XLIO) software library boosts the performance of TCP/IP applications based on Nginx (e.g., CDN, DoH) and storage solutions as part of SPDK. XLIO is a user-space software library that exposes standard socket APIs with kernel-bypass architecture, enabling a hardware-based direct copy between an application's user-space memory and the network interface. In particular, XLIO can boost the performance of applications that use the kTLS hardware offload as OpenSSL and Nginx. Read more about XLIO in the NVIDIA XLIO Documentation and XLIO TLS HW-offload over kTLS in the TLS HW Offload section.

Even though XLIO is a kernel-bypass library, the kernel must support kTLS for the bypass to work properly.

11.8.7 7. Performance Tuning Options

TLS offload performance is related to how fast data can be pumped though the offload engine. In the case of user space applications, certain system configurations can be tuned to optimize its performance.

The following are items that can be tuned for optimal performance, mainly focusing on dedicating the server's work to the NUMA, or non-uniform memory access, cores:

Non-uniform memory access (NUMA) cores are cores with a dedicated memory for each of them, granting cores fast access to their own memory and slower access to others'. This architecture is best for scenarios when it is not necessary to share memory between cores.

1. Add NUMA cores of the NIC to the **isolcpus** kernel boot arguments for each server so that the kernel scheduler does not interrupt the core's running user thread. The following are examples of adding commands:
    a. Identify the NIC NUMA node (see NUMA column):

```
host> mst status -v
DEVICE_TYPE          MST                       PCI        RDMA        NET                 NUMA
ConnectX6DX(rev:0)   /dev/mst/mt4125_pciconf0 41:00.0    mlx5_0      net-enp65s0f0np0    1
```

    b. Identify the cores of the NIC NUMA node using the NUMA node number acquired from the previous output:

```
host> lscpu | grep "NUMA node1"
NUMA node1 CPU(s): 1,3,5,7,9,11,13,15,17,19,21,23
```

c. Add the NIC NUMA cores to a grub file (e.g., **/etc/default/grub**) by adding the line **GRUB_CMDLINE_LINUX_DEFAULT="isolcpus=<NUMA-cores-from-previous-output>"**. For example:

```
GRUB_CMDLINE_LINUX_DEFAULT="isolcpus=1,3,5,7,9,11,13,15,17,19,21,23"
```

d. Update grub:

```
host> sudo update-grub
```

e. Reboot and check that the configuration has been applied:

```
host> cat /proc/cmdline
BOOT_IMAGE=/vmlinuz-5.10.12 root=UUID=1879326c-711f-4f95-a974-d732af14ef04 ro department=general
user_notifier=dovd osi_string None BOOTIF=01-90-b1-1c-14-02-44 quiet splash
isolcpus=1,3,5,7,9,11,13,15,17,19,21,23
```

2. Disable **irqbalance** service:

> ⚠ Interrupt request, or IRQ, determines what hardware interrupts arrive to each core.

```
host> service irqbalance stop
```

3. Run **set_irq_affinity.sh** to redistribute IRQs to various cores.

> ⚠ The script is within MLNX_OFED's sources:
>
> a. You can find it in MLNX_OFED downloads.
>
> b. Under "Download" select the correct version and download the "SOURCES" **.tgz** file.
>
> c. Extract the **.tgz**.
>
> d. Under SOURCES, extract the **mlnx_tools**.
>
> You should find both files **set_irq_affinity.sh** and its helper file **common_irq_affinity.sh** under the **sbin** directory.

```
host> ./set_irq_affinity.sh <ConnectX_or_BlueField_network_interface>
```

4. Set the interface RSS to the number of cores to use:

```
host> ethtool -X <ConnectX_or_BlueField_network_interface> equal <number_of_isolcpus_cores>
```

5. Set the interface queues for number of cores to use:

```
host> ethtool -L <ConnectX_or_BlueField_network_interface> combined <number_of_isolcpus_cores>
```

6. Pin the application with **taskset** to the **isolcpus** cores used. For example:

```
host> taskset -c 1,3,5,7,9,11,13,15,17,19,21,23 openssl s_server -key key.pem -cert cert.pem -tls1_2
-cipher ECDHE-RSA-AES128-GCM-SHA256 -accept 443 -ktls
```

11.8.8 8. Additional Reading

- Linux kernel TLS documentation
- Linux kernel TLS offload documentation
- Autonomous NIC offloads research paper

11.9 DOCA Virtual Functions

This guide provides an overview and configuration of virtual functions for NVIDIA® BlueField® and demonstrates a use case for running the DOCA applications over x86 host.

