Miscellaneous (Runtime)
# Table of contents

NVIDIA DOCA Glossary ............................................. 3  
NVIDIA DOCA Crypto Acceleration ......................... 10  
NVIDIA DOCA Services Fluent Logger ..................... 12  
NVIDIA DOCA DPU CLI ........................................ 16  
NVIDIA DOCA Emulated Devices ............................ 21  
NVIDIA BlueField Modes of Operation .................. 22  
DOCA Switching .................................................. 37  
  
  VirtIO Acceleration through Hardware vDPA .............. 39  
  Bridge Offload ................................................ 45  
  Link Aggregation ............................................. 47  
  Controlling Host PF and VF Parameters ................ 57  
NVIDIA DOCA with OpenSSL .................................. 60  
NVIDIA BlueField DPU Scalable Function User Guide ... 65  
NVIDIA TLS Offload Guide ..................................... 78  
NVIDIA DOCA Troubleshooting Guide ....................... 102  
  
  NVIDIA BlueField Reset and Reboot Procedures ......... 136  
NVIDIA DOCA Virtual Functions User Guide ............. 146  

Miscellaneous (Runtime) .......................................... 1
This section contains the following pages:

- NVIDIA DOCA Glossary
- NVIDIA DOCA Crypto Acceleration
- NVIDIA DOCA Services Fluent Logger
- NVIDIA DOCA DPU CLI
- NVIDIA DOCA Emulated Devices
- NVIDIA BlueField Modes of Operation
- DOCA Switching
- NVIDIA DOCA with OpenSSL
- NVIDIA BlueField DPU Scalable Function User Guide
- NVIDIA TLS Offload Guide
- NVIDIA DOCA Troubleshooting Guide
- NVIDIA DOCA Virtual Functions User Guide
## NVIDIA DOCA Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACS</td>
<td>Access control services</td>
</tr>
<tr>
<td>ASAP²</td>
<td>Accelerated Switching and Packet Processing</td>
</tr>
<tr>
<td>ASN</td>
<td>Autonomous system number</td>
</tr>
<tr>
<td>ATF</td>
<td>Arm-trusted firmware</td>
</tr>
<tr>
<td>BAR</td>
<td>Base address register</td>
</tr>
<tr>
<td>BDF address</td>
<td>Bus, device, function address. This is the device's PCIe bus address to uniquely identify the specific device.</td>
</tr>
<tr>
<td>BFB</td>
<td>BlueField bootstream</td>
</tr>
<tr>
<td>BGP</td>
<td>Border gateway protocol</td>
</tr>
<tr>
<td>BMC</td>
<td>Board management controller</td>
</tr>
<tr>
<td>BUF</td>
<td>Buffer</td>
</tr>
<tr>
<td>BSP</td>
<td>BlueField support package</td>
</tr>
<tr>
<td>CBS</td>
<td>Committed burst size</td>
</tr>
<tr>
<td>CIR</td>
<td>Committed information rate</td>
</tr>
<tr>
<td>CMDQ</td>
<td>Command queue</td>
</tr>
<tr>
<td>CPDS</td>
<td>Control pipe dynamic size</td>
</tr>
<tr>
<td>CQE</td>
<td>Completion queue events</td>
</tr>
<tr>
<td>CTX</td>
<td>Context</td>
</tr>
<tr>
<td>DEK</td>
<td>Data encryption key</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct memory access</td>
</tr>
<tr>
<td>DOCA</td>
<td>DPU SDK</td>
</tr>
<tr>
<td>DPA</td>
<td>Data path accelerator; an auxiliary processor designed to accelerate datapath operations</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>DPCP</td>
<td>Direct packet control plane</td>
</tr>
<tr>
<td>DPDK</td>
<td>Data plane development kit</td>
</tr>
<tr>
<td>DPI</td>
<td>Deep packet inspection</td>
</tr>
<tr>
<td>DPIF</td>
<td>Datapath offload interface</td>
</tr>
<tr>
<td>DPU</td>
<td>Data processing unit, the third pillar of the data center with CPU and GPU. BlueField is available as a DPU and as a SuperNIC.</td>
</tr>
<tr>
<td>DW</td>
<td>Dword</td>
</tr>
<tr>
<td>EBS</td>
<td>Excess burst size</td>
</tr>
<tr>
<td>ECE</td>
<td>Enhanced connection establishment</td>
</tr>
<tr>
<td>ECPF</td>
<td>Embedded CPU physical function</td>
</tr>
<tr>
<td>EIR</td>
<td>Excess information rate</td>
</tr>
<tr>
<td>EM</td>
<td>Exact match</td>
</tr>
<tr>
<td>eMMC</td>
<td>Embedded multi-media card</td>
</tr>
<tr>
<td>ESP</td>
<td>EFI system partition</td>
</tr>
<tr>
<td>ESP</td>
<td>Encapsulating security payload</td>
</tr>
<tr>
<td>EU</td>
<td>Execution unit. HW thread; a logical DPA processing unit.</td>
</tr>
<tr>
<td>FLR</td>
<td>Function level reset</td>
</tr>
<tr>
<td>FIFO</td>
<td>First-in-first-out</td>
</tr>
<tr>
<td>FIPS</td>
<td>Federal Information Processing Standards</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-programmable gate arrays</td>
</tr>
<tr>
<td>FW</td>
<td>Firmware</td>
</tr>
<tr>
<td>GDAKIN</td>
<td>GPUDirect async kernel-initiated network</td>
</tr>
<tr>
<td>GDB</td>
<td>GNU debugger</td>
</tr>
<tr>
<td>HCA</td>
<td>Host-channel adapter</td>
</tr>
<tr>
<td>Host</td>
<td>When referring to “the host” this documentation is referring to the <strong>server host</strong>. When referring to the Arm based host, the documentation will specifically call out &quot;Arm host&quot;.</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Server host OS refers to the Host Server OS (Linux or Windows)</td>
<td></td>
</tr>
<tr>
<td>Arm host refers to the AARCH64 Linux OS which is running on the BlueField Arm Cores</td>
<td></td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
</tr>
<tr>
<td>hwmon</td>
<td>Hardware monitoring</td>
</tr>
<tr>
<td>IB</td>
<td>InfiniBand</td>
</tr>
<tr>
<td>ICM</td>
<td>Interface configuration memory</td>
</tr>
<tr>
<td>ICV</td>
<td>Integrity check value</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated development environment</td>
</tr>
<tr>
<td>IKE</td>
<td>Internet key exchange</td>
</tr>
<tr>
<td>IR</td>
<td>Intermediate representation</td>
</tr>
<tr>
<td>IRQ</td>
<td>Interrupt request</td>
</tr>
<tr>
<td>KPI</td>
<td>Key performance indicator</td>
</tr>
<tr>
<td>LRO</td>
<td>Large receive offload</td>
</tr>
<tr>
<td>LSO</td>
<td>Large send offload</td>
</tr>
<tr>
<td>LTO</td>
<td>Link-time optimization</td>
</tr>
<tr>
<td>MFT</td>
<td>Mellanox firmware tools</td>
</tr>
<tr>
<td>MLNX_OFED</td>
<td>Mellanox OpenFabrics Enterprise Distribution</td>
</tr>
<tr>
<td>MPU</td>
<td>Message passing interface</td>
</tr>
<tr>
<td>MSB</td>
<td>Most significant bit</td>
</tr>
<tr>
<td>MSI-X</td>
<td>Message signaled interrupts extended</td>
</tr>
<tr>
<td>MSS</td>
<td>Maximum segment size</td>
</tr>
<tr>
<td>MSS</td>
<td>Memory subsystem</td>
</tr>
<tr>
<td>MST</td>
<td>Mellanox software tools</td>
</tr>
<tr>
<td>MTU</td>
<td>Maximum transmission unit</td>
</tr>
<tr>
<td>NAT</td>
<td>Network address translation</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td>NIC</td>
<td>Network interface card</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>NP</td>
<td>Notification point</td>
</tr>
<tr>
<td>NS</td>
<td>Namespace</td>
</tr>
<tr>
<td>NUMA</td>
<td>Non-uniform memory access</td>
</tr>
<tr>
<td>OOB</td>
<td>Out-of-band</td>
</tr>
<tr>
<td>OS</td>
<td>Operating system</td>
</tr>
<tr>
<td>OVS</td>
<td>Open vSwitch</td>
</tr>
<tr>
<td>PBA</td>
<td>Pending bit array</td>
</tr>
<tr>
<td>PBS</td>
<td>Peak burst size</td>
</tr>
<tr>
<td>PCIe</td>
<td>PCI Express; Peripheral Component Interconnect Express</td>
</tr>
<tr>
<td>PF</td>
<td>Physical function</td>
</tr>
<tr>
<td>PE</td>
<td>Progress engine</td>
</tr>
<tr>
<td>PHC</td>
<td>Physical hardware clock</td>
</tr>
<tr>
<td>PIR</td>
<td>Peak information rate</td>
</tr>
<tr>
<td>PK</td>
<td>Platform key</td>
</tr>
<tr>
<td>PKA</td>
<td>Public key accelerator</td>
</tr>
<tr>
<td>POC</td>
<td>Proof of concept</td>
</tr>
<tr>
<td>PUD</td>
<td>Process under debug</td>
</tr>
<tr>
<td>RD</td>
<td>Route distinguisher</td>
</tr>
<tr>
<td>RDMA</td>
<td>Remote direct memory access</td>
</tr>
<tr>
<td>RDMA CM</td>
<td>RDMA connection manager</td>
</tr>
<tr>
<td>RegEx</td>
<td>Regular expression</td>
</tr>
<tr>
<td>REQ</td>
<td>Request</td>
</tr>
<tr>
<td>RES</td>
<td>Response</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| RN   | Request node  
RN-F – Fully coherent request node  
RN-D – IO coherent request node with DVM support  
RN-I – IO coherent request node |
<p>| RNG  | Random number generator/generation |
| RoCE | Ethernet and RDMA over converged Ethernet |
| RP   | Reaction point |
| RQ   | Receive queue |
| RShim| Random shim |
| RSP  | Remote serial protocol |
| RT   | Route target |
| RTOS | Real-time operating system |
| RTT  | Round-trip time |
| RX   | Receive |
| RXP  | Regular expression processor |
| SA   | Security association |
| SBSA | Server base system architecture |
| SDK  | Software development kit |
| SF   | Sub-function or scalable function |
| SFC  | Services function chaining |
| SG   | Scatter-gather |
| SHA  | Secure hash algorithm |
| SN   | Sequence number |
| SNAP | Storage-defined network-accelerated processing |
| SPDK | Storage performance development kit |
| SPI  | Security parameters index |
| SQ   | Send queue |</p>
<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR-IOV</td>
<td>Single-root IO virtualization</td>
</tr>
<tr>
<td>SuperNIC</td>
<td>a configuration of a DPU that is specific for E-W networking. BlueField has a SuperNIC configuration</td>
</tr>
<tr>
<td>SVI</td>
<td>Switch virtual interface</td>
</tr>
<tr>
<td>Sync event</td>
<td>Synchronization event</td>
</tr>
<tr>
<td>TAI</td>
<td>International Atomic Time</td>
</tr>
<tr>
<td>TIR</td>
<td>Transport interface receive</td>
</tr>
<tr>
<td>TIS</td>
<td>Transport interface send</td>
</tr>
<tr>
<td>TLS</td>
<td>Transport layer security</td>
</tr>
<tr>
<td>TSO</td>
<td>TCP segmentation offload</td>
</tr>
<tr>
<td>TX</td>
<td>Transmit</td>
</tr>
<tr>
<td>UDS</td>
<td>Unix domain socket</td>
</tr>
<tr>
<td>UEFI</td>
<td>Unified extensible firmware interface</td>
</tr>
<tr>
<td>UTC</td>
<td>Coordinated Universal Time</td>
</tr>
<tr>
<td>vDPA</td>
<td>Virtual data path acceleration</td>
</tr>
<tr>
<td>VF</td>
<td>Virtual function</td>
</tr>
<tr>
<td>VFE</td>
<td>Virtio full emulation</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual machine</td>
</tr>
<tr>
<td>VMA</td>
<td>NVIDIA® Messaging Accelerator</td>
</tr>
<tr>
<td>VNI</td>
<td>• Virtual network identifier</td>
</tr>
<tr>
<td></td>
<td>• VXLAN network identifier</td>
</tr>
<tr>
<td>VPI</td>
<td>Virtual protocol interconnect</td>
</tr>
<tr>
<td>VRF</td>
<td>Virtual routing and forwarding</td>
</tr>
<tr>
<td>VTEP</td>
<td>VXLAN tunnel endpoint</td>
</tr>
<tr>
<td>WorkQ</td>
<td>Work queue</td>
</tr>
<tr>
<td>WorkQ or</td>
<td>Work queue</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>workq</td>
<td></td>
</tr>
<tr>
<td>WQE</td>
<td>Work queue elements</td>
</tr>
<tr>
<td>WR</td>
<td>Write</td>
</tr>
<tr>
<td>XLIO</td>
<td>NVIDIA® Accelerated IO</td>
</tr>
</tbody>
</table>
NVIDIA DOCA Crypto Acceleration

NVIDIA® BlueField® DPU incorporates several Public Key Acceleration (PKA) engines to offload the processor of the Arm host, providing high-performance computation of PK algorithms. BlueField's PKA is useful for a wide range of security applications. It can assist with SSL acceleration, or a secure high-performance PK signature generator/checker and certificate related operations.

BlueField's PKA software libraries implement a simple, complete framework for crypto public key infrastructure (PKI) acceleration. It provides direct access to hardware resources from the user space, and makes available a number of arithmetic operations—some basic (e.g., addition and multiplication), and some complex (e.g., modular exponentiation and modular inversion)—and high-level operations such as RSA, Diffie-Hallman, Elliptic Curve Cryptography, and the Federal Digital Signature Algorithm (DSA as documented in FIPS-186) public-private key systems.

Some of the use cases for the BlueField PKA involve integrating OpenSSL software applications with BlueField's PKA hardware. The BlueField PKA dynamic engine for OpenSSL allows applications integrated with OpenSSL (e.g., StrongSwan) to accomplish a variety of security-related goals and to accelerate the cryptographic processing with the BlueField PKA hardware. OpenSSL versions ≥1.0.0, ≤1.1.1, and 3.0.2 are supported.

Note

With CentOS 7.6, only OpenSSL 1.1 (not 1.0) works with PKA engine and keygen. Use openssl11 with PKA engine and keygen.

The engine supports the following operations:

- RSA
- DH
- DSA
- ECDSA
- ECDH

- Random number generation that is cryptographically secure.

Up to 4096-bit keys for RSA, DH, and DSA operations are supported. Elliptic Curve Cryptography support of (nist) prime curves for 160, 192, 224, 256, 384 and 521 bits.

For example:

To sign a file using BlueField's PKA engine:

```
$ openssl dgst -engine pka -sha256 -sign <privatekey> -out <signature> <filename>
```

To verify the signature, execute:

```
$ openssl dgst -engine pka -sha256 -verify <publickey> -signature <signature> <filename>
```

For further details on BlueField PKA, please refer to "PKA Driver Design and Implementation Architecture Document" and/or "PKA Programming Guide". Directions and instructions on how to integrate the BlueField PKA software libraries are provided in the README files on our [PKA GitHub](https://github.com/bluefield).
NVIDIA DOCA Services
Fluent Logger

This guide provides instructions on how to use the logging infrastructure for DOCA services on top of NVIDIA® BlueField® DPU.

Introduction

Fluent Bit is a fast log collector that collects information from multiple sources and then forwards the data onward using Fluent.

On NVIDIA DPUs, the Fluent Bit logger can be easily configured to collect system data and the logs from the different DOCA services.

Deployment

The deployment is based on a recommended configuration template for the existing Fluent Bit container.

For information about the deployment of DOCA containers on top of the BlueField DPU, refer to NVIDIA DOCA Container Deployment Guide.

