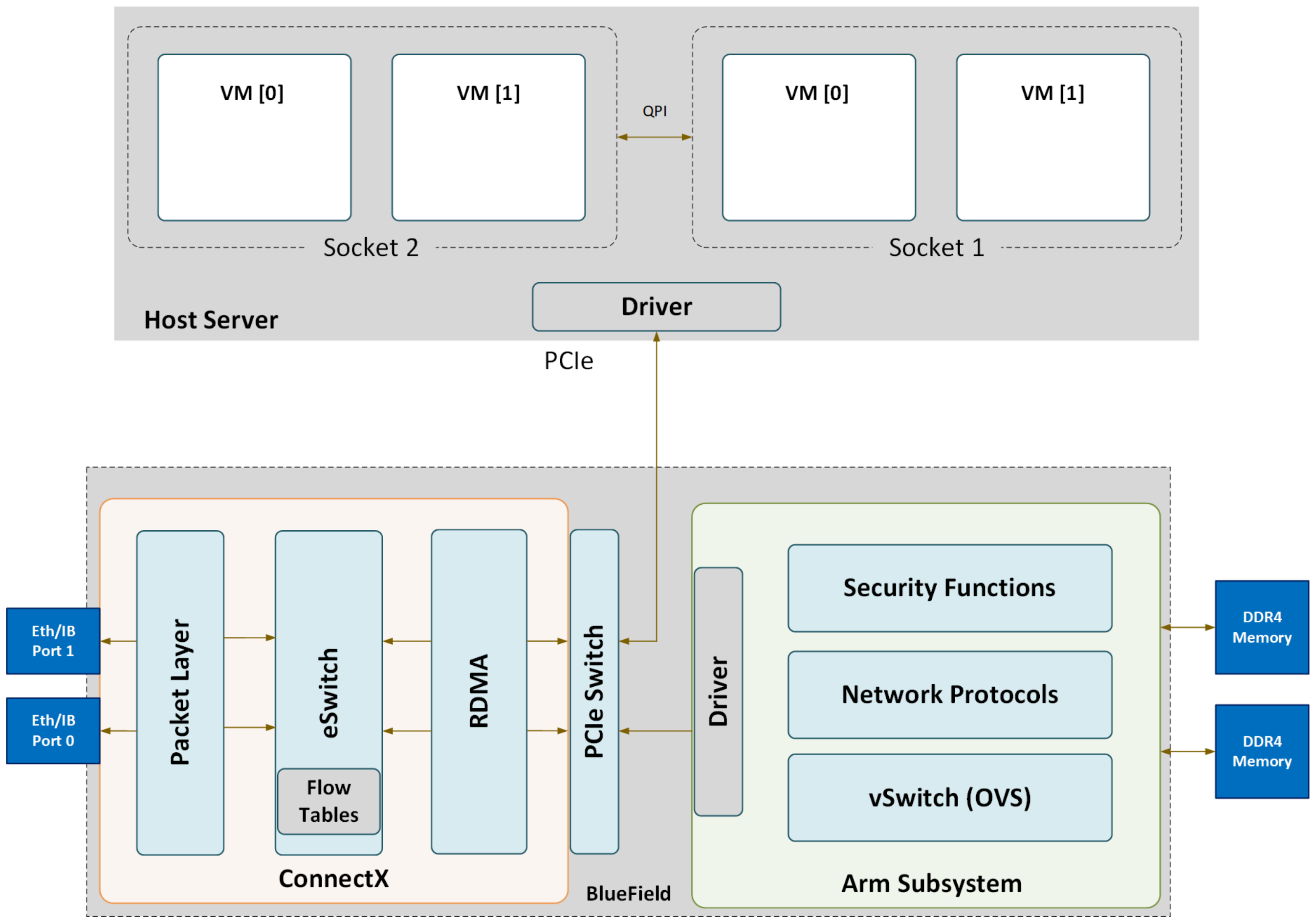
BlueField Operation

Table of contents

The NVIDIA® BlueField® networking platform family delivers the flexibility to accelerate a range of applications while leveraging ConnectX-based network controllers hardware-based offloads with unmatched scalability, performance, and efficiency.

Functional Diagram

The following is a functional diagram of the NVIDIA® BlueField®-2 DPU.



For each network port of the BlueField networking platform (DPU or SuperNIC), there are 2 physical PCIe networking functions exposed:

- To the embedded Arm subsystem
- To the host over PCIe

Note

Different functions have different default grace period values during which functions can recover from/handle a single fatal error:

- ECPFs have a graceful period of 3 minutes
- PFs have a graceful period of 1 minute
- VFs/SFs have a graceful period of 30 seconds

The mlx5 drivers and their corresponding software stacks must be loaded on both hosts (Arm and the host server). The OS running on each one of the hosts would probe the drivers. BlueField-2 network interfaces are compatible with NVIDIA® ConnectX®-6 and higher. BlueField-3 network interfaces are compatible with ConnectX-7 and higher.

The same network drivers are used both for BlueField and the ConnectX NIC family.

Modes of Operation

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

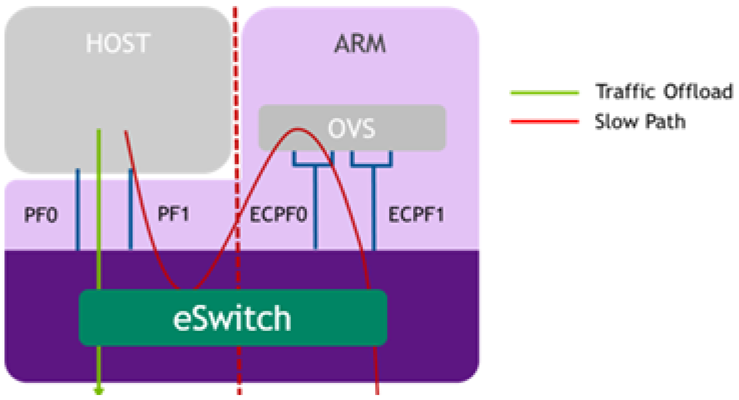
DPU Mode

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 – t he 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 .
   - If any --disable_* flags are used, users must perform BlueField system-level reset.

Disabling Zero-trust Mode

To disable host restriction:

1. Set the mode to privileged. Run:

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

2. Applying configuration:

    - I f host restriction had not been applied using any `--disable_*` flags, users must perform BlueField system reboot .
    - If host restriction had been applied using any `--disable_*` flags, users must perform BlueField system-level reset.