11.9.1 Introduction

Single root IO virtualization (SR-IOV) is a technology that allows a physical PCIe device to present itself multiple times through the PCIe bus. This technology enables multiple virtual instances of the device with separate resources. NVIDIA adapters are able to expose virtual instances or functions (VFs) for each port individually. These virtual functions can then be provisioned separately.

Each VF can be seen as an additional device connected to the physical interface or function (PF). It shares the same resources with the PF, and its number of ports equals those of the PF.

SR-IOV is commonly used in conjunction with an SR-IOV-enabled hypervisor to provide virtual machines direct hardware access to network resources, thereby increasing its performance.

There are several benefits to running applications on the host. For example, one may want to utilize a strong and high-resource host machine, or to start DOCA integration on the host before offloading it to the BlueField DPU.

The configuration in this document allows the entire application to run on the host's memory, while utilizing the HW accelerators on BlueField.

When VFs are enabled on the host, VF representors are visible on the Arm side which can be bridged to corresponding PF representors (e.g., the uplink representor and the host representor). This allows the application to only scan traffic forwarded to the VFs as configured by the user and to behave as a simple "bump-on-the-wire". DOCA installed on the host allows access to the hardware capabilities of the BlueField DPU without comprising features which use HW offload/ steering elements embedded inside the eSwitch.

11.9.2 Prerequisites

To run all the reference applications over the host, you must install the host DOCA package. Refer to the DOCA Installation Guide for Linux for more information on host installation.
VFs must be configured as trusted for the hardware jump action to work as intended. The following steps configure "trusted" mode for VFs:

1. Delete all existing VFs
    a. To delete all VFs on a PF run the following on the host:

```
host $ echo 0 > /sys/class/net/<physical_function>/device/sriov_numvfs
```

For example:

```
host $ echo 0 > /sys/class/net/ens1f0/device/sriov_numvfs
```

2. Delete all existing SFs.

   Refer to BlueField Scalable Functions for instructions on deleting SFs.

3. Stop the main driver on the host:

```
host $ /etc/init.d/openibd stop
```

4. Before creating the VFs, set them to "trusted" mode on the device by running the following commands on the DPU side.
   a. Setting VFs on port 0:

```
host $ mlxreg -d /dev/mst/mt41686_pciconf0 --reg_id 0xc007 --reg_len 0x40 --indexes
"0x0.0:32=0x80000000" --yes --set "0x4.0:32=0x1"
```

   b. Setting VFs on port 1:

```
host $ mlxreg -d /dev/mst/mt41686_pciconf0.1 --reg_id 0xc007 --reg_len 0x40 --indexes
"0x0.0:32=0x80000000" --yes --set "0x4.0:32=0x1"
```

   These commands set trusted mode for all created VFs/SFs after their execution on the DPU.

   Setting trusted mode should be performed once per reboot.

5. Restart the main driver on the host by running the following command:

```
host $ /etc/init.d/openibd restart
```

11.9.3 VF Creation

1. Make sure mst driver is running:

```
host $ sudo mst status
```

   If it is not loaded, run:

```
host $ sudo mst start
```

2. Enable SR-IOV. Run:

```
host $ sudo mlxconfig -y -d /dev/mst/mt41686_pciconf0 s SRIOV_EN=1
```

3. Set number of VFs. Run:

```
host $ sudo mlxconfig -y -d /dev/mst/mt41686_pciconf0 s NUM_OF_VFS=X
```

⚠ Perform a BlueField system reboot for the **mlxconfig** settings to take effect.

```
host $ echo X > /sys/class/net/<physical_function>/device/sriov_numvfs
```

For example:

```
host $ sudo mlxconfig -y -d /dev/mst/mt41686_pciconf0 s NUM_OF_VFS=2
host $ reboot
host $ echo 2 > /sys/class/net/ens1f0/device/sriov_numvfs
```

After enabling VF, the representor appears on the DPU. The function itself is seen at the x86 side.