The following is an example YAML file for deploying the Fluent Bit pod:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: fluent-bit
spec:
  hostNetwork: true
  containers:
    - name: fluent-bit
      image: fluent/fluent-bit:latest
      imagePullPolicy: Always
```
As explained in the "Configuration" section, Fluent Bit uses a configuration file. As such, to ensure that the example YAML file is shared from the DPU to the deployed Fluent Bit container, use the following:

```yaml
# Example resource definitions
resources:
  requests:
    memory: "100Mi"
    cpu: "200m"
  limits:
    memory: "200Mi"
    cpu: "300m"
volumes:
  - name: varlog
    mountPath: /var/log
  - name: config-file
    mountPath: /fluent-bit/etc/fluent-bit.conf
volumes:
  - name: varlog
    hostPath:
      path: /var/log
  - name: config-file
    hostPath:
      path: /opt/mellanox/doca/services/fluent-bit.conf
type: File
```

Note

The path below is just an example for where the user can place the fluent-bit.conf file. The file could be placed in a different directory on the DPU as long as the YAML file points to the updated location.
The Fluent Bit configuration file should have the following sections:

- **[SERVICE]** – to define the service specifications
- **[INPUT]** – to define folders to collect logs from (there could be multiple inputs)
- **[OUTPUT]** – IP and port to stream the data to

Example configuration file:

```
[SERVICE]
  Flush 2
  Log_Level info
  Daemon off
  Parsers_File parsers.conf
  HTTP_Server On
  HTTP_Listen 0.0.0.0
  HTTP_Port 2020

[INPUT]
  Name tail
  Tag kube.*
  Path /var/log/containers/*/*.log
  Parser docker
  Mem_Buf_Limit 5MB
  Skip_Long_Lines On
  Refresh_Interval 10

[INPUT]
  Name tail
  Tag sys.*
  Path /var/log/doca/*/*.log
  Mem_Buf_Limit 5MB
  Skip_Long_Lines On
  Refresh_Interval 10

[OUTPUT]
  Name es
  Match *
  Host 10.20.30.40
  Port 9201
  Index fluent_bit
```
More information about the full specifications can be found in the official Fluent Bit manual.

**Troubleshooting**

For container-related troubleshooting, refer to the "Troubleshooting" section in the NVIDIA DOCA Container Deployment Guide.

For general troubleshooting, refer to the NVIDIA DOCA Troubleshooting Guide.

When copying the above YAML file, it is possible that the container infrastructure logs give an error related to RFC 1123". These errors are usually a result of a spacing error in the file, which sometimes occur when copying the file as is from this page. To fix this issue, make sure that only the space character (' ') is used as a spacer in the file and not other whitespace characters that might have been added during the copy operation.
NVIDIA DOCA DPU CLI

This guide provides quick access to a useful set of CLI commands and utilities on the NVIDIA® BlueField® DPU environment.

Introduction

This guide provides a concise guide on useful commands for DOCA deployment and configuration.

The tables in this guide provide two categories of commands:

- General commands for Linux/networking environment
- DOCA/DPU-specific commands

Note

For more information about these commands, such as usage instructions, flag options, arguments and so on, use the -h option after the command or use the manual (e.g., man lspci).

General Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifconfig</td>
<td>Used to configure kernel-resident network interfaces. It is used at boot time to set up interfaces as necessary. After that, it is usually only needed when debugging or when system tuning is needed. If no arguments are given, ifconfig displays the status of the currently active interfaces. If a single interface argument is given, it displays the status of the</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ethtool</td>
<td>Used to query and control network device driver and hardware settings, particularly for wired Ethernet devices. &lt;devname&gt; is the name of the network device on which ethtool should operate.</td>
</tr>
</tbody>
</table>
|            | **Note**  
This command shows the speed of the network card of the DPU.                                                                                                                                                 |
| lspci      | Displays information about PCIe buses in the system and devices connected to them. By default, it shows a brief list of devices.                                                                                 |
| tcpdump    | Dump traffic on a network. Usage: tcpdump -i <interface> where <interface> is any port interface (physical/SF rep/VF port rep).                                                                                  |
| ovs-vsctl  | Utility for querying and configuring ovs-vswitchd. The ovs-vsctl program supports the model of a bridge implemented by Open vSwitch in which a single bridge supports ports on multiple VLANs. |
| mount      | Used for mounting a work directory on the DPU.                                                                                                                                                                 |
|            | **Note**  
Must be used after creating a new directory named myshare under root (i.e., mkdir /myshare)                                                                                                                |
| scp        | Secure copy (remote file copy program). Useful for copying files from BlueField to the host and vice versa.                                                                                                     |
| iperf      | Used for server-client connection. Useful to check if the network connection achieves the speed of the network card on the DPU (line rate).                                                            |
# DPU/DOCA Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibdev2netdev</td>
<td>Displays available mlnx interfaces</td>
</tr>
<tr>
<td>mst</td>
<td>Used to start MST service, to stop it, and for other operations with NVIDIA devices like reset and enabling remote access</td>
</tr>
<tr>
<td>cat /etc/mlnx-release</td>
<td>Displays the full BlueField image (bfb) version</td>
</tr>
<tr>
<td>cat /etc/os-release</td>
<td>Displays the details of the underlying OS installed on BlueField</td>
</tr>
<tr>
<td>ibv_devinfo</td>
<td>Displays the current InfiniBand connected devices and relevant information. Useful for checking current firmware version.</td>
</tr>
<tr>
<td>ipmitool power cycle</td>
<td>Power cycle</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong> Prior to performing a power cycle, make sure to do a graceful shutdown.</td>
</tr>
<tr>
<td>echo 1024 &gt;</td>
<td>/sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages</td>
</tr>
<tr>
<td>mlxdevm tool</td>
<td>The mlxdevm tool is found under /opt/mellanox/iproute2/sbin/. With this tool it is possible to create an SF and set its state to active, configure a HW address and set it to trusted, deploy the created SF and print info about it.</td>
</tr>
<tr>
<td>/opt/mellanox/iproute2/sbin/mlxdevm port add pci/&lt;pci_address&gt;flavour pcisf pfnum &lt;corresponding_physical_function_number&gt; sfnunm &lt;unique_sf_number&gt;</td>
<td>Creates an SF in the flavor of the given PF with the given unique SF number. Example:</td>
</tr>
<tr>
<td>/opt/mellanox/iproute2/sbin/mlxdevm port show</td>
<td>Displays information about the available SFs</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| /opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/<sf_index> hw_addr <HW_address> trust on state active | Configures SF capabilities such as setting the HW address, making it "trusted", and setting its state to active. `<sf_index>` the SF. To obtain this index, you may run mlxdevm port show. Example:
```
/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/229377 hw_addr 02:25:f2:8d:a2:4c trust on state active
```
| $ echo mlx5_core.sf. <next_serial> > /sys/bus/auxiliary/drivers/mlx5_core.sfCfg/unbind $ echo mlx5_core.sf. <next_serial> > /sys/bus/auxiliary/drivers/mlx5_core.sf/bind | These two commands deploy the created SF. The first command unbinds the SF from the default driver, while the second command binds the SF to the actual driver. The deployment phase should be done after the capabilities of the SF are configured. The SF is identified by `<next_serial>` which can be obtained by running the command below.
```
ls /sys/bus/auxiliary/devices/mlx5_core.sf.*
```
| Displays additional information about the created SFs and their "next serial numbers". For example, if mlx5_core.sf.2 exists in the output of the command, then running `cat /sys/bus/auxiliary/devices/mlx5_core.sf.2/sfnum` would output the sfnum related to mlx5_core.sf.2. | These two commands must be executed to delete a given SF. First, users must set the state of the SF to inactive, and only then should it be deleted.
```
/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/<pci_address>/<sf_index> state inactive /opt/mellanox/iproute2/sbin/mlxdevm port del pci/<pci_address>/<sf_index>
```
<p>| /opt/mellanox/iproute2/sbin/mlxdevm port help | Displays additional information about operations that can be used on created SF ports |
| /opt/mellanox/iproute2/sbin/mlxdevm port help | Displays currently active K8S pods, and their IDs (it might take up to 20-30 seconds for the pod to start) |
| /opt/mellanox/iproute2/sbin/mlxdevm port help | Displays currently active containers and their IDs |
| /opt/mellanox/iproute2/sbin/mlxdevm port help | Displays all containers, including containers that recently finished their execution |
| /opt/mellanox/iproute2/sbin/mlxdevm port help | Examines the logs of a given container |</p>
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id&gt;</td>
<td></td>
</tr>
<tr>
<td>crictl exec -it &lt;container-id&gt; /bin/bash</td>
<td>Attaches a shell to a running container</td>
</tr>
<tr>
<td>journalctl -u kubelet</td>
<td>Examines the Kubelet logs. Useful when a pod/container fails to spawn.</td>
</tr>
<tr>
<td>crictl stopp &lt;pod-id&gt;</td>
<td>Stops a running K8S pod</td>
</tr>
<tr>
<td>crictl stop &lt;container-id&gt;</td>
<td>Stops a running container</td>
</tr>
<tr>
<td>crictl rmi &lt;image-id&gt;</td>
<td>Removes a container image from the local K8S registry</td>
</tr>
</tbody>
</table>
NVIDIA DOCA Emulated Devices

For information on virtio-net emulation, please refer to NVIDIA BlueField Virtio-net documentation.
NVIDIA BlueField Modes of Operation

This document describes the modes of operation available for NVIDIA® BlueField® networking platforms (DPUs or SuperNICs).

Introduction

Content:

The NVIDIA® BlueField® networking platform (DPU or SuperNIC) has several modes of operation:

- **DPU mode**, or embedded function (ECPF) ownership, where the embedded Arm system controls the NIC resources and data path

- **Zero-trust mode** which is an extension of the ECPF ownership with additional restrictions on the host side

- **NIC mode** where BlueField behaves exactly like an adapter card from the perspective of the external host

**Note**

The default mode of operation for the BlueField DPU is DPU mode

The default mode of operation for the BlueField SuperNIC is NIC mode

DPU Mode
This mode, known also as embedded CPU function ownership (ECPF) mode, is the default mode for the BlueField DPU.

In DPU mode, the NIC resources and functionality are owned and controlled by the embedded Arm subsystem. All network communication to the host flows through a virtual switch control plane hosted on the Arm cores, and only then proceeds to the host. While working in this mode, the BlueField is the trusted function managed by the data center and host administrator—to load network drivers, reset an interface, bring an interface up and down, update the firmware, and change the mode of operation on BlueField.

A network function is still exposed to the host, but it has limited privileges. In particular:

1. The driver on the host side can only be loaded after the driver on the BlueField has loaded and completed NIC configuration.

2. All ICM (Interface Configuration Memory) is allocated by the ECPF and resides in the BlueField's memory.

3. The ECPF controls and configures the NIC embedded switch which means that traffic to and from the host (BlueField) interface always lands on the Arm side.

When the server and BlueField are initiated, the networking to the host is blocked until the virtual switch on the BlueField is loaded. Once it is loaded, traffic to the host is allowed by default.

There are two ways to pass traffic to the host interface: Either using representors to forward traffic to the host (every packet to/from the host would be handled also by the network interface on the embedded Arm side) or push rules to the embedded switch which allows and offloads this traffic.
In DPU mode, OpenSM must be run from the BlueField side (not the host side). Also, management tools (e.g., sminfo, ibdev2netdev, ibnetdiscover) can only be run from the BlueField side (not from the host side).

**Zero-trust Mode**

Zero-trust mode is a specialization of DPU mode which implements an additional layer of security where the host system administrator is prevented from accessing BlueField from the host. Once zero-trust mode is enabled, the data center administrator should control BlueField entirely through the Arm cores and/or BMC connection instead of through the host.

For security and isolation purposes, it is possible to restrict the host from performing operations that can compromise the BlueField. The following operations can be restricted individually when changing the BlueField host to zero-trust mode:

- Port ownership – the host cannot assign itself as port owner
- Hardware counters – the host does not have access to hardware counters
- Tracer functionality is blocked
- RShim interface is blocked
- Firmware flash is restricted

**Enabling Zero-trust Mode**

To enable host restriction:

1. Start the MST service.
2. Set zero-trust mode. From the Arm side, run:

   ```
   $ sudo mlxprivhost -d /dev/mst/<device> r --disable_rshim --disable_tracer --disable_counter_rd -
   --disable_port_owner
   ```

   - If no --disable_* flags are used, users must perform BlueField system reboot as explained in the "NVIDIA BlueField Reset and Reboot Procedures"
Disabling Zero-trust Mode

To disable host restriction:

1. Set the mode to privileged. Run:

   ```bash
   $ sudo mlxprivhost -d /dev/mst/<device> p
   ```

2. Applying configuration:

   - If host restriction had not been applied using any `--disable_*` flags, users must perform BlueField system reboot as explained in the "NVIDIA BlueField Reset and Reboot Procedures" troubleshooting page.

   - If host restriction had been applied using any `--disable_*` flags, users must perform BlueField system-level reset as explained in the "NVIDIA BlueField Reset and Reboot Procedures" troubleshooting page.

NIC Mode

In this mode, BlueField behaves exactly like an adapter card from the perspective of the external host.

ℹ️ Note
The following instructions presume BlueField to be operating in DPU mode. If BlueField is operating in zero-trust mode, please return to DPU mode before proceeding.

Note

The following notes are relevant for updating the BFB bundle in NIC mode:

- During BFB Bundle installation, Linux is expected to boot to upgrade NIC firmware and BMC software

- During the BFB Bundle installation, it is expected for the mlx5 driver to error messages on the x86 host. These prints may be ignored as they are resolved by a mandatory, post-installation power cycle.

- It is mandatory to power cycle the host after the installation is complete for the changes to take effect

- As Linux is booting during BFB Bundle installation, it is expected for the mlx5 core driver to timeout on the BlueField Arm

NIC Mode for BlueField-3

Note

When BlueField-3 is configured to operate in NIC mode, Arm OS will not boot.
NIC mode for BlueField-3 saves power, improves device performance, and improves the host memory footprint.

**Configuring NIC Mode on BlueField-3 from Linux**

**Enabling NIC Mode on BlueField-3 from Linux**

Before moving to NIC mode, make sure you are operating in DPU mode by running:

```bash
host/bf> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 -e q
```

The output should have `INTERNAL_CPU_MODEL=EMBEDDED_CPU(1)` and `EXP_ROM_UEFI_ARM_ENABLE = True (1)` (default).

To enable NIC mode from DPU mode:

1. Run the following on the host or Arm:

```bash
host/bf> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s INTERNAL_CPU_OFFLOAD_ENGINE=1
```

2. Perform a BlueField system-level reset, for the `mlxconfig` settings to take effect. Refer to the "NVIDIA BlueField Reset and Reboot Procedures" troubleshooting page for instructions.

**Disabling NIC Mode on BlueField-3 from Linux**

To return to DPU mode from NIC mode:

1. Run the following on the host:

```bash
host> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s INTERNAL_CPU_OFFLOAD_ENGINE=0
```
2. Perform a BlueField system-level reset for the mlxconfig settings to take effect. Refer to the "NVIDIA BlueField Reset and Reboot Procedures" troubleshooting page for instructions.

Configuring NIC Mode on BlueField-3 from Host BIOS HII UEFI Menu

1. Select the network device that presents the uplink (i.e., select the device with the uplink MAC address).

2. Select "BlueField Internal Cpu Configuration".

![Main Configuration Page](image-url)
To enable NIC mode, set "Internal Cpu Offload Engine" to "Disabled".

To switch back to DPU mode, set "Internal Cpu Offload Engine" to "Enabled".

---

**Configuring NIC Mode on BlueField-3 from Arm UEFI**

1. Access the Arm UEFI menu by pressing the Esc button twice.

2. Select "Device Manager".

3. Select "System Configuration".

4. Select "BlueField Modes".

5. Set the "NIC Mode" field to NicMode to enable NIC mode.
6. Exit "BlueField Modes" and "System Configuration" and make sure to save the settings. Exit the UEFI setup using the 'reset' option. The configuration is not yet applied and BlueField is expected to boot regularly, still in DPU mode.

7. Perform a BlueField system-level reset, to change to NIC Mode. Refer to the "NVIDIA BlueField Reset and Reboot Procedures" troubleshooting page for instructions.

**Configuring NIC Mode on BlueField-3 Using Redfish**

Run the following from the BlueField BMC:

1. Get the current BIOS attributes:

   ```bash
   ```

2. Change BlueField mode from **DpuMode** to **NicMode**:

   ```bash
   curl -k -u root: '<password>' -H 'content-type: application/json' -d '{ "Attributes": { "NicMode":
   ```

   ```json
   Info
   Configuring Unavailable is inapplicable.
   ```
3. Verify that the BMC has registered the new settings:

```
curl -k -u root:’<password>’ -H 'content-type: application/json' -d '{
  "Attributes": {
    "NicMode": "DpuMode" } }
'-X PATCH https://<bmc_ip>/redfish/v1/Systems/Bluefield/Oem/Nvidia
```

4. Issue a software reset then power cycle the host for the change to take effect.

5. Verify the mode is changed:

```
curl -k -u root:’<password>’ -H 'content-type: application/json' -X GET
https://<bmc_ip>/redfish/v1/Systems/Bluefield/Oem/Nvidia
```

### Note

To retrieve the mode via BIOS attributes, another BlueField software reset is required before running the command:

```
curl -k -u root:’<password>’ -H 'content-type: application/json' -X
```
Updating Firmware Components in BlueField-3 NIC Mode

Once in NIC mode, updating ATF and UFEI can be done using the standard *.bfb image:

```
# bfb-install --bfb <BlueField-BSP>.bfb --rshim rshim0
```

NIC Mode for BlueField-2

In this mode, the ECPFs on the Arm side are not functional but the user is still able to access the Arm system and update mlxconfig options.

Note

When NIC mode is enabled, the drivers and services on the Arm are no longer functional.

Configuring NIC Mode on BlueField-2 from Linux

Enabling NIC Mode on BlueField-2 from Linux

To enable NIC mode from DPU mode:

1. Run the following from the x86 host side:

```
$ mst start
$ mlxconfig -d /dev/mst/<device> s
```
INTERNAL_CPU_PAGE SUPPLIER=1 \ 
INTERNAL_CPU_ESWITCH_MANAGER=1 \ 
INTERNAL_CPU_IB_VPORT0=1 \ 
INTERNAL_CPU_OFFLOAD ENGINE=1

**Note**

To restrict RShim PF (optional), make sure to configure INTERNAL_CPU_RSHIM=1 as part of the mlxconfig command.

2. Perform BlueField system-level reset to load the new configuration.

**Info**

Refer to the troubleshooting section of the guide for a step-by-step procedure.

**Note**

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

**Note**

To obtain firmware BINs for BlueField-2 devices, please refer to the [BlueField-2 firmware download page](#).
Disabling NIC Mode on BlueField-2 from Linux

To change from NIC mode back to DPU mode:

1. Install and start the RShim driver on the host.

2. Disable NIC mode. Run:

```
$ mst start
$ mlxconfig -d /dev/mst/<device> s \n  INTERNAL_CPU_PAGE_SUPPLIER=0 \n  INTERNAL_CPU_ESWITCH_MANAGER=0 \n  INTERNAL_CPU_IB_VPORT0=0 \n  INTERNAL_CPU_OFFLOAD_ENGINE=0
```

Note

If INTERNAL_CPU_RSHIM=1, then make sure to configure
INTERNAL_CPU_RSHIM=0 as part of the mlxconfig command.

3. Perform a BlueField system reboot for the mlxconfig settings to take effect. Refer to the "NVIDIA BlueField Reset and Reboot Procedures" troubleshooting page for instructions.

Configuring NIC Mode on BlueField-2 from Arm UEFI

Follow the same instructions in section "Configuring NIC Mode on BlueField-3 from Arm UEFI".
Configuring NIC Mode on BlueField-2 Using Redfish

Follow the same instructions in section "Configuring NIC Mode on BlueField-3 Using Redfish".

Separated Host Mode (Obsolete)

⚠️ Warning

This BlueField mode of operation is obsolete. Please do not use it!

In separated host mode, a network function is assigned to both the Arm cores and the host cores. The ports/functions are symmetric in the sense that traffic is sent to both physical functions simultaneously. Each one of those functions has its own MAC address, which allows one to communicate with the other, and can send and receive Ethernet and RDMA over Converged Ethernet (RoCE) traffic. There is an equal bandwidth share between the two functions.

There is no dependency between the two functions. They can operate simultaneously or separately. The host can communicate with the embedded function as two separate hosts, each with its own MAC and IP addresses (configured as a standard interface).

In separated host mode, the host administrator is a trusted actor who can perform all configuration and management actions related to either network function.
This mode enables the same operational model of a SmartNIC (that does not have a separated control plane). In this case, the Arm control plane can be used for different functions but does not have any control on the host steering functions.

The limitations of this mode are as follows:

- Switchdev (virtual switch offload) mode is not supported on either of the functions
- SR-IOV is only supported on the host side

To configure separated host mode from DPU mode:

1. Enable separated host mode. Run:

   ```
   $ mst start
   $ mlxconfig -d /dev/mst/<device> s INTERNAL_CPU_MODEL=0
   ```

2. Power cycle.

3. Verify configuration. Run:

   ```
   $ mst start
   $ mlxconfig -d /dev/mst/<device> q | grep -i model
   ```

4. Remove OVS bridges configuration from the Arm-side. Run:

   ```
   $ ovs-vsctl list-br | xargs -r -l ovs-vsctl del-br
   ```
DOCA Switching

NVIDIA® BlueField® and NVIDIA® ConnectX® platforms provide robust support for diverse applications through hardware-based offloads, offering unparalleled scalability, performance, and efficiency.

This section lists the extensive switching capabilities enabled by DOCA libraries and services on these platforms. It includes detailed configurations of Open Virtual Switch (OVS) such as the setup of representors, virtualization options, and optional bridge configurations. These subsections guide users through the steps to effectively implement these software components.

DOCA Representors Model

1. **Note**
   
   This model is only applicable when the BlueField is operating DPU mode.

BlueField® DPU uses netdev representors to map each one of the host side physical and virtual functions:

1. Serve as the tunnel to pass traffic for the virtual switch or application running on the Arm cores to the relevant PF or VF on the Arm side.

2. Serve as the channel to configure the embedded switch with rules to the corresponding represented function.

Those representors are used as the virtual ports being connected to OVS or any other virtual switch running on the Arm cores.

When in ECPF ownership mode, we see 2 representors for each one of the DPU’s network ports: one for the uplink, and another one for the host side PF (the PF representor
created even if the PF is not probed on the host side). For each one of the VFs created on the host side a corresponding representor would be created on the Arm side. The naming convention for the representors is as follows:

- Uplink representors: p<port_number>
- PF representors: pf<port_number>hpf
- VF representors: pf<port_number>v<function_number>

The diagram below shows the mapping of between the PCIe functions exposed on the host side and the representors. For the sake of simplicity, we show a single port model (duplicated for the second port).

The red arrow demonstrates a packet flow through the representors, while the green arrow demonstrates the packet flow when steering rules are offloaded to the embedded switch. More details on that are available in the switch offload section.

Note
The MTU of host functions (PF/VF) must be smaller than the MTUs of both the uplink and corresponding PF/VF representor. For example, if the host PF MTU is set to 9000, both uplink and PF representor must be set to above 9000.

This section contains the following pages:

- VirtIO Acceleration through Hardware vDPA
- Bridge Offload
- Link Aggregation
- Controlling Host PF and VF Parameters

Note

DOCA also provides OpenvSwitch Acceleration (OVS in DOCA) which implements a virtual switch service, designed to work with NVIDIA NICs and DPUs to utilize ASAP ² (Accelerated Switching and Packet Processing) technology for data-path acceleration, providing the most efficient performance and feature set due to its architecture and use of DOCA libraries.

VirtIO Acceleration through Hardware vDPA

Hardware vDPA Installation

Hardware vDPA requires QEMU v2.12 (or with upstream 6.1.0) and DPDK v20.11 as minimal versions.
To install QEMU:

1. Clone the sources:
   
   ```
   git clone https://git.qemu.org/git/qemu.git
   cd qemu
   git checkout v2.12
   ```

2. Build QEMU:
   
   ```
   mkdir bin
   cd bin
   ../configure --target-list=x86_64-softmmu --enable-kvm
   make -j24
   ```

To install DPDK:

1. Clone the sources:
   
   ```
   git clone git://dpdk.org/dpdk
   cd dpdk
   git checkout v20.11
   ```

2. Install dependencies (if needed):
   
   ```
   yum install cmake gcc libnl3-devel libudev-devel make pkgconfig valgrind-devel pandoc
   libibverbs libmlx5 libmnl-devel -y
   ```

3. Configure DPDK:
   
   ```
   export RTE_SDK=$PWD
   make config T=x86_64-native-linuxapp-gcc
   cd build
   ```
4. Build DPDK:

   make -j

5. Build the vDPA application:

   cd $RTE_SDK/examples/vdpa/
   make -j

### Hardware vDPA Configuration

To configure huge pages:

   mkdir -p /hugepages
   mount -t hugetlbfs hugetlbfs /hugepages
   echo <more> > /sys/devices/system/node/node0/hugepages/hugepages-1048576kB/nr_hugepages
   echo <more> > /sys/devices/system/node/node1/hugepages/hugepages-1048576kB/nr_hugepages

To configure a vDPA VirtIO interface in an existing VM's xml file (using libvirt):

1. Open the VM's configuration XML for editing:

   virsh edit <domain name>

2. Perform the following:

   1. Change the top line to:
2. Assign a memory amount and use 1GB page size for huge pages (size must be the same as that used for the vDPA application), so that the memory configuration looks as follows:

```xml
<memory unit='KiB'>4194304</memory>
<currentMemory unit='KiB'>4194304</currentMemory>
<memoryBacking>
  <hugepages>
    <page size='1048576' unit='KiB'/>
  </hugepages>
</memoryBacking>
```

3. Assign an amount of CPUs for the VM CPU configuration, so that the vcpu and cputune configuration looks as follows:

```xml
<vcpu placement='static'>5</vcpu>
<cputune>
  <vcpupin vcpu='0' cpuset='14'/>
  <vcpupin vcpu='1' cpuset='16'/>
  <vcpupin vcpu='2' cpuset='18'/>
  <vcpupin vcpu='3' cpuset='20'/>
  <vcpupin vcpu='4' cpuset='22'/>
</cputune>
```

4. Set the memory access for the CPUs to be shared, so that the cpu configuration looks as follows:

```xml
<cpu mode='custom' match='exact' check='partial'>
  <model fallback='allow'>Skylake-Server-IBRS</model>
  <numa>
    <cell id='0' cpus='0-4' memory='8388608' unit='KiB' memAccess='shared'/>
  </numa>
</cpu>
```
5. Set the emulator in use to be the one built in step 2, so that the emulator configuration looks as follows:

```xml
<emulator><path to qemu executable></emulator>
```

6. Add a virtio interface using QEMU command line argument entries, so that the new interface snippet looks as follows:

```xml
<qemu:commandline>
  <qemu:arg value='-chardev'/>
  <qemu:arg value='socket,id=charnet1,path=/tmp/sock-virtio0'/>
  <qemu:arg value='-netdev'/>
  <qemu:arg value='vhost-user,chardev=charnet1,queues=16,id=hostnet1'/>
  <qemu:arg value='-device'/>
  <qemu:arg value='virtio-net-pci,mq=on,vectors=6,netdev=hostnet1,id=net1,mac=e4:11:c6:d3:45:f2,bus=pci.0,addr=0x6,
    page-per-vq=on,rx_queue_size=1024,tx_queue_size=1024'/>
</qemu:commandline>
```

**Note**

In this snippet, the vhostuser socket file path, the amount of queues, the MAC and the PCIe slot of the virtio device can be configured.

**Running Hardware vDPA**

**Note**

...
1. Create the ASAP\textsuperscript{2} environment:
   
   1. Create the VFs.
   2. Enter switchdev mode.
   3. Set up OVS.

2. Run the vDPA application:

   \texttt{cd \$RTE_SDK/examples/vdpa/build}
   
   \texttt{./vdpa \textasciitilde \textless VF PCI BDF\textgreater ,\texttt{class=vdpa \textasciitilde log-level=pmd,info \textasciitilde \textendash i}}

3. Create a vDPA port via the vDPA application CLI:

   \texttt{create /tmp/sock-virtio0 <PCI DEVICE BDF>}

\begin{itemize}
  \item \textbf{Note}
  
  The vhostuser socket file path must be the one used when configuring the VM.
\end{itemize}

4. Start the VM:

   \texttt{virsh start <domain name>}

Hardware vDPA supports switchdev mode only.
For further information on the vDPA application, visit the Vdpa Sample Application DPDK documentation.

Bridge Offload

Note

Bridge offload is supported switchdev mode only.

Note

Bridge offload is supported from kernel version 5.15 onward.

A Linux bridge is an in-kernel software network switch (based on and implementing a subset of IEEE 802.1D standard) used to connect Ethernet segments together in a protocol-independent manner. Packets are forwarded based on L2 Ethernet header addresses.

mlx5 provides the ability to offload bridge dataplane unicast packet forwarding and VLAN management to hardware.

Basic Configuration

1. Initialize the ASAP\(^2\) environment:
   1. Create the VFs.
   2. Enter switchdev mode.

2. Create a bridge and add mlx5 representors to bridge:
Configuring VLAN

1. Enable VLAN filtering on the bridge:

```bash
ip link set bridge0 type bridge vlan_filtering 1
```

2. Configure port VLAN matching (trunk mode). In this configuration, only packets with specified VID are allowed.

```bash
bridge vlan add dev enp8s0f0_0 vid 2
```

3. Configure port VLAN tagging (access mode). In this configuration, VLAN header is pushed/popped upon reception/transmission on port.

```bash
bridge vlan add dev enp8s0f0_0 vid 2 pvid untagged
```

VF LAG Support

Bridge supports offloading on bond net device that is fully initialized with mlx5 uplink representors and is in single (shared) FDB LAG mode. Details about initialization of LAG are provided in section "SR-IOV VF LAG".

To add a bonding net device to bridge:

```bash
ip link set bond0 master bridge0
```
For further information on interacting with Linux bridge via iproute2 bridge tool, refer to `man 8 bridge`.

## Link Aggregation

### Contents:

Network bonding enables combining two or more network interfaces into a single interface. It increases the network throughput, bandwidth and provides redundancy if one of the interfaces fails.

NVIDIA ® BlueField ® networking platforms (DPUs or SuperNICs) have an option to configure network bonding on the Arm side in a manner transparent to the host. Under such configuration, the host would only see a single PF.

**Note**

This functionality is supported when BlueField is set in embedded function ownership mode for both ports.

**Note**

While LAG is being configured (starting with step 2 under section "LAG Configuration"), traffic cannot pass through the physical ports.

The diagram below describes this configuration:
LAG Modes

Two LAG modes are supported on BlueField:

- Queue Affinity mode
- Hash mode

Queue Affinity Mode

In this mode, packets are distributed according to the QPs.

1. To enable this mode, run:

   ```
   $ mlxconfig -d /dev/mst/<device-name> s LAG_RESOURCE_ALLOCATION=0
   ```

   Example device name: mt41686_pciconf0.

2. Add/edit the following field from `/etc/mellanox/mlnx-bf.conf` as follows:
3. Perform a BlueField system reboot for the mlxconfig settings to take effect. Refer to the "NVIDIA BlueField Reset and Reboot Procedures" troubleshooting page for instructions.

Hash Mode

In this mode, packets are distributed to ports according to the hash on packet headers.

⚠️ Note

For this mode, prerequisite steps 3 and 4 are not required.

1. To enable this mode, run:

```
$ mlxconfig -d /dev/mst/<device-name> s LAG_RESOURCE_ALLOCATION=1
```

Example device name: mt41686_pciconf0.

2. Add/edit the following field from /etc/mellanox/mlnx-bf.conf as follows:

```
LAG_HASH_MODE="yes"
```

3. Perform a BlueField system reboot for the mlxconfig settings to take effect. Refer to the "NVIDIA BlueField Reset and Reboot Procedures" troubleshooting page for instructions.
Prerequisites

1. Set the LAG mode to work with.

2. (Optional) Hide the second PF on the host. Run:

```
$ mlxconfig -d /dev/mst/<device-name> s HIDE_PORT2_PF=True NUM_OF_PF=1
```

Example device name: mt41686.pciconf0.

**Note**

Perform a BlueField system reboot for the mlxconfig settings to take effect. Refer to the "NVIDIA BlueField Reset and Reboot Procedures" troubleshooting page for instructions.

3. Delete any installed Scalable Functions (SFs) on the Arm side.

4. Stop the driver on the host side. Run:

```
$ systemctl stop openibd
```

5. The uplink interfaces (p0 and p1) on the Arm side must be disconnected from any OVS bridge.

LAG Configuration

1. Create the bond interface. Run:

```
$ ip link add bond0 type bond
$ ip link set bond0 down
```
2. Subordinate both the uplink representors to the bond interface. Run:

   $ ip link set p0 down
   $ ip link set p1 down
   $ ip link set p0 master bond0
   $ ip link set p1 master bond0

3. Bring the interfaces up. Run:

   $ ip link set p0 up
   $ ip link set p1 up
   $ ip link set bond0 up

The following is an example of LAG configuration in Ubuntu:

   # cat /etc/network/interfaces

   # interfaces(5) file used by ifup(8) and ifdown(8)
   # Include files from /etc/network/interfaces.d:
   source /etc/network/interfaces.d/*
   auto lo
   iface lo inet loopback
   #p0
   auto p0
   iface p0 inet manual
       bond-master bond1

Note

While LAG is being configured (starting with the next step), traffic cannot pass through the physical ports.
As a result, only the first PF of the BlueFields would be available to the host side for networking and SR-IOV.

⚠️ **Warning**

When in shared RQ mode (enabled by default), the uplink interfaces (p0 and p1) must always stay enabled. Disabling them will break LAG support and VF-to-VF communication on same host.

For OVS configuration, the bond interface is the one that needs to be added to the OVS bridge (interfaces p0 and p1 should not be added). The PF representor for the first port (pf0hpf) of the LAG must be added to the OVS bridge. The PF representor for the second port (pf1hpf) would still be visible, but it should not be added to OVS bridge. Consider the following examples:

```bash
# #p1
auto p1
iface p1 inet manual
    bond-master bond1
#bond1
auto bond1
iface bond1 inet static
    address 192.168.1.1
    netmask 255.255.0.0
    mtu 1500
    bond-mode 2
    bond-slaves p0 p1
    bond-miimon 100
    pre-up (sleep 2 && ifup p0) &
    pre-up (sleep 2 && ifup p1) &

#p1 auto p1
iface p1 inet manual
    bond-master bond1
#bond1 auto bond1
iface bond1 inet static
    address 192.168.1.1
    netmask 255.255.0.0
    mtu 1500
    bond-mode 2
    bond-slaves p0 p1
    bond-miimon 100
    pre-up (sleep 2 && ifup p0) &
    pre-up (sleep 2 && ifup p1) &
```

```bash
ovs-vsctl add-br bf-lag
ovs-vsctl add-port bf-lag bond0
ovs-vsctl add-port bf-lag pf0hpf
```
⚠️ Warning

Trying to change bonding configuration in Queue Affinity mode (including bringing the subordinated interface up/down) while the host driver is loaded would cause FW syndrome and failure of the operation. Make sure to unload the host driver before altering BlueField bonding configuration to avoid this.

 אותן Note

When performing driver reload (openibd restart) or reboot, it is required to remove bond configuration and to reapply the configurations after the driver is fully up. Refer to steps 1-4 of "Removing LAG Configuration".

Removing LAG Configuration

1. If Queue Affinity mode LAG is configured (i.e., LAG_RESOURCE_ALLOCATION=0):

   1. Delete any installed Scalable Functions (SFs) on the Arm side.

   2. Stop driver (openibd) on the host side. Run:

       ```bash
       systemctl stop openibd
       ```

   2. Delete the LAG OVS bridge on the Arm side. Run:

       ```bash
       ovs-vsctl del-br bf-lag
       ```

       This allows for later restoration of OVS configuration for non-LAG networking.
3. Stop OVS service. Run:

```
systemctl stop openvswitch-switch.service
```

4. Run:

```
ip link set bond0 down
modprobe -rv bonding
```

As a result, both of the BlueField's network interfaces would be available to the host side for networking and SR-IOV.

5. For the host to be able to use BlueField's ports, make sure to attach the ECPF and host representor in an OVS bridge on the Arm side. Refer to "Virtual Switch on BlueField" for instructions on how to perform this.

6. Revert from `HIDE_PORT2_PF`, on the Arm side. Run:

```
mlxconfig -d /dev/mst/<device-name> s HIDE_PORT2_PF=False NUM_OF_PF=2
```

7. Restore default LAG settings in BlueField's firmware. Run:

```
mlxconfig -d /dev/mst/<device-name> s LAG_RESOURCE_ALLOCATION=DEVICE_DEFAULT
```

8. Delete the following line from `/etc/mellanox/mlnx-bf.conf` on the Arm side:

```
LAG_HASH_MODE=...
```

9. Perform a BlueField system reboot for the `mlxconfig` settings to take effect. Refer to the "NVIDIA BlueField Reset and Reboot Procedures" troubleshooting page for instructions.
LAG on Multi-host

Only LAG hash mode is supported with BlueField multi-host.

LAG Multi-host Prerequisites

1. Enable LAG hash mode.

2. Hide the second PF on the host. Run:

   ```bash
   $ mlxconfig -d /dev/mst/<device-name> s HIDE_PORT2_PF=True NUM_OF_PF=1
   ```

3. Make sure NVME emulation is disabled:

   ```bash
   $ mlxconfig -d /dev/mst/<device-name> s NVME_EMULATION_ENABLE=0
   ```

   Example device name: mt41686_pciconf0.

4. The uplink interfaces (p0 and p4) on the Arm side, representing port0 and port1, must be disconnected from any OVS bridge. As a result, only the first PF of BlueField would be available to the host side for networking and SR-IOV.

LAG Configuration on Multi-host

1. Create the bond interface. Run:

   ```bash
   $ ip link add bond0 type bond
   $ ip link set bond0 down
   $ ip link set bond0 type bond miimon 100 mode 4 xmit_hash_policy layer3+4
   ```

2. Subordinate both the uplink representors to the bond interface. Run:
3. Bring the interfaces up. Run:

```bash
$ ip link set p0 down
$ ip link set p4 down
$ ip link set p0 master bond0
$ ip link set p4 master bond0
$ ip link set p0 up
$ ip link set p4 up
$ ip link set bond0 up
```

4. For OVS configuration, the bond interface is the one that must be added to the OVS bridge (interfaces p0 and p4 should not be added). The PF representor, pf0hpf, must be added to the OVS bridge with the bond interface. The rest of the uplink representors must be added to another OVS bridge along with their PF representors. Consider the following examples:

```bash
ovs-vsctl add-br br-lag
ovs-vsctl add-port br-lag bond0
ovs-vsctl add-port br-lag pf0hpf
ovs-vsctl add-br br1
ovs-vsctl add-port br1 p1
ovs-vsctl add-port br1 pf1hpf
ovs-vsctl add-br br2
ovs-vsctl add-port br2 p2
ovs-vsctl add-port br2 pf2hpf
ovs-vsctl add-br br3
ovs-vsctl add-port br3 p3
ovs-vsctl add-port br3 pf3hpf
```

Note
Removing LAG Configuration on Multi-host

Refer to section "Removing LAG Configuration".

Controlling Host PF and VF Parameters

Contents:

NVIDIA® BlueField® networking platforms (DPUs or SuperNICs) allow control over some of the networking parameters of the PFs and VFs running on the host side.

Setting Host PF and VF Default MAC Address

From the Arm, users may configure the MAC address of the physical function in the host. After sending the command, users must reload the NVIDIA driver in the host to see the newly configured MAC address. The MAC address goes back to the default value in the FW after system reboot.

Example:

```
$ echo "c4:8a:07:a5:29:59" > /sys/class/net/p0/smart_nic/pf/mac
$ echo "c4:8a:07:a5:29:61" > /sys/class/net/p0/smart_nic/vf0/mac
```

Setting Host PF and VF Link State
vPort state can be configured to Up, Down, or Follow. For example:

```bash
$ echo "Follow" > /sys/class/net/p0/smart_nic/pf/vport_state
```

### Querying Configuration

To query the current configuration, run:

```bash
$ cat /sys/class/net/p0/smart_nic/pf/config
MAC : e4:8b:01:a5:79:5e
MaxTxRate : 0
State : Follow
```

Zero signifies that the rate limit is unlimited.

### Disabling Host Networking PFs

It is possible to not expose ConnectX networking functions to the host for users interested in using storage or VirtIO functions only. When this feature is enabled, the host PF representors (i.e. `pf0` and `pf1`) will not be seen on the Arm.

- Without a PF on the host, it is not possible to enable SR-IOV, so VF representors will not be seen on the Arm either
- Without PFs on the host, there can be no SFs on it

To disable host networking PFs, run:

```bash
mlxconfig -d /dev/mst/mt41686_pciconf0 s NUM_OF_PF=0
```

To reactivate host networking PFs:

- For single-port BlueFields, run:
For dual-port BlueFields, run:

mlxconfig -d /dev/mst/mt41686_pciconf0 s NUM_OF_PF=1

Note

When there are no networking functions exposed on the host, the reactivation command must be run from the Arm.

Note

Perform a BlueField system reboot for the mlxconfig settings to take effect. Refer to the "NVIDIA BlueField Reset and Reboot Procedures" troubleshooting page for instructions.
NVIDIA DOCA with OpenSSL

This guide provides instructions on using DOCA SHA for OpenSSL implementations.

Introduction

The doca_sha_offload_engine is an OpenSSL dynamic engine with the ability of offloading SHA calculation. It can offload the OpenSSL one-shot SHA-1, SHA-256, and SHA-512. It supports synchronous mode and asynchronous mode by leveraging the OpenSSL async_jobs library. For more information on the async_jobs library, please refer to official OpenSSL documentation.

This engine is based on the doca_sha library and the OpenSSL dynamic engine interface API. For more information on the OpenSSL dynamic engine, please refer to official OpenSSL documentation.

This engine can be called by an OpenSSL application through the OpenSSL high-level algorithm call interface, EVP_Digest. For more information on the EVP_Digest, please refer to official OpenSSL documentation.

Prerequisites

- Hardware-based doca_sha engine which can be verified by calling doca_sha_get_hardware_supported()
- Installed OpenSSL version ≥ 1.1.1

Architecture

The following diagram shows the software hierarchy of doca_sha_offload_engine and its location in the whole DOCA repository.

From the perspective of OpenSSL, this engine is an instantiation of the OpenSSL dynamic engine interface API by leveraging the doca_sha library.
<table>
<thead>
<tr>
<th>Capabilities and Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Only one-shot OpenSSL SHA is supported</td>
</tr>
<tr>
<td>- The maximum message length ≤ 2GB, the same as doca_sha library</td>
</tr>
</tbody>
</table>

**OpenSSL Command Line Verification**

Verify that the engine can be loaded:

```
$ openssl engine dynamic -pre NO_VCHECK:1 -pre
SO_PATH:${DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_sha_offload_engine.so -pre
LOAD -vvv -t -c
(dynamic) Dynamic engine loading support
[Success]:
SO_PATH:${DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_sha_offload_engine.so
[Success]: LOAD
Loaded: (doca_sha_offload_engine) OpenSSL SHA offloading engine based on doca_sha
[SHA1, SHA256, SHA512]
```
For SHA-1:

$ echo "hello world" | openssl dgst -sha1 -engine
{DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_sha_offload_engine.so -engine_impl

For SHA-256:

$ echo "hello world" | openssl dgst -sha256 -engine
{DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_sha_offload_engine.so -engine_impl

For SHA-512:

$ echo "hello world" | openssl dgst -sha512 -engine
{DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_sha_offload_engine.so -engine_impl

OpenSSL Throughput Test

openssl-speed is the OpenSSL throughput benchmark tool. For more information, consult official OpenSSL documentation. doca_sha_offload_engine throughput can also be measured using openssl-speed.

- SHA-1, each job 10000 bytes, using engine:

$ openssl speed -evp sha1 -bytes 10000 -elapsed --engine
{DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_sha_offload_engine.so

- SHA-256, each job 10000 bytes, using engine, async_jobs=256:
Using DOCA SHA Offload Engine in OpenSSL Application

More information on the dynamic engine usage can be found in the official OpenSSL documentation.

1. To load the doca_sha_offload_engine (optionally, set engine PCIe address):

```c
ENGINE *e;
const char *doca_engine_path = "$(DOCA_DIR)/infrastructure/doca_sha_offload_engine/libdoca_sha_offload_engine.so";
const char *default_doca_pci_addr = "03:00.0";
ENGINE_load_dynamic();
e = ENGINE_by_id(doca_engine_path);
ENGINE_ctrl_cmd_string(e, "set_pci_addr", doca_engine_pci_addr, 0);
ENGINE_init(e);
ENGINE_set_default_digests(e);
```

2. To perform SHA calculation by calling the OpenSSL high-level function EVP_XXX:

```c
const EVP_MD *evp_md = EVP_sha1();
EVP_MD_CTX *mdctx = EVP_MD_CTX_create();
EVP_DigestInit_ex(mdctx, evp_md, e);
EVP_DigestUpdate(mdctx, msg, msg_len);
EVP_DigestFinal_ex(mdctx, digest, digest_len);
```
3. To unload the engine:

```c
EVP_MD_CTX_destroy(mdctx);

ENGINE_unregister_digests(e);
ENGINE_finish(e);
ENGINE_free(e);
```
NVIDIA BlueField DPU
Scalable Function User Guide

This document provides an overview and configuration of scalable functions (sub-functions, or SFs) for NVIDIA® BlueField® DPU.

Introduction

Scalable functions (SFs), or sub-functions, are very similar to virtual functions (VFs) which are part of a Single Root I/O Virtualization (SR-IOV) solution. I/O virtualization is one of the key features used in data centers today. It improves the performance of enterprise servers by giving virtual machines direct access to hardware I/O devices. The SR-IOV specification allows one PCI Express (PCIe) device to present itself to the host as multiple distinct "virtual" devices. This is done with a new PCIe capability structure added to a traditional PCIe function (i.e., a physical function or PF).

The PF provides control over the creation and allocation of new VFs. VFs share the device's underlying hardware and PCIe. A key feature of the SR-IOV specification is that VFs are very lightweight so that many of them can be implemented in a single device.

To utilize the capabilities of VF in the BlueField, SFs are used. SFs allow support for a larger number of functions than VFs, and more importantly, they allow running multiple services concurrently on the DPU.

An SF is a lightweight function which has a parent PCIe function on which it is deployed. The SF, therefore, has access to the capabilities and resources of its parent PCIe function and has its own function capabilities and its own resources. This means that an SF would also have its own dedicated queues (i.e., txq, rxq).

SFs co-exist with PCIe SR-IOV virtual functions (on the host) but also do not require enabling PCIe SR-IOV.
SFs support E-Switch representation offload like existing PF and VF representors. An SF shares PCIe-level resources with other SFs and/or with its parent PCIe function.

Prerequisites

Refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField related software.

- Make sure your firmware version is 20.30.1004 or higher
- To enable SF support on the device, change the PCIe address for each port:

```
$ mlxconfig -d 0000:03:00.0 s PF_BAR2_ENABLE=0 PER_PF_NUM_SF=1 PF_TOTAL_SF=236
PF_SF_BAR_SIZE=10
```

PF_BAR2_ENABLE: if this config is set, then all PFs and ECPFs have the same number of SFs. This should be off (deprecated).
If set, PF_TOTAL_SF and PF_SF_BAR_SIZE won't work.
PER_PF_NUM_SF: If this config is set, each PF and ECPF configure/control its own number of SFs.
THE ABOVE TWO CONFIGS AFFECT BOTH BF AND HOST, TREAT WITH CARE!
Also, only one of them can be set. It is INVALID to set them both

PF_TOTAL_SF: maximum number of SFs we wish to configure for the given PF/ECPF.
PF_SF_BAR_SIZE: size of each SF at the BAR2. The size is in powers of 2 in KB.
For example: PF_SF_BAR_SIZE=10 means each SF is taking 1MB of the BAR.
PF_TOTAL_SF=14 means this PCI function can create up to 14 SFs.
In total: FW will allocate 14MB of BAR2.

Note
Perform a BlueField system-level reset for the mlxconfig settings to take effect.

SF Configuration

To use an SF, a 3-step setup sequence must be followed first:

1. Create.
2. Configure.
3. Deploy.

These steps can be performed using mlxdevm tool.
When working on top of an upstream-based kernel, on which the mlxdevm tool is unavailable, please refer to the Upstream Guide on Scalable Functions for instructions on using the devlink tool which should be used instead.

**Configuration Using mlxdevm Tool**

1. Create the SF.

SFs are managed using the mlxdevm tool supplied with iproute2 package. The tool is found at `/opt/mellanox/iproute2/sbin/mlxdevm`.

An SF is created using the mlxdevm tool. The SF is created by adding a port of pcisf flavor.

To create an SF port representor, run:

```bash
/opt/mellanox/iproute2/sbin/mlxdevm port add pci/<pci_address> flavour pcisf pfnum <corresponding pfnum> sfnum <sfnum>
```

**Note**

Each SF must have a unique number (`<sfnum>`).

For example:

```bash
/opt/mellanox/iproute2/sbin/mlxdevm port add pci/0000:03:00.0 flavour pcisf pfnum 0 sfnum 4
```

Output example:
The number 229409 is required to complete the following two steps (i.e., configuration and deployment).

pci/0000:03:00.0/229409 is called the SF index.

pci/<pci_address>/<sf_index> can be replaced with <representor_name>. For example:

pci/0000:03:00.0/229409 = en3f0pf0sf4

To see information about the created SF such as its MAC address, trust mode, or state (active/inactive), run the following command:

```
/opt/mellanox/iproute2/sbin/mlxdevm port show
```

Output example:

```
pci/0000:30:00.0/229409: type eth netdev en3f0pf0sf4 eth0 flavor pcisf controller 0 pfnum 0 sfnum 4
  function:
    hw_addr 00:00:00:00:00:00 state inactive opstate detached roce true max_uc_macs 128 trust off
```

2. Configure the SF.

A subfunction representor (SF port representor) is created but it is not deployed yet. Users should configure the hardware address (e.g., MAC address), set trust mode to on, and activate the SF before deploying it.
The following steps can be executed as separate commands (at any order) or combined as one:

- To configure the hardware address, run:

  ```
  /opt/mellanox/iproute2/sbin/mlxdevm port function set pci/<pci_address>/<sf_index> hw_addr <MAC address>
  ```

- To set the trust mode to on, run:

  ```
  /opt/mellanox/iproute2/sbin/mlxdevm port function set pci/<pci_address>/<sf_index> trust on
  ```

- To activate the created SF, run:

  ```
  /opt/mellanox/iproute2/sbin/mlxdevm port function set pci/<pci_address>/<sf_index> state active
  ```

Alternatively, to configure the MAC address, set trust mode on, and set the state as active, run:

```
/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/<pci_address>/<sf_index> hw_addr <mac_address> trust on state active
```

For example:

```
/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/229409 hw_addr 00:00:00:00:04:0 trust on state active
```

**Note**
3. Deploy the SF.

To unbind the SF from the default config driver and bind the actual SF driver, run:

```
echo mlx5_core.sf.<next_serial> > /sys/bus/auxiliary/drivers/mlx5_core.sf_cfg/unbind
echo mlx5_core.sf.<next_serial> > /sys/bus/auxiliary/drivers/mlx5_core.sf/bind
```

For example:

```
echo mlx5_core.sf.4 > /sys/bus/auxiliary/drivers/mlx5_core.sf_cfg/unbind
echo mlx5_core.sf.4 > /sys/bus/auxiliary/drivers/mlx5_core.sf/bind
```

**Note**

<nest_serial> is a number produced by the firmware when creating the SF (this is the gvmi number of the SF). mlxdevm tool when creating the SF. To obtain it, refer to the *useful commands* provided below.

Useful commands:

- To see the available sub-functions, run:

  ``
  $ devlink dev show
  ``

  For example, if you run the command before creating, configuring, and deploying the SF (using the steps detailed earlier), the output would appear as
follows:

```
pci/0000:03:00.0
pci/0000:03:00.1
auxiliary/mlx5_core.sf.2
auxiliary/mlx5_core.sf.3
```

After creating, configuring, and deploying the SF, the output would be:

```
pci/0000:03:00.0
pci/0000:03:00.1
auxiliary/mlx5_core.sf.2
auxiliary/mlx5_core.sf.3
auxiliary/mlx5_core.sf.4
```

Note that the `<next_serial>` number is 4 for the created SF.

- To see the `sfnum` of each sub-function, run:

```
cat /sys/bus/auxiliary/devices/mlx5_core.sf.<next_serial>/sfnum
```

For example:

```
cat /sys/bus/auxiliary/devices/mlx5_core.sf.4/sfnum
```

Example output:

```
cat /sys/bus/auxiliary/devices/mlx5_core.sf.4/sfnum
4
```

- To remove an SF, you must first make its state inactive and only then remove the SF representer.
To make the SF's state inactive, run:

```
/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/<pci_address>/<sf_index> state inactive
```

To delete the SF port representor, run:

```
/opt/mellanox/iproute2/sbin/mlxdevm port del pci/<pci_address>/<sf_index>
```

For example:

```
/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/229409 state inactive
/opt/mellanox/iproute2/sbin/mlxdevm port del pci/0000:03:00.0/229409
```

4. Use the SF.
Running the application on the DPU requires OVS configuration. By creating SFs, an SF representor for the OVS is also created and named en3f0pf*sf*. Therefore, each representor needs to be connected to the correct OVS bridge.

**Note**

Two SFs related to the same PCIe are necessary for the configuration in the illustration.

The following example configures 2 SFs and adds their representors to the OVS.

1. Create, configure, and deploy the SFs. Run:

   ```
   /opt/mellanox/iproute2/sbin/mlxdevm port add pci/0000:03:00.0 flavour pcisf pfnum 0 sfnum 4
   ```
Using the command `mlxdevm port show`, you can see the SF indices of the created SFs.

Output example:

```
pci/0000:30:00.0/229409: type eth netdev en3f0pf0sf4 flavour pcisf controller 0 pfnum 0 sfnum 4
  function:
    hw_addr 00:00:00:00:00:00 state inactive opstate detached roce true max_uc_macs 128 trust off
pci/0000:30:00.0/229410: type eth netdev en3f0pf0sf5 flavour pcisf controller 0 pfnum 0 sfnum 5
  function:
    hw_addr 00:00:00:00:00:00 state inactive opstate detached roce true max_uc_macs 128 trust off
```

2. Configure the MAC address, set trust mode on, and activate the created SFs:

```
/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/229409 hw_addr 02:25:f2:8d:a2:4c trust on state active
/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/229410 hw_addr 02:25:f2:8d:a2:5c trust on state active
```

Using `ifconfig`, you may see that there are 2 added network interfaces: `en3f0pf0sf4` and `en3f0pf0sf5` for the two respective SF port representors.

3. Delete existing OVS bridges (optional).

For example, run the following command to delete an OVS bridge called `ovsbr1`:
4. Create two bridges `sf_bridge1` and `sf_bridge2` and configure them as follows:

```bash
ovs-vsctl del-br ovsbr1

ovs-vsctl add-br sf_bridge1
ovs-vsctl add-br sf_bridge2
ovs-vsctl add-port sf_bridge1 p0
ovs-vsctl add-port sf_bridge2 pf0hpf
```

5. Add the port representors to the OVS bridges:

```bash
ovs-vsctl add-port sf_bridge1 en3f0pf0sf4
ovs-vsctl add-port sf_bridge2 en3f0pf0sf5
```

The OVS bridges after adding the SF representors:

```plaintext
Bridge sf_bridge1
  Port p0
    Interface p0
  Port sf_bridge1
    Interface sf_bridge1
      type: internal
  Port en3f0pf0sf4
    Interface en3f0pf0sf4

Bridge sf_bridge2
  Port sf_bridge2
    Interface sf_bridge2
      type: internal
  Port en3f0pf0sf5
    Interface en3f0pf0sf5
  Port pf0hpf
    Interface pf0hpf

ovs_version: "2.14.1"
```
To run the application, use the following command to initialize the SFs during runtime:

```
*Executable_binary* -a auxiliary:mlx5_core.sf.* -a auxiliary:mlx5_core.sf.*
```

For example:

```
doca_<app_name> -a auxiliary:mlx5_core.sf.4 -a auxiliary:mlx5_core.sf.5 -- [application_flags]
```
NVIDIA TLS Offload Guide

This guide provides an overview and configuration steps of TLS hardware offloading via kernel-TLS, using hardware capabilities of NVIDIA® BlueField® DPU.

Introduction

Transport layer security (TLS) is a cryptographic protocol designed to provide communications security over a computer network. The protocol is widely used in applications such as email, instant messaging, and voice over IP (VoIP), but its use in securing HTTPS remains the most publicly visible.

The TLS protocol aims primarily to provide cryptography, including privacy (confidentiality), integrity, and authenticity using certificates, between two or more communicating computer applications. It runs in the application layer and is itself composed of two layers: the TLS record and the TLS handshake protocols.

TLS works over TCP and consists of 3 phases:

1. Handshake – establishment of a connection
2. Application – sending and receiving encrypted packets
3. Termination – connection termination

TLS Handshake

In the handshake phase, the client and server decide on which cipher suites they will use, and exchange keys and certificates according to the following flow:

1. Client hello, provides the server at a minimum with the following:
   - A key exchange algorithm, to determine how symmetric keys are exchanged
   - An authentication or digital signature algorithm, which dictates how server authentication and client authentication (if required) are implemented
- A bulk encryption cipher, which is used to encrypt the data
- A hash/MAC (message authentication code) function, which determines how data integrity checks are carried out
- The version of the protocol it understands
- The cipher suites it is capable of working with
- A unique random number, which is important to guard against replay attacks

2. Server hello:

- Selects a cipher suite
- Generates its own random number
- Assigns a session ID to the TLS connection
- Sends enough information to complete a key exchange—most often, this means sending a certificate including an RSA public key

3. Client:

- Responsible for completing the key exchange using the information the server provided

At this point, the connection is secured, both sides have agreed on an encryption algorithm, a MAC algorithm, and respective keys.

**kTLS**

The Linux kernel provides TLS offload infrastructure. kTLS (kernel TLS) offloads TLS handling from the user-space to the kernel-space.

kTLS has 3 modes of operation:

- **SW** – all operation is handled in kernel (i.e., handshake, encryption, decryption)
• HW-offload (the focus of this guide) – handshake and error handling are performed in software. Packets are encrypted/decrypted in hardware. In this case, there is an additional offload from the kernel to the hardware.

• HW-record – all operations are handled by the hardware (driver and firmware) including the handshake. It also handles its own TCP session. This option is currently not supported.

Note

It is important to understand that Rx (receiving) and Tx (sending) can have two separate modes. For example, Rx can be dealt in SW mode but Tx in HW-offload mode (i.e., the hardware will only encrypt but not decrypt).

HW-offloading kTLS

In general, the TLS HW-offload performs best and provides optimal value on longer lived sessions, with relatively large packets. Scaling in terms of concurrent connections and connections per second is use-case dependent (e.g., the amount of active concurrent connections from the overall open concurrent connections is material).

It is necessary to learn the following terms before proceeding:

• The transport interface send (TIS) object is responsible for performing all transport-related operations of the transmit side. Messages from Send Queues (SQs) get segmented and transmitted by the TIS including all transport required implications. For example, in the case of a large send offload, the TIS is responsible for the segmentation. The NVIDIA® ConnectX® hardware uses a TIS object to save and access the TLS crypto information and state of an offloaded Tx kTLS connection.

• The transport interface receive (TIR) object is responsible for performing all transport-related operations on the receive side. TIR performs the packet processing and reassembly and is also responsible for demultiplexing the packets into different receive queues (RQs).
• Both TIS and TIR hold the data encryption key (DEK).

**kTLS Offload Flow in High Level**

![Note]

The following flow does not include resync and errors.

1. Establishes a TLS connection with remote host (server or client) by handling a TLS handshake by kernel on current host.

2. Initializes the following state for each connection, Rx and Tx:
   - Crypto secrets (e.g., public key)
   - Crypto processing state
   - Record metadata (e.g., record sequence number, offset)
   - Expected TCP sequence number

**Tx flow:**

1. Packets belonging to device offloaded sockets arrive to the kernel and it does not encrypt them.

2. Kernel performs record framing and marks the packet with a connection identifier.

3. Kernel sends packets to the device driver for offloading.

4. Device checks that the sequence number matches the state in the TIS and performs encryption and authentication.

**Rx flow:**

1. When the connection is created, a HW steering rule is added to steer packets to their respective TIR.
2. Device receives the packet then validates and checks that sequence number of TCP matches the state in the TIR.

3. Performs decryption and authentication, and indicates in the CQE (completion queue entry).

4. Kernel understands that the packet is already decrypted so it does not decrypt it itself and passes it on to the user-space.

**Resync and Error Handling**

When the sequence number does not match expectations or if any other error occurs, the hardware gives control back to the SW which handles the problem.

See more about kTLS modes, resync, and error handling in the Linux Kernel documentation.

**Prerequisites**

All commands in this section should be performed on host (not on BlueField) unless stated otherwise.

**Checking Hardware Support for Crypto Acceleration**

To check if the BlueField or ConnectX have crypto acceleration, run the following command from host:

```
host> mst start  # turn on mst driver
host> flint -d <device under /dev/mst/ directory> dc | grep Crypto
```

The output should include Crypto Enabled. For example:

```
host> flint -d /dev/mst/mt41686_pCONF0 dc | grep Crypto
```
Kernel Requirements

- Operating system must be either:
  - FreeBSD 13.0+
  - A Linux distribution built on Linux kernel version 5.3 or later for Tx support and version 5.9 or later for Rx support. We recommend using the latest version when possible for the best available optimizations.

Note

TIS Pool optimization is added to Linux kernel version 6.0. Instead of creating TIS per new connection, unused TIS from previous connection, will be recycled. This will improve Tx connection rate. No further installations required beyond installing the kernel itself.

- Check the current kernel version on the host. Run:

  host> uname -r

- The kernel must be configured to support TLS by setting the options TLS_DEVICE and MLX5_TLS to y. To check if TLS is configured, run:

  host> cat /boot/config-$\{uname -r\} | grep TLS
Example output:

```bash
host> cat /boot/config-5.4.0-121-generic | grep TLS
... CONFIG_TLS_DEVICE=y
CONFIG_MLX5_TLS=y
... 
```

If the current kernel does not support one of the options, you can change the configurations and recompile, or build a new kernel.

![Note](image)

Follow the build instructions provided with the kernel provider.

Schematic flow for building a Linux kernel:

1. Enter the Linux kernel directory downloaded (usually in `/usr/src`):

   ```bash
   host> make menuconfig # Set TLS_DEVICE=y and MLX5_TLS=y in options. Setting location in the menu can be found by pressing '/' and typing 'setting'.
   host> make -j <num-of-cores> && make -j <num-of-cores> modules_install && make -j <num of cores> install
   ```

2. Update the grub to the new configured kernel then reboot.

**Configurations and Useful Commands**

**TLS Setup**
Finding NVIDIA Interfaces

```
host> mst start       # if mst driver is not loaded.
host> mst status -v

NVIDIA's netdev interfaces are found be under the NET column.

For example:

```
host> mst status -v
....
DEVICE_TYPE       MST          PCI      RDMA   NET              NUMA
BlueField2(rev:0) /dev/mst/mt41686_pciconf0.1 b1:00.1 mlx5_1       net-ens5f1  1
BlueField2(rev:0) /dev/mst/mt41686_pciconf0 b1:00.0 mlx5_0       net-ens5f0  1
```

In this example, the interfaces ens5f1 and ens5f0 are NVIDIA's netdev interfaces.
Configuring TLS Offload

- To check if the offload option is on or off, run:

  ```
  host> ethtool -k $iface | grep tls
  ```

  Example output:

  ```
  tls-hw-tx-offload: on
  tls-hw-rx-offload: off
  tls-hw-record: off [fixed]
  ```

**Note**

-tls-hw-record is not required for the device as kTLS does not support "HW Record" mode.

- To turn Tx offload on or off:

  ```
  host> ethtool -K $iface tls-hw-tx-offload <on | off>
  ```

- To turn Rx offload on or off:

  ```
  host> ethtool -K $iface tls-hw-rx-offload <on | off>
  ```
Configuring OVS Bridge on BlueField

When the host is connected to a BlueField device, an OVS bridge must be configured on the BlueField so traffic passes bidirectionally from host to uplink. If no OVS bridge is configured, the host is isolated from the network (see diagram above).

Note

On BlueField image version 3.7.0 or higher the default OVS configuration can be used without additional modifications.

To configure the OVS bridge on BlueField, run the following commands on BlueField:

```
dpu> for br in $(ovs-vsctl list-br); do ovs-vsctl del-br $br; done # erasing existing bridges
dpu> ovs-vsctl add-br ovs-br0 && ovs-vsctl add-port ovs-br0 p0 && ovs-vsctl add-port ovs-br0 pf0hpf
dpu> ovs-vsctl add-br ovs-br1 && ovs-vsctl add-port ovs-br1 p1 && ovs-vsctl add-port ovs-br1 pf1hpf
dpu> ovs-vsctl set Open_vSwitch . other_config:hw-offload=true && systemctl restart openvswitch-switch
```

Where p0/p1 are the uplink interfaces and pf0hpf/pf1hpf are the interfaces facing the host.

Common Use Cases

OpenSSL

OpenSSL is an all-around cryptography library that offers open-source application of the TLS protocol. It is the main library for using kTLS and other applications since Nginx depends on it as their base library.

Note
kTLS is supported only in OpenSSL version 3.0.0 or higher, and only on the supported kernel versions. The supported OpenSSL version is available for download from distro packages, or it can be downloaded and compiled from the OpenSSL GitHub.

**Warning**

Many modules depend on OpenSSL. Changing the default version may cause problems. Adding `--prefix=/var/tmp/ssl --openssldir=/var/tmp/ssl` in the `./Configure` command below may prevent the built OpenSSL from becoming the default one used by the system. Make sure the directory of the OpenSSL you build manually is not located in any paths listed in the PATH environment variable.

1. Check the version of the default OpenSSL:

```
host> openssl version
```

2. Follow OpenSSL installation instructions from OpenSSL's supplied guides. During the configuration process, make sure to set the `enable-ktls` option before building it by running it from within the OpenSSL directory (works in version 3.0 and higher). For example:

```
host> ./Configure linux-$({uname -p}) enable-ktls --prefix=/var/tmp/ssl --openssldir=/var/tmp/ssl
```

Add "threads" as well for multithread support.
3. Check if kTLS is enabled in OpenSSL by running the following command from within the OpenSSL directory, and check whether ktls is listed under Enabled features:

```
host> perl configdata.pm --dump | less
```

If OpenSSL has been downloaded manually, the OpenSSL executable would be located in the `/<openssl-dir>/apps/` directory. For example, checking the version from within OpenSSL directory is done using the command `./apps/openssl version`.

**Note**

Installing a new OpenSSL requires recompiling user tools that were configured over OpenSSL (e.g., Nginx).

**Note**

In OpenSSL's master source code, there is a feature "Support for kTLS Zero-Copy sendfile() on Linux" (Zero-Copy commit). If the Zero-Copy option is set, `SSL_sendfile()` uses the Zero-Copy TX mode which means that the data itself is not copied from the user space to Kernel space. This gives a performance boost when used with kTLS hardware offload. Be aware that invalid TLS records may be transmitted if the file is changed while being sent.

**Nginx**

Nginx is a free and open-source software web server that can also be used as a reverse proxy, load balancer, mail proxy and HTTP cache. Nginx can be configured to depend
on OpenSSL library and therefore Nginx could have the great advantages of TLS HW-offload on ConnectX-6 Dx, ConnectX-7 or the DPU.

**Prerequisites**

Refer to the [OpenSSL](#) section for setting OpenSSL.

**Configuration**

1. Install dependencies. For Ubuntu distribution, for example:

   ```
   host> apt install libpcre3 libpcre3-dev
   ```

2. Clone Nginx's repository and enter directory:

   ```
   host> git clone https://github.com/nginx/nginx.git && cd nginx
   ```

3. Configure Nginx components to support kTLS:

   ```
   host> ./auto/configure --with-openssl=/<insert_path_to_openssl_directory> --with-debug --with-http_ssl_module --with-openssl-opt="enable-ktls -DOPENSSL_LINUX_TLS -g3"
   ```

4. Build Nginx:

   ```
   host> make -j <num of cores> && sudo make -j <num-of-cores> install
   ```

**Note**
If make fails with a deprecated openssl functions error, remove `-Werror` for `CFLAGS` in `objs/Makefile` and try again.

5. Add the following lines to the end of the `/usr/local/nginx/conf/nginx.conf` file (before the last closing bracket):

```nginx
server {
    listen 443 ssl default_server reuseport;
    server_name localhost;
    root /tmp/nginx/docs/html/;

    include /etc/nginx/default.d/*.conf;
    ssl_certificate /usr/local/nginx/conf/cert.pem;
    ssl_certificate_key /usr/local/nginx/conf/key.pem;
    ssl_ciphers ECDHE-RSA-AES128-GCM-SHA256;
    ssl_protocols TLSv1.2;

    location / {
        index index.html;
    }

    error_page 404 /404.html;
    location = /40x.html {
    }

    error_page 500 502 503 504 /50x.html;
    location = /50x.html {
    }
}
```

6. Notice that the key and certificate of the Nginx server should be located in `/usr/local/nginx/conf/`. Therefore, after creating a key and certificate (as mentioned in section "Adding Certificate and Key") they should be copied to the aforementioned directory:

```bash
host> cp key.pem /usr/local/nginx/conf/ && cp cert.pem /usr/local/nginx/conf/
```
7. To run Nginx:

```
host> cd nginx && objs/nginx
```

This command starts Nginx Server in the background.

### Stopping Nginx

```
host> pkill nginx
```

### Wrk – Client

A simple client for requesting Nginx's server is "wrk". It can be installed by running the following:

```
host> git clone https://github.com/wg/wrk.git && cd wrk/ && make -j <num-of-cores>
```

### Using Wrk

The following is an example of using the wrk client to request the page index.html from the Nginx server in address 4.4.4.4 (run within wrk's directory):

```
host> taskset -c 0 ./wrk -t1 -c10 -d30s https://4.4.4.4:443/index.html
```

⚠️ **Note**
Testing Offload via OpenSSL

This chapter demonstrates how to test the kTLS hardware offload.