NIC Mode

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

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:

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

The output should have `INTERNAL_CPU_MODEL= EMBBEDDED_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:

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

2. Perform BlueField system-level reset for the `mlxconfig` settings to take effect.

Disabling NIC Mode on BlueField-3 from Linux

To return to DPU mode from NIC mode:

1. Run the following on the host:

```
host> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s INTERNAL_CPU_OFFLOAD_ENGINE=0
```

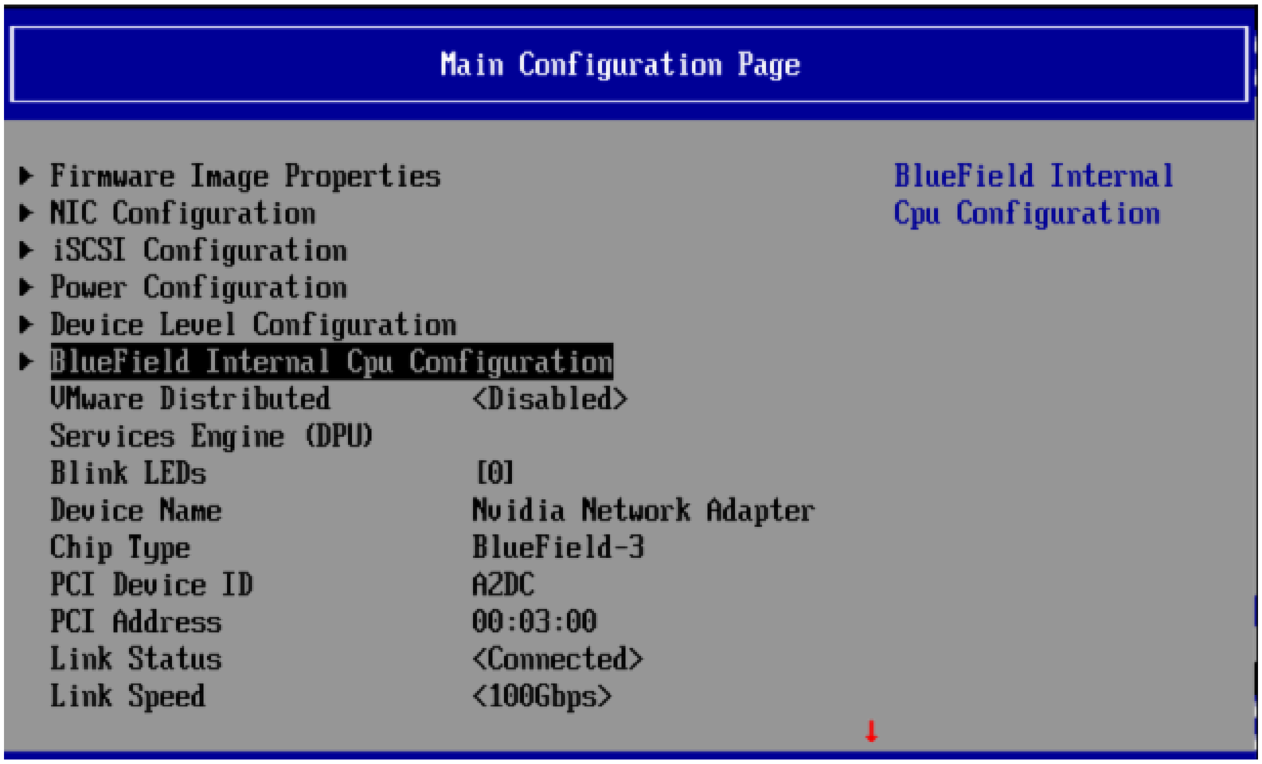
2. Perform BlueField system-level reset for the `mlxconfig` settings to take effect.

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

Info

The screenshots in this section are examples only and may vary depending on the vendor of your specific host.

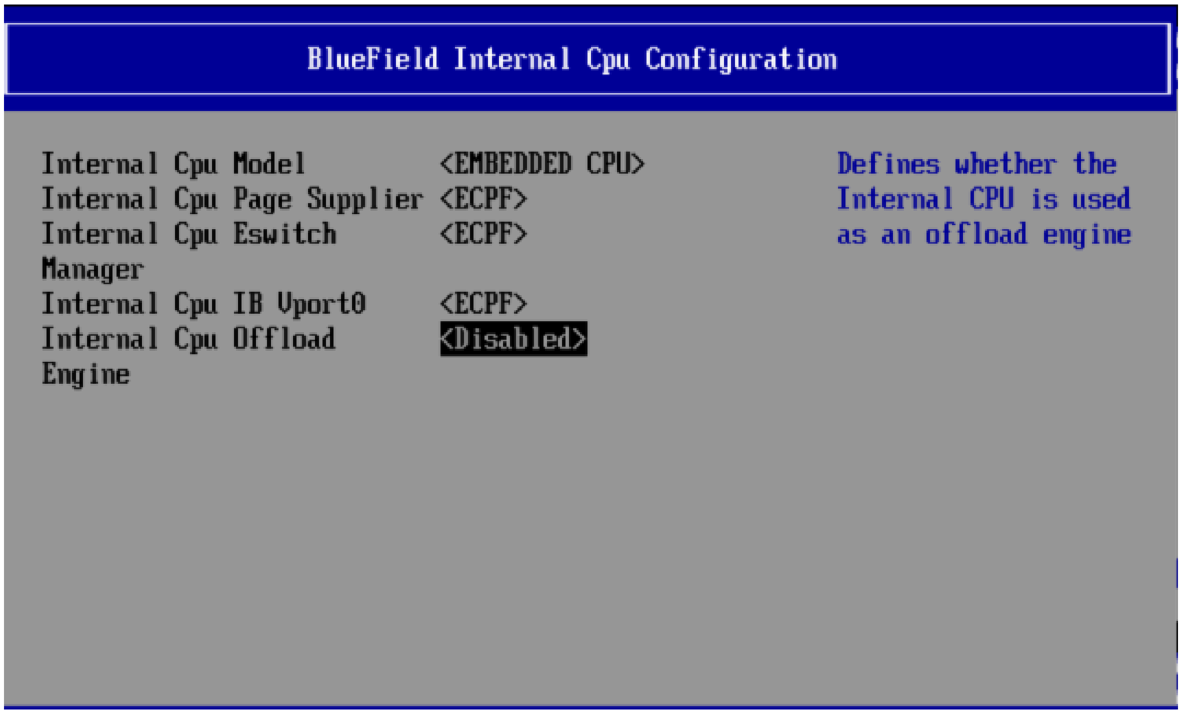
1. Select the network device that presents the uplink (i.e., select the device with the uplink MAC address).
2. Select "BlueField Internal Cpu Configuration".



```
Main Configuration Page

▶ Firmware Image Properties                              BlueField Internal
▶ NIC Configuration                                      Cpu Configuration
▶ iSCSI Configuration
▶ Power Configuration
▶ Device Level Configuration
▶ BlueField Internal Cpu Configuration
  VMware Distributed           <Disabled>
  Services Engine (DPU)
  Blink LEDs                   [0]
  Device Name                  Nvidia Network Adapter
  Chip Type                    BlueField-3
  PCI Device ID                A2DC
  PCI Address                  00:03:00
  Link Status                  <Connected>
  Link Speed                   <100Gbps>
```

- To enable NIC mode, set "Internal Cpu Offload Engine" to "Disabled".
- To switch back to DPU mode, set "Internal Cpu Offload Engine" to "Enabled".