4. To verify that the VFs have been created. Run:

```
host $ lspci | grep Virtual
b1:00.3 Ethernet controller: Mellanox Technologies ConnectX Family mlx5Gen Virtual Function (rev 01)
b1:00.4 Ethernet controller: Mellanox Technologies ConnectX Family mlx5Gen Virtual Function (rev 01)
b1:01.3 Ethernet controller: Mellanox Technologies ConnectX Family mlx5Gen Virtual Function (rev 01)
```

⚠ 2 new virtual Ethernet devices are created in this example.

5. It is advisable to limit VF firmware (ICM) pages (e.g., via VF **page_limit** sysfs) to avoid uncontrolled memory growth and host resource exhaustion. For example, this command prevents a VF from allocating more than the configured page limit:

```
echo X  >  /sys/class/infiniband/mlx_$/device/sriov/$vf_id/paging_control/page_limit
```

11.9.4 Running DOCA Application on Host

⚠ Allocate the required number of VFs as explained previously.

⚠ Allocate any other resources as specified by the application (e.g., huge pages).

The following is the CLI example for running a reference application over the host using VF:

```
doca_<app_name> -a "pci address VF0" -a "pci address VF1" -c 0xff -- [application flags]
```

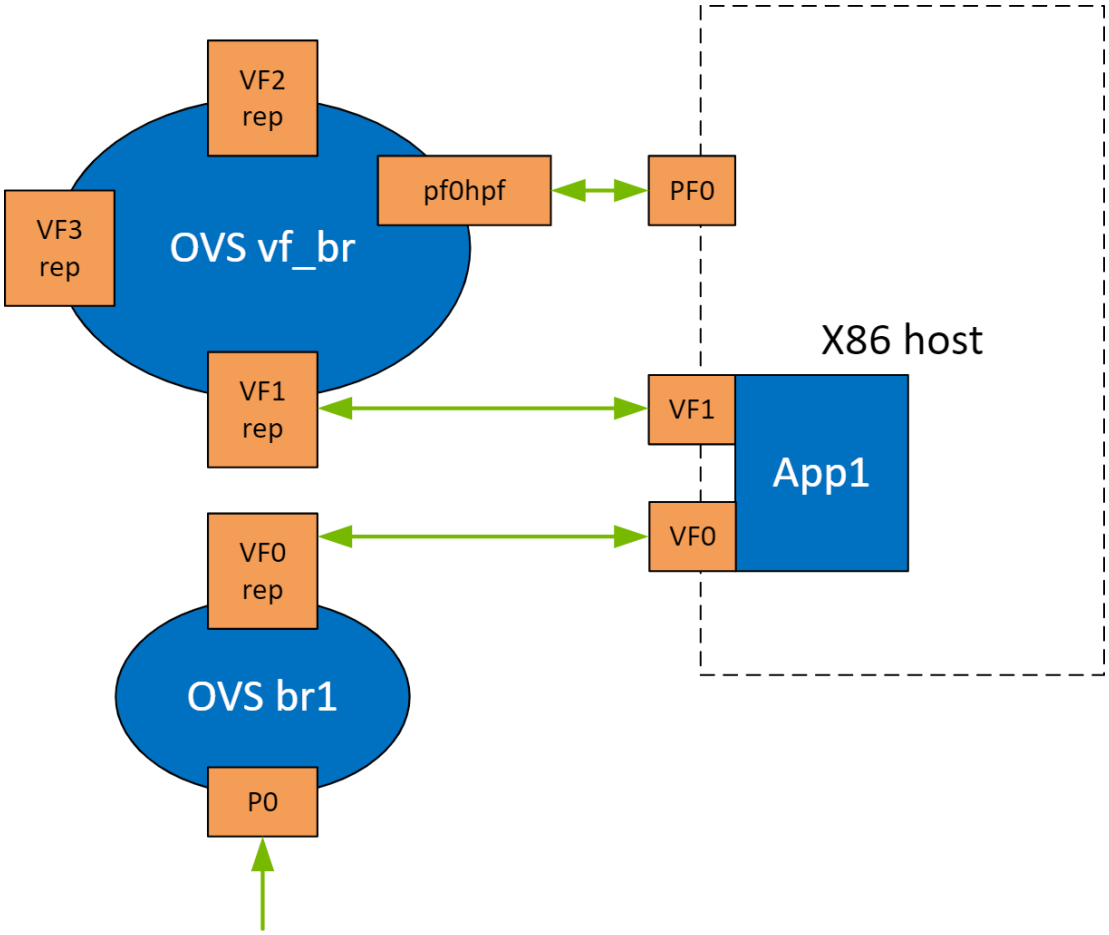
The following is an example with specific PCIe addresses for the VFs:

```
doca_<app_name> -a b1:00.3 -a b1:00.4 -c 0xff -- -l 60
```