Note

Make sure to refer to section "OpenSSL" before proceeding.

TLS Testing Setup

For testing purposes, a server and a client are required. The testing section only tests a single setup of a host and BlueField-2 or a host ConnectX which will participate either as a server or as a client. Setting a back-to-back setup of the same kind and installing the same OpenSSL version can help avoid misconfigurations. Nevertheless, it is required to have the same OpenSSL version on both the client and server.

Make sure the desired kTLS is configured as detailed in section "Configuring TLS Offload". To test hardware offload, make sure tls-hw-tx-offload and/or tls-hw-rx-offload are on. To test kTLS software mode, make sure to turn them off.

In addition, make sure both hosts (server and client) can communicate bidirectionally through ConnectX or BlueField. One can set the interface that supports the offload (on the host) with an IP, in same subnet. Make sure that when using BlueField, an OVS bridge is set on BlueField as shown in "Configuring OVS Bridge on BlueField".
Adding Certificate and Key

The server side should create a certificate and key. The client can also use a certificate, but it is not necessary for this test case. Run the following command in the installed OpenSSL directory and fill in all the requested details:

```
host> openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem -days 365 -nodes
```

The following files are created:

- `key.pem` – private-key file used to generate the CSR and, later, to secure and verify connections using the certificate
- `cert.pem` – certificate signing request (CSR) file used to order your SSL certificate and, later, to encrypt messages that only its corresponding private key can decrypt

⚠️ Note
The server side should be run before client side so that client's request are answered by server.

### Running Server Side

The following example works on OpenSSL version 3.1.0:

```
host> openssl s_server -key key.pem -cert cert.pem -tls1_2 -cipher ECDHE-RSA-AES128-GCM-SHA256 -accept 443 -ktls
```

**Note**

Notice the `-ktls` flag.

**Note**

Refer to official OpenSSL documentation on `s_server` for more information.

In this example, the key and certificate are provided, the cipher suite and TLS version are configured, and the server listens to port 443 and is instructed to use kTLS.

### Running Client Side

The following example works on OpenSSL version 3.1.0:
Where 4.4.4.4 is the IP of the remote server.

Note

Refer to official OpenSSL documentation on `s_client` for more information.

Testing kTLS

After the connection is established (handshake is done), a prompt will open and the user, both on the client and server side, can send a message to other side in a chat-like manner. Messages should appear on the other side once they are received.

The following example checks kTLS hardware offload on the tested setup by tracking Rx and Tx TLS on device counters:

```
host> openssl s_client -connect 4.4.4.4:443 -tls1_2
```

```
Note

Refer to official OpenSSL documentation on `s_client` for more information.
```

To check kTLS over kernel counters:

```
host> ethtool -S $iface | grep -i 'tx_tls_encrypted\|rx_tls_decrypted' # ($iface is the interface that offloads)
```

```
To check kTLS over kernel counters:

```

```

Output example:

```
Note
```

```

```
The comments are not part of the output and are added as explanation.

```bash
host> cat /proc/net/tls_stat
TlsCurrTxSw 0 # Current Tx connections opened in SW mode
TlsCurrRxSw 0 # Current Rx connections opened in SW mode
TlsCurrTxDevice 0 # Current Tx connections opened in HW-offload mode
TlsCurrRxDevice 0 # Current Rx connections opened in HW-offload mode
TlSTxSw 2323828 # Accumulated number of Tx connections opened in SW mode
TlsRxSw 1 # Accumulated number of Rx connections opened in SW mode
TlSTxDevice 12203652 # Accumulated number of Tx connections opened in HW-offload mode
TlsRxDevice 0 # Accumulated number of Rx connections opened in HW-offload mode
TlsDecryptError 0 # Failed record decryption (e.g., due to incorrect authentication tag)
TlsRxDeviceResync 0 # Rx resyncs sent to HW's handling cryptography
TlsDecryptRetry 0 # All Rx records re-decrypted due to TLS_RX_EXPECT_NO_PAD misprediction
TlsRxNoPadViolation 0 # Data Rx records re-decrypted due to TLS_RX_EXPECT_NO_PAD misprediction
```

**Note**

More information about the kernel counters can be found in the Statistics section of the Kernel TLS documentation.

**Optimizations over kTLS**

**XLIO**
The NVIDIA accelerated IO (XLIO) software library boosts the performance of TCP/IP applications based on Nginx (e.g., CDN, DoH) and storage solutions as part of SPDK. XLIO is a user-space software library that exposes standard socket APIs with kernel-bypass architecture, enabling a hardware-based direct copy between an application’s user-space memory and the network interface. In particular, XLIO can boost the performance of applications that use the kTLS hardware offload as OpenSSL and Nginx. Read more about XLIO in the NVIDIA XLIO Documentation and XLIO TLS HW-offload over kTLS in the TLS HW Offload section.

**Note**

Even though XLIO is a kernel-bypass library, the kernel must support kTLS for the bypass to work properly.

**Performance Tuning Options**

TLS offload performance is related to how fast data can be pumped though the offload engine. In the case of user space applications, certain system configurations can be tuned to optimize its performance.

The following are items that can be tuned for optimal performance, mainly focusing on dedicating the server’s work to the NUMA, or non-uniform memory access, cores:

**Note**

Non-uniform memory access (NUMA) cores are cores with a dedicated memory for each of them, granting cores fast access to their own memory and slower access to others'. This architecture is best for scenarios when it is not necessary to share memory between cores.
1. Add NUMA cores of the NIC to the isolcpus kernel boot arguments for each server so that the kernel scheduler does not interrupt the core's running user thread. The following are examples of adding commands:

1. Identify the NIC NUMA node (see NUMA column):

```bash
host> mst status -v
DEVICE_TYPE      MST          PCI     RDMA     NET           NUMA
ConnectX6DX(rev:0) /dev/mst/mt4125_pciconf0 41:00.0 mlx5_0  net-enp65s0f0np0
```

2. Identify the cores of the NIC NUMA node using the NUMA node number acquired from the previous output:

```bash
host> lscpu | grep "NUMA node1"
NUMA node1 CPU(s): 1,3,5,7,9,11,13,15,17,19,21,23
```

3. Add the NIC NUMA cores to a grub file (e.g., /etc/default/grub) by adding the line GRUB_CMDLINE_LINUX_DEFAULT="isolcpus=<NUMA-cores-from-previous-output>". For example:

```bash
GRUB_CMDLINE_LINUX_DEFAULT="isolcpus=1,3,5,7,9,11,13,15,17,19,21,23"
```

4. Update grub:

```bash
host> sudo update-grub
```

5. Reboot and check that the configuration has been applied:

```bash
host> cat /proc/cmdline
```
2. Disable `irqbalance` service:

**Note**

Interrupt request, or IRQ, determines what hardware interrupts arrive to each core.

```
host> service irqbalance stop
```

3. Run `set_irq_affinity.sh` to redistribute IRQs to various cores.

**Note**

The script is within MLNX_OFED's sources:

1. You can find it in MLNX_OFED downloads.

2. Under "Download" select the correct version and download the "SOURCES".tgz file.

3. Extract the .tgz.

4. Under SOURCES, extract the mlnx_tools.

You should find both files `set_irq_affinity.sh` and its helper file `common_irq_affinity.sh` under the sbin directory.
4. Set the interface RSS to the number of cores to use:

```
host> ./set_irq_affinity.sh <ConnectX_or_BlueField_network_interface>
```

5. Set the interface queues for number of cores to use:

```
host> ethtool -X <ConnectX_or_BlueField_network_interface> equal <number_of_isolcpus_cores>

host> ethtool -L <ConnectX_or_BlueField_network_interface> combined <number_of_isolcpus_cores>
```

6. Pin the application with `taskset` to the `isolcpus` cores used. For example:

```
host> taskset -c 1,3,5,7,9,11,13,15,17,19,21,23 openssl s_server -key key.pem -cert cert.pem -tls1_2 -cipher ECDHE-RSA-AES128-GCM-SHA256 -accept 443 -ktls
```

**Additional Reading**

- [Linux kernel TLS documentation](#)
- [Linux kernel TLS offload documentation](#)
- [Autonomous NIC offloads](#) research paper
NVIDIA DOCA
Troubleshooting Guide

This guide provides troubleshooting information for common issues and misconfigurations encountered when using DOCA for NVIDIA® BlueField® DPU.

DOCA Infrastructure

RShim Troubleshooting and How-Tos

Another backend already attached

Several generations of BlueField DPUs are equipped with a USB interface in which RShim can be routed, via USB cable, to an external host running Linux and the RShim driver.

In this case, typically following a system reboot, the RShim over USB prevails and the DPU host reports RShim status as "another backend already attached". This is correct behavior, since there can only be one RShim backend active at any given time. However, this means that the DPU host does not own RShim access.

To reclaim RShim ownership safely:

1. Stop the RShim driver on the remote Linux. Run:

   ```bash
   systemctl stop rshim
   systemctl disable rshim
   ```

2. Restart RShim on the DPU host. Run:

   ```bash
   systemctl enable rshim
   systemctl start rshim
   ```
The "another backend already attached" scenario can also be attributed to the RShim backend being owned by the BMC in DPUs with integrated BMC. This is elaborated on further down on this page.

**RShim driver not loading**

Verify whether your DPU features an integrated BMC or not. Run:

```
# sudo lspci -s $(sudo lspci -d 15b3: | head -1 | awk '{print $1}') -vvv | grep "Product Name"
```

Example output for DPU **with integrated BMC**:

```
Product Name: BlueField-2 DPU 25GbE Dual-Port SFP56, integrated BMC, Crypto and Secure Boot Enabled, 16GB on-board DDR, 1GbE OOB management, Tall Bracket, FHHL
```

If your DPU has an integrated BMC, refer to [RShim driver not loading on host with integrated BMC](#).

If your DPU does not have an integrated BMC, refer to [RShim driver not loading on host on DPU without integrated BMC](#).

**RShim driver not loading on DPU with integrated BMC**

**RShim driver not loading on host**

1. Access the BMC via the RJ45 management port of the DPU.

2. Delete RShim on the BMC:

   - systemctl stop rshim
   - systemctl disable rshim

3. Enable RShim on the host:
4. Restart RShim service. Run:

```
sudo systemctl restart rshim
```

If RShim service does not launch automatically, run:

```
sudo systemctl status rshim
```

This command is expected to display "active (running)".

5. Display the current setting. Run:

```
# cat /dev/rshim<N>/misc | grep DEV_NAME
DEV_NAME        pcie-0000:04:00.2
```

This output indicates that the RShim service is ready to use.

**RShim driver not loading on BMC**

1. Verify that the RShim service is not running on host. Run:

```
systemctl status rshim
```

If the output is active, then it may be presumed that the host has ownership of the RShim.

2. Delete RShim on the host. Run:
3. Enable RShim on the BMC. Run:

```bash
systemctl enable rshim
systemctl start rshim
```

4. Display the current setting. Run:

```bash
# cat /dev/rshim<N>/misc | grep DEV_NAME
DEV_NAME        usb-1.0
```

This output indicates that the RShim service is ready to use.

**RShim driver not loading on host on DPU without integrated BMC**

1. Download the suitable DEB/RPM for RShim (management interface for DPU from the host) driver.

2. Reinstall RShim package on the host.
   - For Ubuntu/Debian, run:

     ```bash
     sudo dpkg --force-all -i rshim-<version>.deb
     ```
   - For RHEL/CentOS, run:

     ```bash
     sudo rpm -Uhv rshim-<version>.rpm
     ```
3. Restart RShim service. Run:

```bash
sudo systemctl restart rshim
```

If RShim service does not launch automatically, run:

```bash
sudo systemctl status rshim
```

This command is expected to display "active (running)".

4. Display the current setting. Run:

```bash
# cat /dev/rshim<N>/misc | grep DEV_NAME
DEV_NAME   pcie-0000:04:00.2
```

This output indicates that the RShim service is ready to use.

**Change ownership of RShim from NIC BMC to host**

1. Verify that your card has BMC. Run the following on the host:

```bash
# sudo lspci -s $(sudo lspci -d 15b3: | head -1 | awk '{print $1}') -vvv | grep "Product Name"
Product Name: BlueField-2 DPU 25GbE Dual-Port SFP56, integrated BMC, Crypto and Secure Boot Enabled, 16GB on-board DDR, 1GbE OOB management, Tall Bracket, FHHL
```

The product name is supposed to show "integrated BMC".

2. Access the BMC via the RJ45 management port of the DPU.

3. Delete RShim on the BMC:
4. Enable RShim on the host:

```
 systemctl enable rshim
 systemctl start rshim
```

5. Restart RShim service. Run:

```
 sudo systemctl restart rshim
```

If RShim service does not launch automatically, run:

```
 sudo systemctl status rshim
```

This command is expected to display "active (running)".

6. Display the current setting. Run:

```
 # cat /dev/rshim<N>/misc | grep DEV_NAME
 DEV_NAME     pcie-0000:04:00.2
```

This output indicates that the RShim service is ready to use.

---

**Connectivity Troubleshooting**

**Connection (ssh, screen console) to the DPU is lost**
The UART cable in the Accessories Kit (OPN: MBF20-DKIT) can be used to connect to the DPU console and identify the stage at which BlueField is hanging.

Follow this procedure:

1. Connect the UART cable to a USB socket, and find it in your USB devices.

   ```bash
   sudo lsusb
   Bus 002 Device 003: ID 0403:6001 Future Technology Devices International, Ltd FT232 Serial (UART) IC
   ```

   **Note**
   
   For more information on the UART connectivity, please refer to the DPU's hardware user guide under Supported Interfaces > Interfaces Detailed Description > NC-SI Management Interface.

   **Info**
   
   It is good practice to connect the other end of the NC-SI cable to a different host than the one on which the BlueField DPU is installed.

2. Install the minicom application.

<table>
<thead>
<tr>
<th>OS</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>CentOS/RHEL</td>
<td>sudo yum install minicom -y</td>
</tr>
</tbody>
</table>
3. Open the minicom application.

```
sudo minicom -s -c on
```

4. Go to "Serial port setup".

5. Enter "F" to change "Hardware Flow control" to NO.

6. Enter "A" and change to /dev/ttyUSB0 and press Enter.

7. Press ESC.

8. Type on "Save setup as dfl".

9. Exit minicom by pressing Ctrl + a + z.

**Driver not loading in host server**

What this looks like in dmsg:

```
[275604.216789] mlx5_core 0000:af:00.1: 63.008 Gb/s available PCIe bandwidth, limited by 8 GT/s x8 link at 0000:ae:00.0 (capable of 126.024 Gb/s with 16 GT/s x8 link)
[275624.187596] mlx5_core 0000:af:00.1: wait_fw_init:316:(pid 943): Waiting for FW initialization, timeout abort in 100s
[275644.152994] mlx5_core 0000:af:00.1: wait_fw_init:316:(pid 943): Waiting for FW initialization, timeout abort in 79s
[275664.118404] mlx5_core 0000:af:00.1: wait_fw_init:316:(pid 943): Waiting for FW initialization, timeout abort in 59s
[275684.083806] mlx5_core 0000:af:00.1: wait_fw_init:316:(pid 943): Waiting for FW initialization, timeout abort in 39s
```
The driver on the host server is dependent on the Arm side. If the driver on Arm is up, then the driver on the host server will also be up.

Please verify that:

- The driver is loaded in the BlueField DPU
- The Arm is booted into OS
- The Arm is not in UEFI Boot Menu
- The Arm is not hanged

Then:

1. Perform **graceful shutdown**.
2. Power cycle on the host server.
3. If the problem persists, reset nvconfig (`sudo mlxconfig -d /dev/mst/<device> -y reset`) and power cycle the host.

**Note**

If your DPU is VPI capable, please be aware that this configuration will reset the link type on the network ports to IB. To change the network port's link type to Ethernet, run:

```
sudo mlxconfig -d <device> s LINK_TYPE_P1=2 LINK_TYPE_P2=2
```
4. If this problem persists, please make sure to install the latest bfb image and then restart the driver in host server. Please refer to this page for more information.

**No connectivity between network interfaces of source host to destination device**

Verify that the bridge is configured properly on the Arm side.

The following is an example for default configuration:

```
$ sudo ovs-vsctl show
f6740bfb-0312-4cd8-88c0-a9680430924f
  Bridge ovsbr1
    Port pf0sf0
      Interface pf0sf0
    Port p0
      Interface p0
    Port pf0hpf
      Interface pf0hpf
  Port ovsbr1
    Interface ovsbr1
      type: internal
  Bridge ovsbr2
    Port p1
      Interface p1
    Port pf1sf0
      Interface pf1sf0
    Port pf1hpf
      Interface pf1hpf
  Port ovsbr2
    Interface ovsbr2
      type: internal
  ovs_version: "2.14.1"
```

If no bridge configuration exists, refer to "Virtual Switch on DPU".

**Uplink in Arm down while uplink in host server up**
Please check that the cables are connected properly into the network ports of the DPU and the peer device.

**Performance Degradation**

Degradation in performance indicates that openvswitch may not be offloaded.

Verify offload state. Run:

```
# ovs-vsctl get Open_vSwitch . other_config:hw-offload
```

- If `hw-offload = true` – Fast Pass is configured (desired result)
- If `hw-offload = false` – Slow Pass is configured

If `hw-offload = false`:

- For RHEL/CentOS, run:

  ```
  # ovs-vsctl set Open_vSwitch . other_config:hw-offload=true;
  # systemctl restart openvswitch;
  # systemctl enable openvswitch;
  # ovs-vsctl set Open_vSwitch . other_config:hw-offload=true;
  # /etc/init.d/openvswitch-switch restart
  ```

- For Ubuntu/Debian, run:

  ```
  # ovs-vsctl set Open_vSwitch . other_config:hw-offload=true;
  # /etc/init.d/openvswitch-switch restart
  ```

**SR-IOV Troubleshooting**
Unable to create VFs

1. Please make sure that SR-IOV is enabled in BIOS.

2. Verify `SRIOV_EN` is true and `NUM_OF_VFS` bigger than 1. Run:

   ```bash
   # mlxconfig -d /dev/mst/mt41686_pciconf0 -e q | grep -i "SRIOV_EN\|num_of_vf"
   Configurations:           Default         Current         Next Boot
   *        NUM_OF_VFS       16              16              16
   *        SRIOV_EN         True(1)         True(1)         True(1)
   ``

3. Verify that `GRUB_CMDLINE_LINUX="iommu=pt intel_iommu=on pci=assign-busses"`.

No traffic between VF to external host

1. Please verify creation of representors for VFs inside the Bluefield DPU. Run:

   ```bash
   # /opt/mellanox/iproute2/sbin/rdma link |grep -i up
   ... link mlx5_0/2 state ACTIVE physical_state LINK_UP netdev pf0vf0
   ...
   ```

2. Make sure the representors of the VFs are added to the bridge. Run:

   ```bash
   # ovs-vsctl add-port <bridage_name> pf0vf0
   ```

3. Verify VF configuration. Run:

   ```bash
   $ ovs-vsctl show
   bb993992-7930-4dd2-bc14-73514854b024
     Bridge ovsbr1
       Port pf0vf0
         Interface pf0vf0
   ```
eSwitch Troubleshooting

Unable to configure legacy mode

To set devlink to "Legacy" mode in BlueField, run:

```
# devlink dev eswitch set pci/0000:03:00.0 mode legacy
# devlink dev eswitch set pci/0000:03:00.1 mode legacy
```

Please verify that:

- No virtual functions are open. To verify if VFs are configured, run:

```
# /opt/mellanox/iproute2/sbin/rdma link | grep -i up
link mlx5_0/2 state ACTIVE physical_state LINK_UP netdev pf0vf0
```
If any VFs are configured, destroy them by running:

```bash
# echo 0 > /sys/class/infiniband/mlx5_0/device/mlx5_num_vfs
# echo 0 > /sys/class/infiniband/mlx5_1/device/mlx5_num_vfs
```

- If any SFs are configured, delete them by running:

```bash
/sbin/mlnx-sf -a delete --sfindex <SF-Index>
```

**Note**

You may retrieve the `<SF-Index>` of the currently installed SFs by running:

```bash
# mlnx-sf -a show
```

SF Index: pci/0000:03:00.0/229408
- Parent PCI dev: 0000:03:00.0
- Representor netdev: en3f0pf0sf0
- Function HWADDR: 02:61:f6:21:32:8c
- Auxiliary device: mlx5_core.sf.2
  - netdev: enp3s0f0s0
  - RDMA dev: mlx5_2

SF Index: pci/0000:03:00.1/294944
- Parent PCI dev: 0000:03:00.1
- Representor netdev: en3f1pf1sf0
- Function HWADDR: 02:30:13:6a:2d:2c
- Auxiliary device: mlx5_core.sf.3
  - netdev: enp3s0f1s0
  - RDMA dev: mlx5_3
Pay attention to the SF Index values. For example:

```
/sbin/mlnx-sf -a delete --sfindex pci/0000:03:00.0/229408
/sbin/mlnx-sf -a delete --sfindex pci/0000:03:00.1/294944
```

If the error "Error: mlx5_core: Can't change mode when flows are configured" is encountered while trying to configure legacy mode, please make sure that

1. Any configured SFs are deleted (see above for commands).

2. Shut down the links of all interfaces, delete any `ip xfrm` rules, delete any configured OVS flows, and stop openvswitch service. Run:

```
ip link set dev p0 down
ip link set dev p1 down
ip link set dev pf0hpf down
ip link set dev pf1hpf down
ip link set dev vxlan_sys_4789 down

ip x s f ;
ip x p f ;

tc filter del dev p0 ingress
tc filter del dev p1 ingress
tc qdisc show dev p0
tc qdisc show dev p1
tc qdisc del dev p0 ingress
tc qdisc del dev p1 ingress
tc qdisc show dev p0
tc qdisc show dev p1

systemctl stop openvswitch-switch
```

**DPU appears as two interfaces**

What this looks like:
• Check if you are working in legacy mode.

```
# devlink dev eswitch show pci/0000:03:00.<0|1>
```

If the following line is printed, this means that you are working in legacy mode:

```
pcci/0000:03:00.<0|1>: mode legacy inline-mode none encap enable
```

Please configure the DPU to work in switchdev mode. Run:

```
devlink dev eswitch set pci/0000:03:00.<0|1> mode switchdev
```

• Check if you are working in separated mode:

```
# mlxconfig -d /dev/mst/mt41686_pciconf0 q | grep -i cpu
* INTERNAL_CPU_MODEL SEPERATED_HOST(0)
```

Please configure the DPU to work in embedded mode. Run:

```
# mlxconfig -d /dev/mst/mt41686_pciconf0 s INTERNAL_CPU_MODEL=1
```

**DOCA Applications**
This chapter deals with troubleshooting issues related to DOCA applications.

**EAL Initialization Failure**

EAL initialization failure is a common error that may appear while running various DPDK-related applications.

**Error**

The error looks like this:

```
[DOCA][ERR][NUTILS]: EAL initialization failed
```

There may be many causes for this error. Some of them are as follows:

- The application requires huge pages and none were allocated
- The application requires root privileges to run and it was run without elevated privileges

**Solution**

The following solutions are respective to the possible causes listed above:

- Allocate huge pages. For example, run (on the host or the DPU, depending on where you are running the application):

  ```
sudo echo 2048 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
  ```

- Run the application using sudo (or as root):

  ```
sudo <run_command>
  ```
Ring Memory Issue

This is a common memory issue when running application on the host.

Error

The error looks as follows:

RING: Cannot reserve memory
[13:00:57:290147][DOCA][ERR][UFLTR::Core:156]: DPI init failed

The most common cause for this error is lack of memory (i.e., not enough huge pages per worker threads).

Solution

Possible solutions:

- Recommended: Increase the amount of allocated huge pages. Instructions for allocating huge pages can be found here.

Note

For an SFT application with 64 cores, it is recommended to increase the allocation from 2048 to 8192.

- Alternatively, one can also limit the number of cores used by the application:
  - `-c <core-mask>` – Set the hexadecimal bitmask of the cores to run on.
-l <core-list> – list of cores to run on.

For example:

```
./doca_<app_name> -a 3b:00.3 -a 3b:00.4 -l 0-64 -- -l 60
```

**DOCA Apps Using DPDK in Parallel Issue**

When running two DOCA apps in parallel that use DPDK, the first app runs but the second one fails.

**Error**

The following error is received:

```
Failed to start URL Filter with output: EAL: Detected 16 Icore(s)
EAL: Detected 1 NUMA nodes
EAL: RTE Version: 'MLNX_DPDK 20.11.4.0.3' EAL: Detected shared linkage of DPDK
EAL: Cannot create lock on '/var/run/dpdk/rte/config'. Is another primary process running?
EAL: FATAL: Cannot init config
EAL: Cannot init config
[15:01:57:246339][DOCA][ERR][NUTILS]: EAL initialization failed
```

The cause of the error is that the second application is using `/var/run/dpdk/rte/config` when the first application is already using it.

**Solution**

To run two applications in parallel, the second application needs to be run with DPDK EAL option `--file-prefix <name>`.

In this example, after running the first application (without adding the `eal` option), to run the second with the EAL option. Run:
Failure to Set Huge Pages

When trying to configure the huge pages from an unprivileged user account, a permission error is raised.

Error

Configuring the huge pages results in the following error:

```
sudo echo 600 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
-bash: /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages: Permission denied
```

Solution

Using `sudo` with `echo` works differently than users usually expect. Instead, the command should be as follows:

```
$ echo '600' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

DOCA Libraries

This chapter deals with troubleshooting issues related to DOCA libraries.

DOCA Flow Error

When trying to add new entry to the pipe, an error is received.
**Error**

The error happens after trying to add new entry function. The error message would look similar to the following:

```
mlx5_common: Failed to create TIR using DevX
mlx5_net: Port 0 cannot create DevX TIR.
[10:26:39:622581][DOCA][ERR][dpdk_engine]: create pipe entry fail on index:1, error=Port 0 create flow fail, type 1 message: cannot get hash queue, type=8
```

The issue here seems to be caused by SF/ports configuration.

**Solution**

To fix the issue, apply the following commands on the DPU:

```
dpu# /opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode legacy
dpu# /opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode legacy
dpu# echo none > /sys/class/net/p0/compat/devlink/encap
dpu# echo none > /sys/class/net/p1/compat/devlink/encap
dpu# /opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.0 mode switchdev
dpu# /opt/mellanox/iproute2/sbin/devlink dev eswitch set pci/0000:03:00.1 mode switchdev
```

**DOCA SDK Compilation**

This chapter deals with troubleshooting issues related to compiling DOCA-based programs to use the DOCA SDK (e.g., missing dependencies).

**Meson Complains About Missing Dependencies**

As part of DOCA's installation, a basic set of environment variables are defined so that projects (such as DOCA applications) could easily compile against the DOCA SDK, and to allow users easy access to the various DOCA tools. In addition, the set of DOCA
applications sometimes rely on various 3rd party dependencies, some of which require specific environment variables so to be correctly found by the compilation environment (meson).

**Error**

There are multiple forms this error may appear in, such as:

- **DOCA libraries are missing:**
  
  Run-time dependency doca-common found: NO (tried pkgconfig)
  
  meson.build:230:0: ERROR: Dependency "doca-common" not found, tried pkgconfig

- **DPDK definitions are missing:**
  
  Dependency libdpdk found: NO (tried pkgconfig and cmake)
  
  meson.build:41:1: ERROR: Dependency "libdpdk" not found, tried pkgconfig and cmake

- **mpicc is missing for DPA All to All application:**
  
  ================
  Skipped Applications
  ================
  * dpa_all_to_all: Missing mpicc

**Solution**

All the dependencies mentioned above are installed as part of DOCA’s installation, and yet it is recommended to check that the packages themselves were installed correctly. The packages that install each dependency define the environment variables needed by it, and apply these settings per user login session:
• If DOCA was just installed (on the host or DPU), user session restart is required to apply these definitions (i.e., log off and log in).

• It is important to compile DOCA using the same logged in user. Logging as ubuntu and using sudo su, or compiling using sudo, will not work.

If restarting the user session is not possible (e.g., automated non-interactive session), the following is a list of the needed environment variables:

⚠️ Note

All the following examples use the required environment variables for the DPU. For the host, the values should be adjusted accordingly (aarch64 is for the DPU and x86 is for the host): aarch64-linux-gnu x86_64-linux-gnu.

Tip

It is recommended to define all of the following settings so as to not have to remember which DOCA application requires which module (whether DPDK, FlexIO, etc).

DOCA Tools:

• For Ubuntu:

```bash
export PATH=${PATH}:/opt/mellanox/doca/tools
```

• For CentOS:
DOCA Applications:

- For Ubuntu and CentOS

```bash
export PATH=${PATH}:/opt/mellanox/doca/tools
export PATH=${PATH}:/usr/mpi/gcc/openmpi-4.1.7a1/bin
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/mpi/gcc/openmpi-4.1.7a1/lib
```

DPDK:

- For Ubuntu:

```bash
export PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/opt/mellanox/dpdk/lib/aarch64-linux-gnu/pkgconfig
```

- For CentOS:

```bash
export PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/opt/mellanox/dpdk/lib64/pkgconfig
```

FlexIO:

- For Ubuntu:

```bash
export PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/opt/mellanox/flexio/lib/pkgconfig
```

- For CentOS:

```bash
export PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/opt/mellanox/flexio/lib/pkgconfig
```
CollectX:

- For Ubuntu and CentOS:

  ```
  export PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/opt/mellanox/collectx/lib/aarch64-linux-gnu/pkgconfig
  ```

Meson Complains About Permissions

Our guides for compiling the reference samples and applications of DOCA's SDK are using the meson build system.

Error

A permission error is encountered when trying to reuse a build directory from a previous build:

```
meson /tmp/build
```

```
Traceback (most recent call last):
  File "/usr/lib/python3/dist-packages/mesonbuild/mesonmain.py", line 146, in run
    return options.run_func(options)
  File "/usr/lib/python3/dist-packages/mesonbuild/msetup.py", line 294, in run
    app.generate()
  File "/usr/lib/python3/dist-packages/mesonbuild/msetup.py", line 181, in generate
    mlog.initialize(env.get_log_dir(), self.options.fatal_warnings)
  File "/usr/lib/python3/dist-packages/mesonbuild/mlog.py", line 103, in initialize
    log_file = open(os.path.join(logdir, log_fname), 'w', encoding='utf-8')
```

Solution

Per the meson build instructions, the user can choose any write-accessible directory to be used as the build directory, using the following syntax:
When reusing a build directory, it is best to ensure that the existing directory was created by a user with the same permissions, and only then do one of the following:

- Removing the old build directory:

  ```bash
  rm -rf /tmp/build
  ```

- Reconfiguring the build directory:

  ```bash
  meson --reconfigure /tmp/build
  ```

The above error is an indication that the build directory was created by a different user, and that our user doesn't have permissions to use it. In such cases, it is best to choose a different build directory, in a directory that our user has write-access to. For example:

  ```bash
  meson /tmp/build2
  ```

### Static Compilation on CentOS: Undefined References to C++

When statically compiling against the DOCA SDK on RHEL 7.x machines, there could be a conflict between the libstdc++ version available out-of-the-box and the one used when building DOCA's SDK libraries.

**Error**

There are multiple forms this error may appear in, such as:
$ cc test.o -o test_out `pkg-config --static doca`  
/opt/mellanox/doca/lib64/libdoca_common.a(doca_common_core_src_doca_dev.cpp.o): In function 'doca_devinfo_rep_list_create': 
  (.text.experimental+0x2193): undefined reference to '__cxa_throw_bad_array_new_length'  
/opt/mellanox/doca/lib64/libdoca_common.a(doca_common_core_src_doca_dev.cpp.o): In function 'doca_devinfo_rep_list_create': 
  (.text.experimental+0x2198): undefined reference to '__cxa_throw_bad_array_new_length'  
collect2: error: ld returned 1 exit status

Solution

Upgrading the devtoolset on the machine to the one used when building the DOCA SDK resolves the undefined references issue:

$ sudo yum install epel-release  
$ sudo yum install centos-release-scl-rh  
$ sudo yum install devtoolset-8  
# This will enable the use of devtoolset-8 to the *current* bash session  
$ source /opt/rh/devtoolset-8/enable

Static Compilation on CentOS: Unresolved Symbols

When statically compiling against the DOCA SDK on RHEL 7.x machines, a known issue in the default pkg-config version (0.27) causes a linking error.

Error

There are multiple forms this error may appear in. For example:

$ cc test.o -o test_out 'pkg-config --static doca' ...  
/opt/mellanox/dpdk/lib64/librte_net_mlx5.a(net_mlx5_mlx5_sft.c.o): In function 'mlx5_sft_start':  
mlx5_sft.c:(.text+0x1827): undefined reference to 'mlx5_malloc' ...
Solution

Use an updated version of `pkg-config` or `pkgconf` instead when building applications (as is recommended in DPDK's compilation instructions).

Cross-compiling DOCA and CUDA

This chapter deals with troubleshooting issues related to DOCA-CUDA cross-compilation.

Application Build Error

When trying to build with meson, an architecture-related error is received.

Error

The error may happen when trying to build DOCA or DOCA-CUDA applications.

```
cc1: error: unknown value 'corei7' for -march
```

It indicates that some dependency (usually `libdpdk`) is not taken from the host machine (i.e., the machine the executable file should be running on). This dependency should be taken from the Arm dependencies directories (the path is specified in the cross file) but is skipped if the host's `PKG_CONFIG_PATH` environment variable is used instead.

Solution

Make sure that the cross file contains the following `PKG_CONFIG` related definitions:

```
[built-in options]
pkg_config_path = "" [properties]
```
In addition, verify that `pkg_config_libdir` properly points to all `pkgconfig`-related directories under your cross-build root directory, and that the dependency reported in the error is not missing.

### DOCA Services (Containers)

This section deals with troubleshooting issues related to DOCA-based containers.

**YAML Syntax Error #1**

When deploying the container using the respective YAML file, the pod fails to start.

**Error**

The error may happen after modifying a service's YAML file, or after copying an example YAML file from one of the guides.