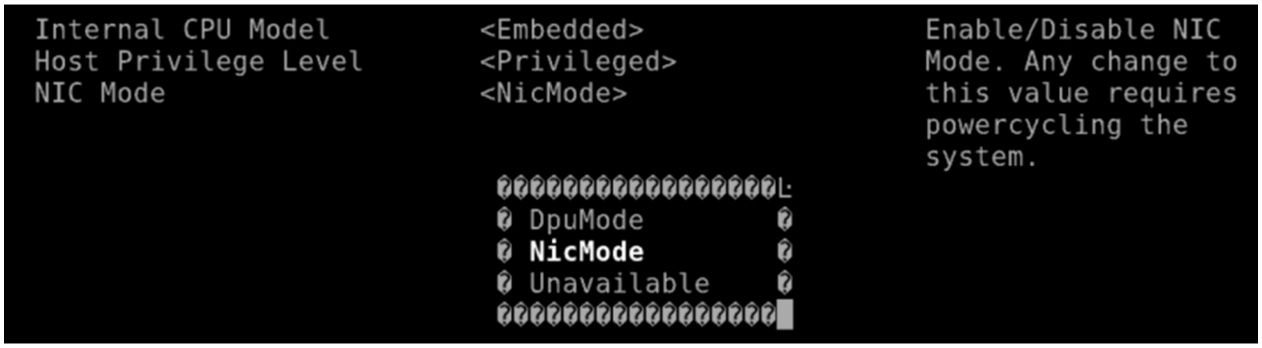


```
BlueField Internal Cpu Configuration

Internal Cpu Model            <EMBEDDED CPU>          Defines whether the
Internal Cpu Page Supplier    <ECPF>                  Internal CPU is used
Internal Cpu Eswitch          <ECPF>                  as an offload engine
Manager
Internal Cpu IB Vport0        <ECPF>
Internal Cpu Offload          <Disabled>
Engine
```

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.



```
Internal CPU Model          <Embedded>                Enable/Disable NIC
Host Privilege Level        <Privileged>              Mode. Any change to
NIC Mode                    <NicMode>                 this value requires
                                                      powercycling the
                                                      system.
                             ��������������������L
                             � DpuMode          �
                             � NicMode          �
                             � Unavailable      �
                             ��������������������
```

> **Info**
>
> Configuring `Unavailable` is inapplicable.

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 BlueField system-level reset to change to NIC mode.

Configuring NIC Mode on BlueField-3 Using Redfish

Run the following from the BlueField BMC:

1. Get the current BIOS attributes:

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

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

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

Info

To revert back to DPU mode, run:

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

3. Verify that the BMC has registered the new settings:

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

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
GET https://<bmc_ip>/redfish/v1/Systems/Bluefield/Bios
```

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 t o 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 \
INTERNAL_CPU_PAGE_SUPPLIER=0 \
INTERNAL_CPU_ESWITCH_MANAGER=0 \
INTERNAL_CPU_IB_VPORT0=0 \
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.

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)

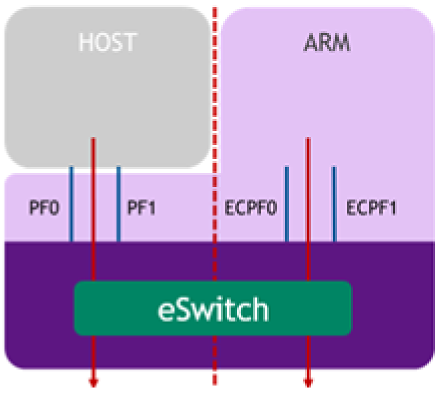
⚠ **Warning**

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
```

Kernel Representors Model

Note

This model is only applicable when the NVIDIA® BlueField® networking platform (DPUs or SuperNIC) is operating in DPU mode.