By default, a DPDK application initializes all the cores of the device. This is usually unnecessary and may even cause unforeseeable issues. It is recommended to limit the number of cores, especially when using an AMD-based system, to 16 cores using the `-c` flag when running DPDK.

11.9.5 Topology Example

The following is a topology example for running the application over the host.



Configure the OVS on BlueField as follows:

```
Bridge vf_br:
    Port vf_br
        Interface vf_br
            type: internal
    Port pf0hpf
        Interface pf0hpf
   Port pf0vf1
        Interface pf0vf1
Bridge ovsbr1
   Port ovsbr1
        Interface ovsbr1
            type: internal
   Port p0
        Interface p0
   Port pf0vf0
        Interface pf0vf0
```

When enabling a new VF over the host, VF representors are created on the Arm side. The first OVS bridge connects the uplink connection (**p0**) to the new VF representor (**pf0vf0**), and the second bridge connects the second VF representor (**pf0vf1**) to the host representors (**pf0phf**). On the host, the 2 PCIe addresses of the newly created function must be initialized when running the applications.

When traffic is received (e.g., from the uplink), the following occurs:

1. Traffic is received over **p0** .
2. Traffic is forwarded to **pf0vf0** .
3. Application "listens" to **pf0vf0** and **pf0vf1** and can, therefore, acquire the traffic from **pf0vf0** , inspect it, and forward to **pf0vf1** .
4. Traffic is forwarded from **pf0vf1** to **pf0hpf** .

11.9.6 VF Creation on Adapter Card

⚠ Supported only for NVIDIA® ConnectX®-6 Dx based adapter cards and higher.

The following steps are required only when running DOCA applications on an adapter card.

1. Set trust level for all VFs. Run:

```
host $ mlxreg -d /dev/mst/mt4125_pciconf0 --reg_name VHCA_TRUST_LEVEL --yes --set
"all_vhca=0x1,trust_level=0x1" --indexes "vhca_id=0x0,all_vhca=0x0"
```

2. Create X VFs (X being the required number of VFs) and run the following to turn on trusted mode for the created VFs:

```
echo ON | tee /sys/class/net/enp1s0f0np0/device/sriov/X/trust
```

   For example, if you are creating 2 VFs, the following commands should be used:

```
echo ON | tee /sys/class/net/enp1s0f0np0/device/sriov/0/trust
echo ON | tee /sys/class/net/enp1s0f0np0/device/sriov/1/trust
```

3. Create a VF representor using the following command, replace the PCIe address with the PCIe address of the created VF:

```
echo 0000:17:00.2 > /sys/bus/pci/drivers/mlx5_core/unbind
echo 0000:17:00.2 > /sys/bus/pci/drivers/mlx5_core/bind
```

11.10 BlueField SR-IOV

This page explains the NVIDIA BlueField SR-IOV solution, detailing how to configure asymmetric Virtual Functions (VFs) per Embedded CPU Function (ECPF) and host Physical Function (PF) to enable scalable network virtualization.

11.10.1 Introduction

The NVIDIA® BlueField® SR-IOV solution is based on asymmetric VF and enables per-ECPF and per PF control over number of VF allocation.

ECPF VFs are intended to be used in switchdev mode. Like SFs and host VFs, ECPF VFs have a representor. Representor naming for ECPF VFs start after the host VFs. For example, if the host has 32 VFs enabled, then the host VF representors are named **pf0vf0** - **pf0vf31**, and the Arm representors continue at **pf0vf32** onward.