```
$ crictl pods
POD ID CREATED STATE NAME NAMESPACE ATTEMPT
RUNTIME
$ journalctl -u kubelet
...
Oct 06 12:10:08 dpu-name kubelet[3260]: E1006 12:10:08.552306   3260 file.go:108] "Unable to process watch event" err="can't process config file "/etc/kubelet.d/file_name.yaml": invalid pod: [metadata.name: Invalid value: "-dpu-name": a lowercase RFC 1123 subdomain must consist of lower case alphanumeric characters, '-', or '.', and must start and end with an alphanumeric character (e.g. 'example.com'), regex used for validation is '[a-z0-9][-a-z0-9]*[a-z0-9]?\.[a-z0-9][-a-z0-9]*[a-z0-9]?)+') spec.containers: Required value"
...
```

This indicates that some of the fields in the YAML file fail to comply with RFC 1123.
Solution

Both the pod name and container name have a strict alphabet (RFC 1123) restrictions. This means that users can only use dash ("-") and not underscore ("_") as the latter is an illegal character and cannot be used in the pod/container name. However, for the container's image name, use underscore ("_") instead of dash ("-”) to help differentiate the two.

YAML Syntax Error #2

When deploying the container using the respective YAML file, the pod fails to start.

Error

The error may happen after modifying a service's YAML file, or after copying an example YAML file from one of the guides.

Note

This error can occur when there is a whitespace issue if the YAML file has been copied from one of the guides causing a formatting mistake. It is important to ensure that the space characters used in the files are indeed spaces (" ") and not some other whitespace character.

$ crictl pods
POD ID CREATED STATE NAME NAMESPACE ATTEMPT
RUNTIME
$ journalctl -u kubelet
This indicates that there is a probable indentation issue in line 48 or in the line above it.

**Solution**

Go over the file and make sure that the file only uses spaces (" ") for indentations (2 per indent). Using any other number of spaces causes undefined behavior.

**Missing Huge Pages**

When deploying the container using the respective YAML file, the pod fails to start.

**Error**

```
$ crictl pods
POD ID      CREATED             STATE       NAME                NAMESPACE       ATTEMPT
RUNTIME
$ journalctl -u kubelet
... 
```

This error indicates that the service expected 1GB (1021313024 bytes) of huge pages of size 2MB per page, and could not find them.

**Solution**

1. Remove the YAML file of the service from the deployment directory (/etc/kubelet.d).
2. Allocate huge pages as described in the service's prerequisites steps:

   1. Make sure that the huge pages are allocated as required per the desired container.

   2. Both the amount and size of the pages are important and must match precisely.

3. Restart the container infrastructure daemons:

   ```
   sudo systemctl restart kubelet.service
   sudo systemctl restart containerd.service
   ```

4. Once the above operations are completed successfully, the container could be deployed (YAML can be copied to `/etc/kubelet.d`).

---

**Failed to Reserve Sandbox Name**

After rebooting the DPU, the respective pods start. However, the containers repeatedly fail to spawn and their "attempt" counter does not increment.

**Error**

```
$ crictl pods
POD ID CREATED STATE NAME NAMESPACE
ATTEMPT RUNTIME
bee147792a85b Less than a second ago Ready doca-hbn-service-my-dpu default 0 (default)
ea66ee46e75a5 Less than a second ago Ready doca-telemetry-service-my-dpu default 0 (default)
```

```
$ crictl ps -a
CONTAINER IMAGE CREATED STATE NAME ATTEMPT
POD ID POD
6a35c025a350 ce4c0cafd583e Less than a second ago Exited init-sfs 0
bee147792a85b doca-hbn-service-my-dpu
```
This error indicates that there has been some collision with prior instances of the `doca-hbn-service` container, probably pre-reboot.

**Note**

This issue indicates irregularities in the time of the machine, and usually that the DPU's time pre-reboot was later than the time post-reboot. This leads to bugs in the recovery of the container infrastructure daemons. It is of utmost importance that the time of the system does not jump backwards.

**Solution**

1. Remove all YAML files from the deployment directory (`/etc/kubelet.d`).

2. Stop all pods:
1. Clear all containers:

```
sudo crictl stopp $(crictl pods | tail -n +2 | awk '{ print $1 }')
```

2. Make sure the system’s time is correct, and adjust it if needed:

```
date
```

3. Restart the container infrastructure daemons:

```
sudo systemctl restart kubelet.service
sudo systemctl restart containerd.service
```

4. Once the above operations are completed successfully, the container could be deployed (YAML can be copied to `/etc/kubelet.d`).

## Collecting DOCA Logs for NVIDIA Inspection

To help NVIDIA Support investigate issues customers may encounter, NVIDIA strongly recommends collecting all relevant logs using the `doca-sosreport` tool. This tool includes plugins to gather logs from various NVIDIA products and more.

On the device customers are facing issues with, run the following command with superuser privileges:

```
sudo sos report
```
This creates a tar file in the /tmp directory. When opening a support ticket for NVIDIA Support, make sure to upload this tar file.

If there is private information that you wish not to include in the tar file, extract the file and edit or remove any sensitive information, then create a new tar package.

**Info**

For more options on running the tool, refer to the tool's readme.

---

**NVIDIA BlueField Reset and Reboot Procedures**

**Contents:**

**BlueField System Reboot**

This section describes the necessary operations to load new NIC firmware, following NVIDIA® BlueField® NIC firmware update. This procedure deprecates the need for full server power cycle.

The following steps are executed in the BlueField OS:

1. Issue a query command to ascertain whether BlueField system reboot is supported by your environment:

   ```bash
   mlfwreset -d 03:00.0 q
   ```

   If the output includes the following lines, proceed to step 2:

   ```text
   3: Driver restart and PCI reset -Supported (default)
   ...```

---
2. Issue a BlueField system reboot:

```
mlxfwreset -d 03:00.0 -y -l 3 --sync 1 r
```

**BlueField System-level Reset**

This section describes the way to perform system-level reset (SLR) which is necessary for firmware configuration changes to take effect.

- **SLR for BlueField running in DPU mode**
- **SLR for BlueField running in NIC mode**
- **SLR for BlueField running in DPU mode on hosts with separate power control** (special use case)

**System-level Reset for BlueField in DPU Mode**

The following is the high-level flow of the procedure:

1. Graceful shutdown of BlueField Arm cores.
2. Query BlueField state to affirm shutdown reached.

### Note

If it says Not Supported instead, then proceed to the instructions under section "BlueField System-level Reset".

### Info
3. Warm reboot the server.

Step by step process:

**Info**

Some of the following steps can be performed using different methods, depending on resource availability and support in the user's environment.

1. Graceful shutdown of BlueField Arm cores.

**Info**

This operation is expected to finish within 15 seconds.

Possible methods:

- From the BlueField OS:

  
  ```
  shutdown -h now
  ```

  Or:
From the host OS:

```
mlxfwreset -d /dev/mst/mt*pciconf0 -l 1 -t 4 --sync 0 r
```

- **Info**
  
  Not relevant when the BlueField is operating in Zero-Trust Mode.

- **From the host OS:**

```
mlxfwreset -d <mst-device> -l 1 -t 4 r
```

- **Using the BlueField BMC:**

```
ipmitool -C 17 -I lanplus -H <bmc_ip> -U root -P <password> power soft
```

Or using Redfish (BlueField-3 and above):

```
```

2. Query BlueField state.

Possible methods:

- From the host OS:

- **Info**
Not relevant when the BlueField is operating in Zero-Trust Mode.

```bash
echo DISPLAY_LEVEL 2 > /dev/rshim0/misc
cat /dev/rshim0/misc
```

Expected output:

```
INFO[BL31]: System Off
```

- Utilizing the BlueField BMC:

```bash
ipmitool -C 17 -I lanplus -H <bmc_ip> -U root -P <password> raw 0x32 0xA3
```

Expected output: 06.

3. Warm reboot the server from the host OS:

```bash
mlxfwreset -d <mst-device> -l 4 r
```

**Note**

If multiple BlueField devices are present in the host, this command must run only once. In this case, the MST device can be of any of the BlueFields for which the reset is necessary and participated in step 1.
Or:

```
reboot
```

**Note**

For external hosts which do not toggle PERST# in their standard reboot command, use the `mlxfwreset` option.

---

**System-level Reset for BlueField in NIC Mode**

Perform warm reboot of the host OS:

```
mlxfwreset -d <mst-device> -I 4 r
```

Or:

```
reboot
```

**Note**

For external hosts which do not toggle PERST# in their standard reboot command, use the `mlxfwreset` option.
System-level Reset for Host with Separate Power Control

This procedure is a special use case relevant only to host platforms with separate power control for the PCIe slot and CPUs, in which the BlueField (running in DPU mode) is provided power while host OS/CPUs may be in shutdown or similar standby state (this allows the BlueField device to be operational while the host CPU is in shutdown/standby state).

The following is the high-level flow of the procedure:

1. Graceful shutdown of host OS or similar CPU standby.
2. Graceful shutdown of BlueField Arm cores.
3. Query BlueField state to affirm shutdown reached.
4. Full BlueField Reset
5. Query BlueField state to affirm operational state reached

Info

In systems with multiple BlueField networking platforms, repeat steps 1 through 5 for all devices before proceeding.

6. Power on the server.

Step by step process:
Some of the following steps can be performed using different methods, depending on resource availability and support in the user's environment.

1. Graceful shutdown of host OS by any means preferable.

2. Graceful shutdown of BlueField Arm cores.

- From the BlueField OS:
  
  ```
  shutdown -h now
  ```

- Utilizing the BlueField BMC:
  
  - Using IPMI:
    
    ```
    ipmitool -C 17 -I lanplus -H <bmc_ip> -U root -P <password> power soft
    ```

  - Using Redfish (for BlueField-3 and above):
    
    ```
    ```

3. Query the BlueField's state utilizing the BlueField BMC:
4. Perform BlueField hard reset utilizing the BlueField BMC:

   - **Info**
     
     This step takes up to 2 minutes to complete.

   - **Using IPMI:**
     
     ```bash
     ipmitool -C 17 -I lanplus -H <bmc_ip> -U root -P <password> power cycle
     ```

   - **Using Redfish (for BlueField-3 and above):**
     
     ```bash
     ```

5. Query BlueField operational state utilizing the BlueField BMC:

   - **Info**
     
     At this point, the BlueField is expected to be operational.
ipmitool -C 17 -I lanplus -H <bmc_ip> -U root -P <password> raw 0x32 0xA3

Expected output: 05.

6. Power on/boot up the host OS.
NVIDIA DOCA Virtual Functions User Guide

This guide provides an overview and configuration of virtual functions for NVIDIA® BlueField® and demonstrates a use case for running the DOCA applications over x86 host.

Introduction

Single root IO virtualization (SR-IOV) is a technology that allows a physical PCIe device to present itself multiple times through the PCIe bus. This technology enables multiple virtual instances of the device with separate resources. NVIDIA adapters are able to expose virtual instances or functions (VFs) for each port individually. These virtual functions can then be provisioned separately.

Each VF can be seen as an additional device connected to the physical interface or function (PF). It shares the same resources with the PF, and its number of ports equals those of the PF.

SR-IOV is commonly used in conjunction with an SR-IOV-enabled hypervisor to provide virtual machines direct hardware access to network resources, thereby increasing its performance.

There are several benefits to running applications on the host. For example, one may want to utilize a strong and high-resource host machine, or to start DOCA integration on the host before offloading it to the BlueField DPU.

The configuration in this document allows the entire application to run on the host's memory, while utilizing the HW accelerators on BlueField.

When VFs are enabled on the host, VF representors are visible on the Arm side which can be bridged to corresponding PF representors (e.g., the uplink representor and the host representor). This allows the application to only scan traffic forwarded to the VFs as configured by the user and to behave as a simple "bump-on-the-wire". DOCA installed on the host allows access to the hardware capabilities of the BlueField DPU without
comprising features which use HW offload/steering elements embedded inside the eSwitch.

**Prerequisites**

To run all the reference applications over the host, you must install the host DOCA package. Refer to the NVIDIA DOCA Installation Guide for Linux for more information on host installation.

VFs must be configured as trusted for the hardware jump action to work as intended. The following steps configure "trusted" mode for VFs:

1. **Delete all existing VFs**
   
   1. To delete all VFs on a PF run the following on the host:

   ```
   $ echo 0 > /sys/class/net/<physical_function>/device/sriov_numvfs
   
   For example:
   ```

   ```
   $ echo 0 > /sys/class/net/ens1f0/device/sriov_numvfs
   ```

   2. **Delete all existing SFs.**

   ```
   Info
   ```

   Refer to NVIDIA BlueField DPU Scalable Function User Guide for instructions on deleting SFs.

   ```
   3. Stop the main driver on the host:
   ```
4. Before creating the VFs, set them to "trusted" mode on the device by running the following commands on the DPU side.

1. Setting VFs on port 0:

   ```
   $ mlxreg -d /dev/mst/mt41686_pciconf0 --reg_id 0xc007 --reg_len 0x40 --indexes "0x0.0:32=0x80000000" --yes --set "0x4.0:32=0x1"
   ```

2. Setting VFs on port 1:

   ```
   $ mlxreg -d /dev/mst/mt41686_pciconf0.1 --reg_id 0xc007 --reg_len 0x40 --indexes "0x0.0:32=0x80000000" --yes --set "0x4.0:32=0x1"
   ```

**Note**

These commands set trusted mode for all created VFs/SFs after their execution on the DPU.

**Note**

Setting trusted mode should be performed once per reboot.

5. Restart the main driver on the host by running the following command:

   ```
   /etc/init.d/openibd stop
   ```
VF Creation

1. Make sure mst driver is running:

```
host $ mst status
```

If it is not loaded, run:

```
host $ mst start
```

2. Enable SR-IOV. Run:

```
host $ mlxconfig -y -d /dev/mst/mt41686_pciconf0 s SRIOV_EN=1
```

3. Set number of VFs. Run:

```
host $ mlxconfig -y -d /dev/mst/mt41686_pciconf0 s NUM_OF_VFS=X
```

![Note](image)

Perform a **BlueField system reboot** for the mlxconfig settings to take effect.
For example:

```
host $ echo X > /sys/class/net/<physical_function>/device/sriov_numvfs
```

After enabling VF, the representor appears on the DPU. The function itself is seen at the x86 side.

4. To verify that the VFs have been created. Run:

```
$ lspci | grep Virtual
b1:00.3 Ethernet controller: Mellanox Technologies ConnectX Family mlx5Gen Virtual Function (rev 01)
b1:00.4 Ethernet controller: Mellanox Technologies ConnectX Family mlx5Gen Virtual Function (rev 01)
b1:01.3 Ethernet controller: Mellanox Technologies ConnectX Family mlx5Gen Virtual Function (rev 01)
```

**Note**

2 new virtual Ethernet devices are created in this example.

**Running DOCA Application on Host**

**Note**
Allocate the required number of VFs as explained previously.

Note

Allocate any other resources as specified by the application (e.g., huge pages).

The following is the CLI example for running a reference application over the host using VF:

```
doca_<app_name> -a "pci address VF0" -a "pci address VF1" -c 0xff -- [application flags]
```

The following is an example with specific PCIe addresses for the VFs:

```
doca_<app_name> -a b1:00.3 -a b1:00.4 -c 0xff -- -l 60
```

Note

By default, a DPDK application initializes all the cores of the device. This is usually unnecessary and may even cause unforeseeable issues. It is recommended to limit the number of cores, especially when using an AMD-based system, to 16 cores using the -c flag when running DPDK.

Topology Example

The following is a topology example for running the application over the host.
Configure the OVS on BlueField as follows:

```
Bridge ovsbr1
  Port ovsbr1
    Interface ovsbr1
      type: internal
  Port pf0hpf
    Interface pf0hpf
  Port pf0vf1
    Interface pf0vf1
Bridge vf_br
  Port p0
    Interface p0
  Port vf_br
    Interface vf_br
      type: internal
  Port pf0vf0
    Interface pf0vf0
```
When enabling a new VF over the host, VF representors are created on the Arm side. The first OVS bridge connects the uplink connection (p0) to the new VF representor (pf0vf0), and the second bridge connects the second VF representor (pf0vf1) to the host representors (pf0phf). On the host, the 2 PCIe addresses of the newly created function must be initialized when running the applications.

When traffic is received (e.g., from the uplink), the following occurs:

1. Traffic is received over p0.
2. Traffic is forwarded to pf0vf0.
3. Application "listens" to pf0vf0 and pf0vf1 and can, therefore, acquire the traffic from pf0vf0, inspect it, and forward to pf0vf1.
4. Traffic is forwarded from pf0vf1 to pf0hpf.

VF Creation on Adapter Card

Note

Supported only for NVIDIA® ConnectX®-6 Dx based adapter cards and higher.

The following steps are required only when running DOCA applications on an adapter card.

1. Set trust level for all VFs. Run:

```
host# mlxreg -d /dev/mst/mt4125_pciconf0 --reg_name VHCA_TRUST_LEVEL --yes --set
"all_vhca=0x1,trust_level=0x1" --indexes "vhca_id=0x0,all_vhca=0x0"
```

2. Create X VFs (X being the required number of VFs) and run the following to turn on trusted mode for the created VFs:
echo ON | tee /sys/class/net/enp1s0f0np0/device/sriov/X/trust

For example, if you are creating 2 VFs, the following commands should be used:

```
3. Create a VF representor using the following command, replace the PCIe address with the PCIe address of the created VF:

echo ON | tee /sys/class/net/enp1s0f0np0/device/sriov/0/trust
echo ON | tee /sys/class/net/enp1s0f0np0/device/sriov/1/trust
```

echo 0000:17:00.2 > /sys/bus/pci/drivers/mlx5_core/unbind
echo 0000:17:00.2 > /sys/bus/pci/drivers/mlx5_core/bind