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

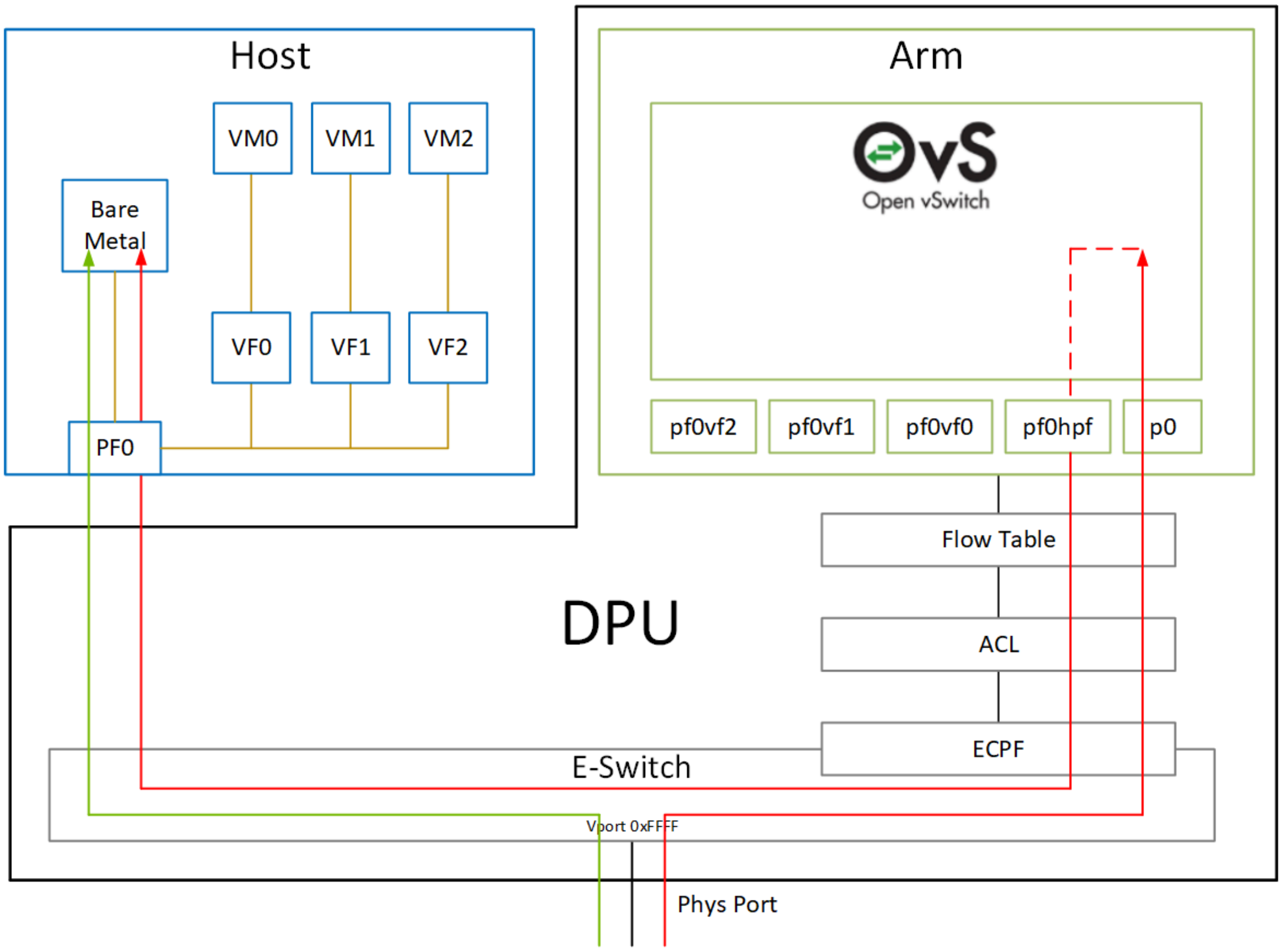
1. Serve as the tunnel to pass traffic for the virtual switch or application running on the Arm cores to the relevant PF or VF on the host side.
2. Serve as the channel to configure the embedded switch with rules to the corresponding represented function.

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

When operating in DPU mode, we see 2 representors for each one of the BlueField'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>vf<function_number>

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



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.

Multi-host

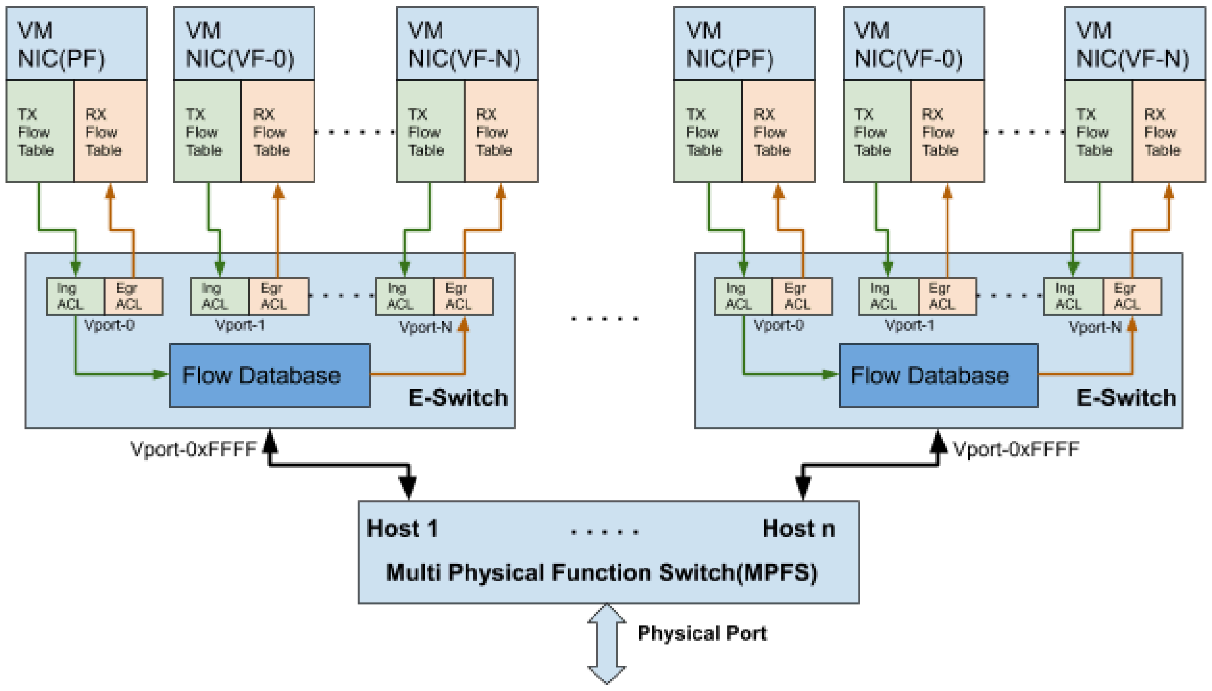
Note

This is only applicable to NVIDIA® BlueField® networking platforms (DPU or SuperNIC) running on multi-host model.

Note

All hosts in multi-host configurations must be of the same type (e.g., all x86 or all Arm); a mix of different types is not supported.

In multi-host mode, each host interface can be divided into up to 4 independent PCIe interfaces. All interfaces would share the same physical port, and are managed by the same multi-physical function switch (MPFS). Each host would have its own e-switch and would control its own traffic.