11.10.2 Enabling BlueField SR-IOV

To enable BlueField SR-IOV, apply the following configuration in the BlueField OS:

```
mlxconfig -d 03:00.0 -y s PF_NUM_OF_VF_VALID=1
```

⚠ Once **PF_NUM_OF_VF_VALID** is set, the **NUM_OF_VFS** mlxconfig option is not relevant and the user must set **PF_NUM_OF_VF** for each host and EC function. It is recommended for the number of VFs for each ECPF and each host PF be the same.

The BlueField should now support setting asymmetric VF configuration per port.

11.10.3 Configuring VF Limits

The following are examples for configuring the number of VFs per port:

1. In the BlueField, issue the following commands to configure 32 VFs per port:

   ```
   bf> mlxconfig -d 03:00.0 -y s PF_NUM_OF_VF=32
   bf> mlxconfig -d 03:00.1 -y s PF_NUM_OF_VF=32
   ```

   ⚠ The BlueField ECPF driver in the BlueField's Arm OS limits the number of VFs it supports to 32 per port.

2. In the host OS, issue the following commands to configure up to 126 VFs per port:

   ```
   host> mlxconfig -d 03:00.0 -y s PF_NUM_OF_VF=126
   host> mlxconfig -d 03:00.1 -y s PF_NUM_OF_VF=126
   ```

11.10.4 Applying Configuration

To apply the configuration, perform a BlueField system reboot for the **mlxconfig** settings to take effect.

11.10.5 Instantiating VFs

Create ECPF VFs:

```
echo 1 > /sys/class/net/p0/device/sriov_numvfs
```

11.10.6 Limitations and Legacy Support

BlueField SR-IOV VFs do not support the following legacy SR-IOV functionalities:

- Virtual switch tagging (VF VLAN)
- Spoof check
- VF trust
- VF rate

11.11 fTPM over OP-TEE

This page provides an overview and configuration instructions for using fTPM over the OP-TEE on NVIDIA® BlueField®-3 DPUs (and higher) to deliver secure, hardware-isolated cryptographic services and protected storage without requiring a separate dedicated chip.

11.11.1 Introduction

> ⚠ fTMP over OP-TEE is supported on BlueField DPUs only on host OS Ubuntu 22.04 or Oracle Linux.

The Trusted Computing Group (TCG) is responsible for the specifications governing the trusted platform module (TPM). In many systems, the TPM provides integrity measurements, health checks and authentication services.

Attributes of a TPM:

- Support for bulk (symmetric) encryption in the platform
- High quality random numbers
- Cryptographic services
- Protected persistent store for small amounts of data, sticky bits, monotonic counters, and extendible registers
- Protected pseudo-persistent store for unlimited amounts of keys and data
- Extensive choice of authorization methods to access protected keys and data
- Platform identities
- Support for platform privacy
- Signing and verifying digital signatures
- Certifying the properties of keys and data
- Auditing the usage of keys and data

With TPM 2.0., the TCG creates a library specification describing all the commands or features that could be implemented and may be necessary in servers, laptops, or embedded systems. Each

platform can select the features needed and the level of security or assurance required. This flexibility allows the newest TPMs to be applied to many embedded applications.

Firmware TPM (fTPM) is implemented in protected software. The code runs on the main CPU so that a separate chip is not required. While running like any other program, the code is in a protected execution environment called a trusted execution environment (TEE) which is separate from the rest of the programs running on the CPU. By doing this, secrets (e.g., private keys perhaps needed by the TPM but should not be accessed by others) can be kept in the TEE creating a more secure environment.

> fTPM provides similar functionality to a chip-based TPM, but does not require extra hardware. It complies with the official TCG reference implementation of the TPM 2.0 specification. The source code of this implementation is located here.

> fTPM fully supports TPM2 Tools and the TCG TPM2 Software Stack (TSS).