Representors

Similar to Kernel Representors Model, each host here has an uplink representor, PF representor, and VF representors (if SR-IOV is enabled). There are 8 sets of representors (uplink/PF; see example code). For each host to work with OVS offload, the corresponding representors must be added to the OVS bridge.

```
139: p0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq master ovs-system state UP
group default qlen 1000
    link/ether 0c:42:a1:70:1d:b2 brd ff:ff:ff:ff:ff:ff
140: p1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen
1000
    link/ether 0c:42:a1:70:1d:b3 brd ff:ff:ff:ff:ff:ff
141: p2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq master ovs-system state UP
group default qlen 1000
    link/ether 0c:42:a1:70:1d:b4 brd ff:ff:ff:ff:ff:ff
142: p3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen
1000
    link/ether 0c:42:a1:70:1d:b5 brd ff:ff:ff:ff:ff:ff
143: p4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen
1000
    link/ether 0c:42:a1:70:1d:b6 brd ff:ff:ff:ff:ff:ff
144: p5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen
1000
```

```
    link/ether 0c:42:a1:70:1d:b7 brd ff:ff:ff:ff:ff:ff
145: p6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 0c:42:a1:70:1d:b8 brd ff:ff:ff:ff:ff:ff
146: p7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 0c:42:a1:70:1d:b9 brd ff:ff:ff:ff:ff:ff
147: pf0hpf: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq master ovs-system state UP group default qlen 1000
    link/ether 86:c5:8a:b7:7c:84 brd ff:ff:ff:ff:ff:ff
148: pf1hpf: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 6e:ea:1b:84:88:49 brd ff:ff:ff:ff:ff:ff
149: pf2hpf: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 92:ec:99:cb:d7:23 brd ff:ff:ff:ff:ff:ff
150: pf3hpf: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 0e:0d:8e:03:2e:27 brd ff:ff:ff:ff:ff:ff
151: pf4hpf: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 5e:42:af:05:67:93 brd ff:ff:ff:ff:ff:ff
152: pf5hpf: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 96:e4:69:4c:b7:7f brd ff:ff:ff:ff:ff:ff
153: pf6hpf: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 5e:67:33:c0:35:05 brd ff:ff:ff:ff:ff:ff
154: pf7hpf: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 12:29:7d:56:07:3e brd ff:ff:ff:ff:ff:ff
```

The following is an example of adding all representors to OVS:

```
    Bridge armBr-3
        Port armBr-3
            Interface armBr-3
                type: internal
        Port p3
            Interface p3
        Port pf3hpf
            Interface pf3hpf
```

```
Bridge armBr-2
    Port p2
        Interface p2
    Port pf2hpf
        Interface pf2hpf
    Port armBr-2
        Interface armBr-2
            type: internal
Bridge armBr-5
    Port p5
        Interface p5
    Port pf5hpf
        Interface pf5hpf
    Port armBr-5
        Interface armBr-5
            type: internal
Bridge armBr-7
    Port pf7hpf
        Interface pf7hpf
    Port armBr-7
        Interface armBr-7
            type: internal
    Port p7
        Interface p7
Bridge armBr-0
    Port p0
        Interface p0
    Port armBr-0
        Interface armBr-0
            type: internal
    Port pf0hpf
        Interface pf0hpf
Bridge armBr-4
    Port p4
        Interface p4
    Port pf4hpf
        Interface pf4hpf
    Port armBr-4
        Interface armBr-4
            type: internal
Bridge armBr-1
    Port armBr-1
        Interface armBr-1
            type: internal
    Port p1
        Interface p1
    Port pf1hpf
        Interface pf1hpf
Bridge armBr-6
    Port armBr-6
        Interface armBr-6
            type: internal
    Port p6
        Interface p6
    Port pf6hpf
        Interface pf6hpf
ovs_version: "2.13.1"
```

For now, users can get the representor-to-host PF mapping by comparing the MAC address queried from host control on the Arm-side and PF MAC on the host-side. In the following example, the user knows p0 is the uplink representor for p6p1 as the MAC address is the same.

From Arm:

```
# cat /sys/class/net/p0/smart_nic/pf/config
MAC        : 0c:42:a1:70:1d:9a
MaxTxRate  : 0
State      : Up
```

From host:

```
# ip addr show p6p1
3: p6p1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 0c:42:a1:70:1d:9a brd ff:ff:ff:ff:ff:ff
```

The implicit mapping is as follows:

- PF0, PF1 = host controller 1
- PF2, PF3 = host controller 2

- PF4, PF5 = host controller 3
- PF6, PF7 = host controller 4

Note

The maximum SF or VF count across all hosts is limited to 488 in total. The user can divide 488 VFs/SFs to single or multiple controllers as desired.

Virtual Switch on BlueField

Note

For general information on OVS offload using ASAP^2 direct, please refer to the MLNX_OFED documentation under OVS Offload Using ASAP^2 Direct.

Note

ASAP^2 is only supported in Embedded (DPU) mode.

NVIDIA® BlueField® networking platforms (DPUs or SuperNICs) support ASAP^2 technology. It utilizes the representors mentioned in the previous section. The BlueField software package includes OVS installation which already supports ASAP^2. The virtual switch running on the Arm cores allows us to pass all the traffic to and from the host functions through the Arm cores while performing all the operations supported by OVS. ASAP^2 allows us to offload the datapath by programming the NIC embedded switch and avoiding the need to pass every packet through the Arm cores. The control plane remains the same as working with standard OVS.

OVS bridges are created by default upon first boot of the BlueField after BFB installation.

If manual configuration of the default settings for the OVS bridge is desired, run:

```
systemctl start openvswitch-switch.service
ovs-vsctl add-port ovsbr1 p0
ovs-vsctl add-port ovsbr1 pf0hpf
ovs-vsctl add-port ovsbr2 p1
```

```
ovs-vsctl add-port ovsbr2 pf1hpf
```

To verify successful bridging:

```
$ ovs-vsctl show
9f635bd1-a9fd-4f30-9bdc-b3fa21f8940a
    Bridge ovsbr2
        Port ovsbr2
            Interface ovsbr2
                type: internal
        Port p1
            Interface p1
        Port pf1sf0
            Interface en3f1pf1sf0
        Port pf1hpf
            Interface pf1hpf
    Bridge ovsbr1
        Port pf0hpf
            Interface pf0hpf
        Port p0
            Interface p0
        Port ovsbr1
            Interface ovsbr1
                type: internal
        Port pf0sf0
            Interface en3f0pf0sf0
    ovs_version: "2.14.1"
```

The host is now connected to the network.

Note

TC-offload is not supported for IPv6 fragment packets. To make IPv6 fragment packets pass through OVS, the MTU of a specific port must be set to equal to or larger than the fragmented packet size. IPv4 fragment packets can be TC-offloaded as their packet size is not checked by OVS.

Verifying Host Connection on Linux

When BlueField is connected to another BlueField on another machine, manually assign IP addresses with the same subnet to both ends of the connection.

1. Assuming the link is connected to p3p1 on the other host, run:

```
$ ifconfig p3p1 192.168.200.1/24 up
```

2. On the host to which BlueField is connected, run:

```
$ ifconfig p4p2 192.168.200.2/24 up
```

3. Have one ping the other. This is an example of the BlueField pinging the host:

```
$ ping 192.168.200.1
```

Verifying Connection from Host to BlueField

There are two SFs configured on the BlueFIeld-2 device, `enp3s0f0s0` and `enp3s0f1s0`, and their representors are part of the built-in bridge. These interfaces will get IP addresses from the DHCP server if it is present. Otherwise it is possible to configure IP address from the host. It is possible to access BlueField via the SF netdev interfaces.

For example:

1. Verify the default OVS configuration. Run:

```
# ovs-vsctl show
```

```
5668f9a6-6b93-49cf-a72a-14fd64b4c82b
    Bridge ovsbr1
        Port pf0hpf
            Interface pf0hpf
        Port ovsbr1
            Interface ovsbr1
                type: internal
        Port p0
            Interface p0
        Port en3f0pf0sf0
            Interface en3f0pf0sf0
    Bridge ovsbr2
        Port en3f1pf1sf0
            Interface en3f1pf1sf0
        Port ovsbr2
            Interface ovsbr2
                type: internal
        Port pf1hpf
            Interface pf1hpf
        Port p1
            Interface p1
    ovs_version: "2.14.1"
```

2. Verify whether the SF netdev received an IP address from the DHCP server. If not, assign a static IP. Run:

```
# ifconfig enp3s0f0s0
enp3s0f0s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.200.125  netmask 255.255.255.0  broadcast 192.168.200.255
        inet6 fe80::8e:bcff:fe36:19bc  prefixlen 64  scopeid 0x20<link>
        ether 02:8e:bc:36:19:bc  txqueuelen 1000  (Ethernet)
        RX packets 3730  bytes 1217558 (1.1 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 22  bytes 2220 (2.1 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

3. Verify the connection of the configured IP address. Run:

```
# ping 192.168.200.25 -c 5
```

```
PING 192.168.200.25 (192.168.200.25) 56(84) bytes of data.
64 bytes from 192.168.200.25: icmp_seq=1 ttl=64 time=0.228 ms
64 bytes from 192.168.200.25: icmp_seq=2 ttl=64 time=0.175 ms
64 bytes from 192.168.200.25: icmp_seq=3 ttl=64 time=0.232 ms
64 bytes from 192.168.200.25: icmp_seq=4 ttl=64 time=0.174 ms
64 bytes from 192.168.200.25: icmp_seq=5 ttl=64 time=0.168 ms

--- 192.168.200.25 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 91ms
rtt min/avg/max/mdev = 0.168/0.195/0.232/0.031 ms
```

Verifying Host Connection on Windows

Set IP address on the Windows side for the RShim or Physical network adapter, please run the following command in Command Prompt:

```
PS C:\Users\Administrator> New-NetIPAddress -InterfaceAlias "Ethernet 16" -IPAddress "192.168.100.1"
-PrefixLength 22
```

To get the interface name, please run the following command in Command Prompt:

```
PS C:\Users\Administrator> Get-NetAdapter
```

Output should give us the interface name that matches the description (e.g. NVIDIA BlueField Management Network Adapter).

```
Ethernet 2              NVIDIA ConnectX-4 Lx Ethernet Adapter         6 Not Present  24-8A-07-0D-E8-1D
Ethernet 6              NVIDIA ConnectX-4 Lx Ethernet Ad...#2        23 Not Present  24-8A-07-0D-E8-1C
Ethernet 16             NVIDIA BlueField Management Netw...#2        15 Up              CA-FE-01-CA-
FE-02
```

Once IP address is set, Have one ping the other.

```
C:\Windows\system32>ping 192.168.100.2
```

```
Pinging 192.168.100.2 with 32 bytes of data:
Reply from 192.168.100.2: bytes=32 time=148ms TTL=64
Reply from 192.168.100.2: bytes=32 time=152ms TTL=64
Reply from 192.168.100.2: bytes=32 time=158ms TTL=64
Reply from 192.168.100.2: bytes=32 time=158ms TTL=64
```

Enabling OVS HW Offloading

OVS HW offloading is set by default by the /sbin/mlnx_bf_configure script upon first boot after installation.

1. Enable TC offload on the relevant interfaces. Run:

   ```
   $ ethtool -K <PF> hw-tc-offload on
   ```

2. Enable the HW offload: run the following commands (after enabling the HW offload):

   ```
   $ ovs-vsctl set Open_vSwitch . Other_config:hw-offload=true
   ```

3. Restarting OVS is required for the configuration to apply:
   - For Ubuntu:

     ```
     $ systemctl restart openvswitch-switch
     ```

   - For CentOS/RHEL:

     ```
     $ systemctl restart openvswitch
     ```

To show OVS configuration:

```
$ ovs-dpctl show
system@ovs-system:
  lookups: hit:0 missed:0 lost:0
  flows: 0
  masks: hit:0 total:0 hit/pkt:0.00
  port 0: ovs-system (internal)
  port 1: armbr1 (internal)
  port 2: p0
  port 3: pf0hpf
  port 4: pf0vf0
  port 5: pf0vf1
  port 6: pf0vf2
```

At this point OVS would automatically try to offload all the rules.

To see all the rules that are added to the OVS datapath:

```
$ ovs-appctl dpctl/dump-flows
```

To see the rules that are offloaded to the HW:

```
$ ovs-appctl dpctl/dump-flows type=offloaded
```

Enabling OVS-DPDK Hardware Offload

1. Remove previously configured OVS bridges. Run:

   ```
   ovs-vsctl del-br <bridge-name>
   ```

   Issue the command `ovs-vsctl show` to see already configured OVS bridges.

2. Enable the Open vSwitch service. Run:

```
systemctl start openvswitch
```

3. Configure huge pages:

```
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

4. Configure DPDK socket memory and limit. Run:

```
# ovs-vsctl set Open_vSwitch . other_config:dpdk-socket-limit=2048
# ovs-vsctl set Open_vSwitch . other_config:dpdk-socket-mem=2048
```

5. Enable hardware offload (disabled by default). Run:

```
ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-init=true
ovs-vsctl --no-wait set Open_vSwitch . other_config:hw-offload=true
```

6. Configure the DPDK whitelist. Run:

```
ovs-vsctl set Open_vSwitch . other_config:dpdk-extra="-a 0000:03:00.0,representor=
[0,65535],dv_flow_en=1,dv_xmeta_en=1,sys_mem_en=1"
```

7. Create OVS-DPDK bridge. Run:

```
ovs-vsctl add-br br0-ovs -- set Bridge br0-ovs datapath_type=netdev -- br-set-external-id br0-ovs
bridge-id br0-ovs -- set bridge br0-ovs fail-mode=standalone
```

8. Add PF to OVS. Run:

```
ovs-vsctl add-port br0-ovs p0 -- set Interface p0  type=dpdk options:dpdk-devargs=0000:03:00.0
```

9. Add representor to OVS. Run:

```
ovs-vsctl add-port br0-ovs pf0vf0 -- set Interface pf0vf0 type=dpdk options:dpdk-
devargs=0000:03:00.0,representor=[0]
ovs-vsctl add-port br0-ovs pf0hpf -- set Interface pf0hpf type=dpdk options:dpdk-
devargs=0000:03:00.0,representor=[65535]
```

10. Restart the Open vSwitch service. This step is required for HW offload changes to take effect.
    - For CentOS, run:

    ```
    systemctl restart openvswitch
    ```

    - For Debian/Ubuntu, run:

    ```
    systemctl restart openvswitch-switch
    ```

For a reference setup configuration for BlueField-2 devices, refer to the article "Configuring OVS-DPDK Offload with BlueField-2".