Characteristics of an fTPM:

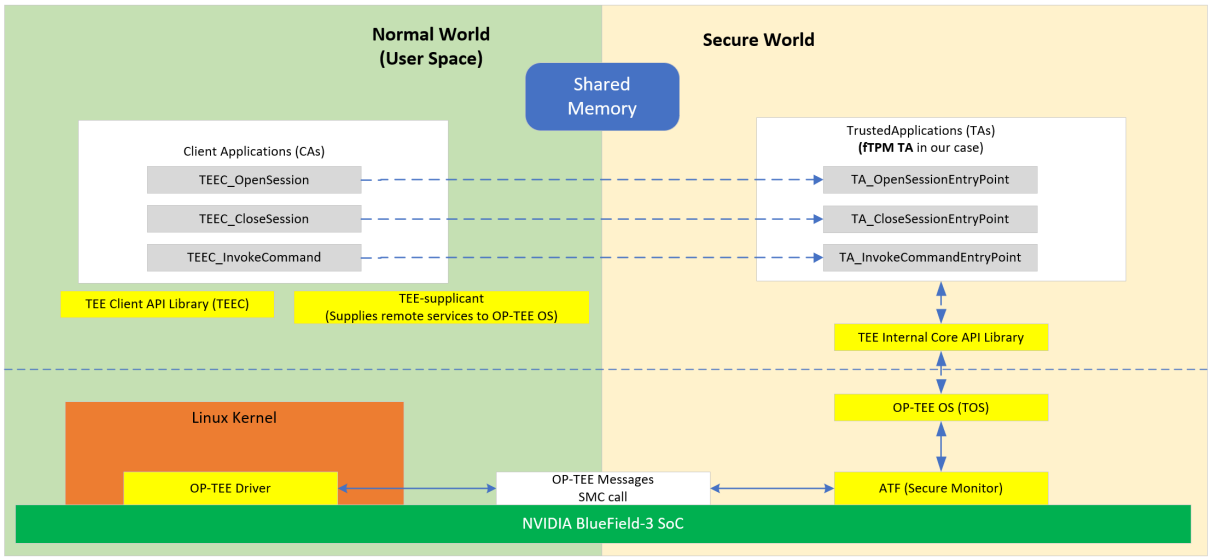
- Emulated TPM using an isolated hardware environment
- Executes in an open-source trusted execution environment (OP-TEE)
- fTPM trusted application (TA) is part of the OP-TEE binary. This allows early access on bootup, runs only in secure DRAM.

  > Currently, the only TA supported is fTPM.

- fTPM is not a task waiting to be woken up. It only executes when TPM primitives are forwarded to it from the user space. It is guaranteed shielded execution via the TEE OS and, when invoked via the TEE Dispatcher, runs to completion.

11.11.2 Architecture and Secure Storage

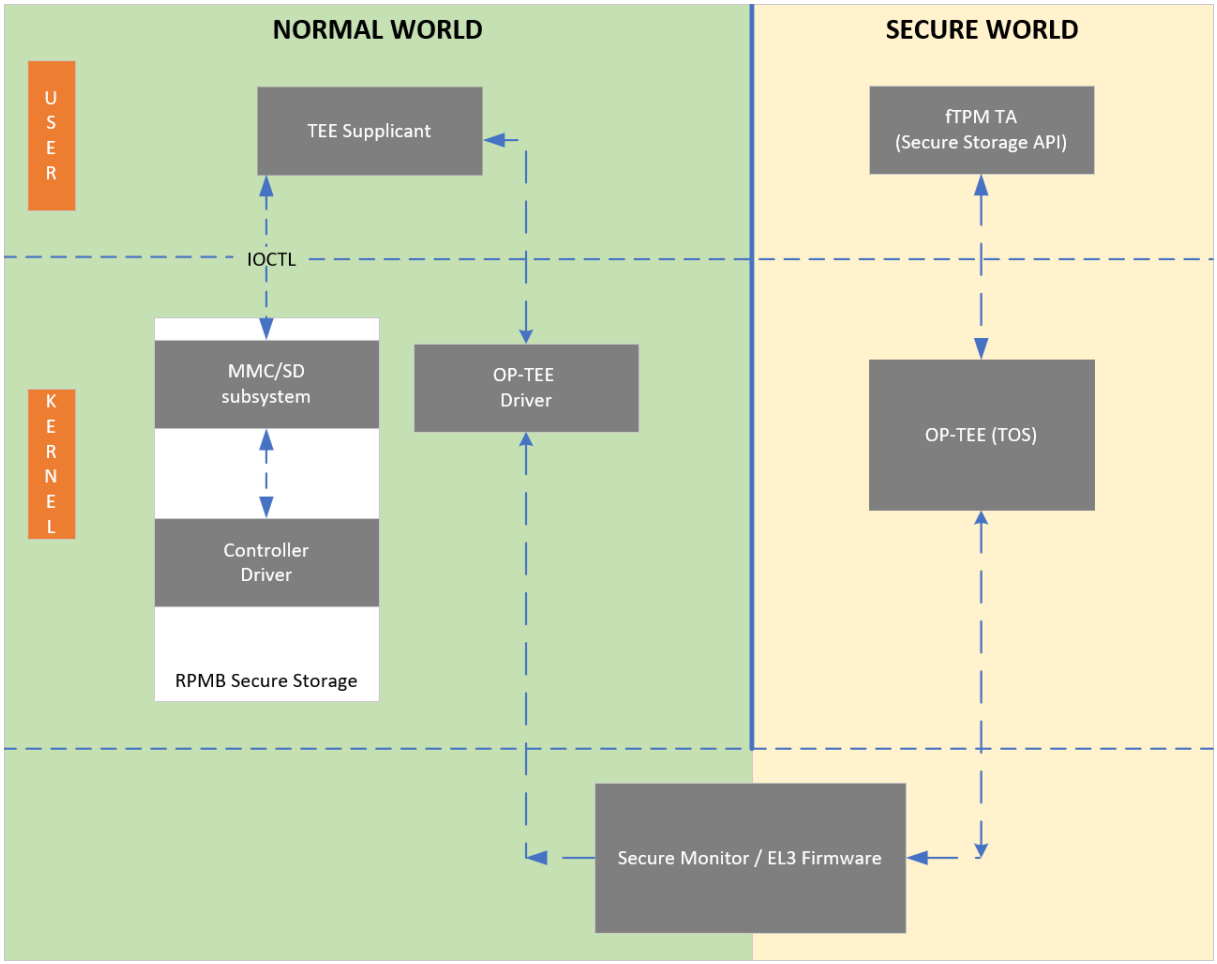
The fTPM TA is the only TA BlueField-3 currently supports. Any TA loaded by OP-TEE must be signed (signing done externally) and then authenticated by OP-TEE before being allowed to load and execute.



A replay-protected memory block (RPMB) is provided as a means for a system to store data to the specific memory area in an authenticated and replay-protected manner, making it readable and writable only after a successful authentication read/write accesses. The RPMB is a dedicated partition available on the eMMC, which makes it possible to store and retrieve data with integrity and authenticity support. A signed access to an RPMB is supported by first programming authentication key information to the eMMC memory (shared secret). The RPMB authentication key is programmed into BlueField at manufacturing time.

RPMB features a 4MB partition secure storage for BlueField-3.

There is no eMMC controller driver in OP-TEE. All device operations have to go through the normal world via the TEE-supplicant daemon, which relies on the Linux kernel's ioctl interface to access the device. All writes to the RPMB are atomic, authenticated, and encrypted. The RPMB partition stores data in an authenticated, replay-protected manner, making it a perfect complement to fTPM for storing and protecting data.



11.11.3 Enabling OP-TEE on BlueField-3

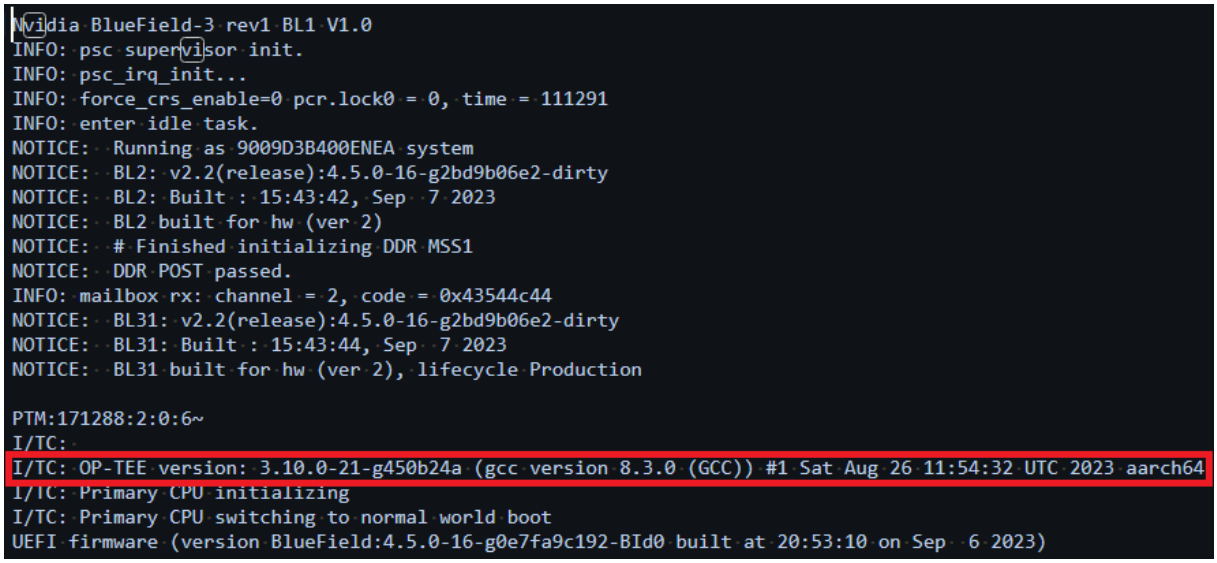
Enable OP-TEE in the UEFI menu:

1. ESC into the UEFI on BlueField boot.
2. Navigate to Device Manager > System Configuration.
3. Check "Enable OP-TEE".
4. Save the change and reset/reboot.
5. Upon reboot OP-TEE is enabled.

> ⚠ OP-TEE is essentially dormant (does not have an OS scheduler) and reacts to external inputs.

11.11.4 Verifying BlueField-3 is Running OP-TEE

Users can see the OP-TEE version during BlueField-3 boot:



The following indicators should all be present if fTPM over OP-TEE is enabled:

- Check "dmesg" for the OP-TEE driver initializing

```
root@localhost ~]# dmesg | grep tee
[    5.646578] optee: probing for conduit method.
[    5.653282] optee: revision 3.10 (450b24ac)
[    5.653991] optee: initialized driver
```

- Verify that the following kernel modules are loaded (running):

```
[root@localhost ~]# lsmod | grep tee
tpm_ftpm_tee           16384  0
optee                  49152  1
tee                    49152  3 optee,tpm_ftpm_tee
```

- Verify that the proper devices are created/available (4 in total):

```
[root@localhost ~]# ls -l /dev/tee*
crw------- 1 root root 234,  0 Sep  8 18:24 /dev/tee0
crw------- 1 root root 234, 16 Sep  8 18:24 /dev/teepriv0

[root@localhost ~]# ls -l /dev/tpm*
crw-rw---- 1 tss root  10,   224 Sep  8 18:24 /dev/tpm0
crw-rw---- 1 tss tss  252, 65536 Sep  8 18:24 /dev/tpmrm0
```

- Verify that the required processes are running (3 in total):

```
[root@localhost ~]# ps axu | grep tee
root         707  0.0  0.0  76208  1372 ?        Ssl  14:42   0:00 /usr/sbin/tee-supplicant
root         715  0.0  0.0      0     0 ?        I<   14:42   0:00 [optee_bus_scan]

[root@localhost ~]# ps axu | grep tpm
root         124  0.0  0.0      0     0 ?        I<   18:24   0:00 [tpm_dev_wq]
```

12 Archives

For documentation and software downloads of previous DOCA software releases, please refer to the "DOCA Downloads and Documentation" page.

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. Neither NVIDIA Corporation nor any of its direct or indirect subsidiaries and affiliates (collectively: “NVIDIA”) make any representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice. Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer (“Terms of Sale”). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Trademarks
NVIDIA, the NVIDIA logo, and Mellanox are trademarks and/or registered trademarks of NVIDIA Corporation and/or its affiliates in the U.S. and in other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright
© 2026 NVIDIA Corporation & affiliates. All Rights Reserved.

NVIDIA