Configuring DPDK and Running TestPMD

1. Configure hugepages. Run:

```
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

2. Run testpmd.

- For Ubuntu/Debian:

  ```
  env LD_LIBRARY_PATH=/opt/mellanox/dpdk/lib/aarch64-linux-gnu
  /opt/mellanox/dpdk/bin/dpdk-testpmd -a 03:00.0,representor=[0,65535] --socket-
  mem=1024 -- --total-num-mbufs=131000 -i
  ```

- For CentOS:

  ```
  env LD_LIBRARY_PATH=/opt/mellanox/dpdk/lib64/ /opt/mellanox/dpdk/bin/dpdk-testpmd
  -a 03:00.0,representor=[0,65535] --socket-mem=1024 -- --total-num-mbufs=131000 -i
  ```

For a detailed procedure with port display, refer to the article "Configuring DPDK and Running testpmd on BlueField-2".

Flow Statistics and Aging

The aging timeout of OVS is given in milliseconds and can be configured by running the following command:

```
$ ovs-vsctl set Open_vSwitch . other_config:max-idle=30000
```

Connection Tracking Offload

This feature enables tracking connections and storing information about the state of these connections. When used with OVS, BlueField can offload connection tracking, so that traffic of established connections bypasses the kernel and goes directly to hardware.

Both source NAT (SNAT) and destination NAT (DNAT) are supported with connection tracking offload.

Configuring Connection Tracking Offload

This section provides an example of configuring OVS to offload all IP connections of host PF0.

1. Enable OVS HW offloading.
2. Create OVS connection tracking bridge. Run:

```
$ ovs-vsctl add-br ctBr
```

3. Add p0 and pf0hpf to the bridge. Run:

```
$ ovs-vsctl add-port ctBr p0
$ ovs-vsctl add-port ctBr pf0hpf
```

4. Configure ARP packets to behave normally. Packets which do not comply are routed to table1. Run:

```
$ ovs-ofctl add-flow ctBr "table=0,arp,action=normal"
$ ovs-ofctl add-flow ctBr "table=0,ip,ct_state=-trk,action=ct(table=1)"
```

5. Configure RoCEv2 packets to behave normally. RoCEv2 packets follow UDP port 4791 and a different source port in each direction of the connection. RoCE traffic is not supported by CT. In order to run RoCE from the host add the following line before `ovs-ofctl add-flow ctBr "table=0,ip,ct_state=-trk,action=ct(table=1)"`:

```
$ ovs-ofctl add-flow ctBr table=0,udp,tp_dst=4791,action=normal
```

   This rule allows RoCEv2 UDP packets to skip connection tracking rules.

6. Configure the new established flows to be admitted to the connection tracking bridge and to then behave normally. Run:

```
$ ovs-ofctl add-flow ctBr "table=1,priority=1,ip,ct_state=+trk+new,action=ct(commit),normal"
```

7. Set already established flows to behave normally. Run:

```
$ ovs-ofctl add-flow ctBr "table=1,priority=1,ip,ct_state=+trk+est,action=normal"
```

Connection Tracking With NAT

This section provides an example of configuring OVS to offload all IP connections of host PF0, and performing source network address translation (SNAT). The server host sends traffic via source IP from 2.2.2.1 to 1.1.1.2 on another host. Arm performs SNAT and changes the source IP to 1.1.1.16. Note that static ARP or route table must be configured to find that route.

1. Configure untracked IP packets to do nat. Run:

```
ovs-ofctl add-flow ctBr "table=0,ip,ct_state=-trk,action=ct(table=1,nat)"
```

2. Configure new established flows to do SNAT, and change source IP to 1.1.1.16. Run:

```
ovs-ofctl add-flow ctBr
"table=1,in_port=pf0hpf,ip,ct_state=+trk+new,action=ct(commit,nat(src=1.1.1.16)), p0"
```

3. Configure already established flows act normal. Run:

```
ovs-ofctl add-flow ctBr "table=1,ip,ct_state=+trk+est,action=normal"
```

Conntrack shows the connection with SNAT applied. Run `conntrack -L` for Ubuntu 22.04 kernel or `cat /proc/net/nf_conntrack` for older kernel versions. Example output:

```
ipv4     2 tcp      6 src=2.2.2.1 dst=1.1.1.2 sport=34541 dport=5001 src=1.1.1.2 dst=1.1.1.16
```

```
sport=5001 dport=34541 [OFFLOAD] mark=0 zone=1 use=3
```

Querying Connection Tracking Offload Status

Start traffic on PF0 from the server host (e.g., iperf) with an external network. Note that only established connections can be offloaded. TCP should have already finished the handshake, UDP should have gotten the reply.

Note

ICMP is not currently supported.

To check if specific connections are offloaded from Arm, run `conntrack -L` for Ubuntu 22.04 kernel or `cat /proc/net/nf_conntrack` for older kernel versions.

The following is example output of offloaded TCP connection:

```
ipv4     2 tcp      6 src=1.1.1.2 dst=1.1.1.3 sport=51888 dport=5001 src=1.1.1.3 dst=1.1.1.2 sport=5001
dport=51888 [HW_OFFLOAD] mark=0 zone=0 use=3
```

Performance Tune Based on Traffic Pattern

Offloaded flows (including connection tracking) are added to virtual switch FDB flow tables. FDB tables have a set of flow groups. Each flow group saves the same traffic pattern flows. For example, for connection tracking offloaded flow, TCP and UDP are different traffic patterns which end up in two different flow groups.

A flow group has a limited size to save flow entries. By default, the driver has 4 big FDB flow groups. Each of these big flow groups can save at most 4000000/(4+1)=800k different 5-tuple flow entries. For scenarios with more than 4 traffic patterns, the driver provides a module parameter (num_of_groups) to allow customization and performance tune.

ⓘ **Note**

The size of each big flow groups can be calculated according to formula: size = 4000000/(num_of_groups+1)

To change the number of big FDB flow groups, run:

```
$ echo <num_of_groups> > /sys/module/mlx5_core/parameters/num_of_groups
```

The change takes effect immediately if there is no flow inside the FDB table (no traffic running and all offloaded flows are aged out), and it can be dynamically changed without reloading the driver.

If there are residual offloaded flows when changing this parameter, then the new configuration only takes effect after all flows age out.

Connection Tracking Aging

Aside from the aging of OVS, connection tracking offload has its own aging mechanism with a default aging time of 30 seconds.

Maximum Tracked Connections

ⓘ **Note**

The maximum number for tracked offloaded connections is limited to 1M by default.

The OS has a default setting of maximum tracked connections which may be configured by running:

```
$ /sbin/sysctl -w net.netfilter.nf_conntrack_max=1000000
```

This changes the maximum tracked connections (both offloaded and non-offloaded) setting to 1 million.

The following option specifies the limit on the number of offloaded connections. For example:

```
# devlink dev param set pci/${pci_dev} name ct_max_offloaded_conns value $max cmode runtime
```

This value is set to 1 million by default from BlueFiled. Users may choose a different number by using the `devlink` command.

Note

Make sure `net.netfilter.nf_conntrack_tcp_be_liberal=1` when using connection tracking.

Offloading VLANs

OVS enables VF traffic to be tagged by the virtual switch.

For BlueField, the OVS can add VLAN tag (VLAN push) to all the packets sent by a network interface running on the host (either PF or VF) and strip the VLAN tag (VLAN pop) from the traffic going from the wire to that interface. Here we operate in Virtual Switch Tagging (VST) mode. This means that the host/VM interface is unaware of the VLAN tagging. Those rules can also be offloaded to the HW embedded switch.

To configure OVS to push/pop VLAN you need to add the tag=$TAG section for the OVS command line that adds the representor ports. So if you want to tag all the traffic of VF0 with VLAN ID 52, you should use the following command when adding its representor to the bridge:

```
$ ovs-vsctl add-port armbr1 pf0vf0 tag=52
```

Note

If the virtual port is already connected to the bridge prior to configuring VLAN, you would need to remove it first:

```
$ ovs-vsctl del-port pf0vf0
```

In this scenario all the traffic being sent by VF 0 will have the same VLAN tag. We could set a VLAN tag by flow when using the TC interface, this is explained in section "Using TC Interface to Configure Offload Rules".

VXLAN Tunneling Offload

VXLAN tunnels are created on the Arm side and attached to the OVS. VXLAN decapsulation/encapsulation behavior is similar to normal VXLAN behavior, including over `hw_offload=true`.

To allow VXLAN encapsulation, the uplink representor (`p0`) should have an MTU value at least 50 bytes greater than that of the host PF/VF. Please refer to "Configuring Uplink MTU" for more information.

Configuring VXLAN Tunnel

1. Consider `p0` to be the local VXLAN tunnel interface (or VTEP).

   Note

   To be consistent with the examples below, it is assumed that `p0` is configured with a 1.1.1.1 IPv4 address.

2. Remove `p0` from any OVS bridge.
3. Build a VXLAN tunnel over OVS arm-ovs. Run:

   ```
   ovs-vsctl add-br arm-ovs -- add-port arm-ovs vxlan11 -- set interface vxlan11 type=vxlan
   options:local_ip=1.1.1.1 options:remote_ip=1.1.1.2 options:key=100
   options:dst_port=4789
   ```

4. Connect any host representor (e.g., `pf0hpf`) for which VXLAN is desired to the same arm-ovs bridge.
5. Configure the MTU of the VTEP (`p0`) used by VXLAN to at least 50 bytes larger than the host representor's MTU.

At this point, the host is unaware of any VXLAN operations done by the BlueField's OVS. If the remote end of the VXLAN tunnel is properly set, any network traffic traversing arm-ovs undergoes VXLAN encap/decap.

Querying OVS VXLAN hw_offload Rules

Run the following:

```
ovs-appctl dpctl/dump-flows type=offloaded
```

```
in_port(2),eth(src=ae:fd:f3:31:7e:7b,dst=a2:fb:09:85:84:48),eth_type(0x0800), packets:1, bytes:98,
used:0.900s, actions:set(tunnel(tun_id=0x64,src=1.1.1.1,dst=1.1.1.2,tp_dst=4789,flags(key))),3
tunnel(tun_id=0x64,src=1.1.1.2,dst=1.1.1.1,tp_dst=4789,flags(+key)),in_port(3),eth(src=a2:fb:09:85:84:48,ds
packets:75, bytes:7350, used:0.900s, actions:2
```

Note

For the host PF, in order for VXLAN to work properly with the default 1500 MTU, follow these steps.

1. Disable host PF as the port owner from Arm (see section "Zero-trust Mode"). Run:

   ```
   $ mlxprivhost -d /dev/mst/mt41682_pciconf0 --disable_port_owner r
   ```

2. The MTU of the end points (`pf0hpf` in the example above) of the VXLAN tunnel must be smaller than the MTU of the tunnel interfaces (`p0`) to account for the size of the VXLAN headers. For example, you can set the MTU of P0 to 2000.

GRE Tunneling Offload

GRE tunnels are created on the Arm side and attached to the OVS. GRE decapsulation/encapsulation behavior is similar to normal GRE behavior, including over `hw_offload=true`.

To allow GRE encapsulation, the uplink representor (`p0`) should have an MTU value at least 50 bytes greater than that of the host PF/VF.

Please refer to "Configuring Uplink MTU" for more information.

Configuring GRE Tunnel

1. Consider `p0` to be the local GRE tunnel interface. `p0` should not be attached to any OVS bridge.

   Note

   To be consistent with the examples below, it is assumed that `p0` is configured with a 1.1.1.1 IPv4 address and that the remote end of the tunnel is 1.1.1.2.

2. Create an OVS bridge, `br0`, with a GRE tunnel interface, `gre0`. Run:

   ```
   ovs-vsctl add-port br0 gre0 -- set interface gre0 type=gre options:local_ip=1.1.1.1
   options:remote_ip=1.1.1.2 options:key=100
   ```

3. Add `pf0hpf` to `br0`.

   ```
   ovs-vsctl add-port br0 pf0hpf
   ```

4. At this point, any network traffic sent or received by the host's PF0 undergoes GRE processing inside the BlueField OS.

Querying OVS GRE hw_offload Rules

Run the following:

```
ovs-appctl dpctl/dump-flows type=offloaded
recirc_id(0),in_port(3),eth(src=50:6b:4b:2f:0b:74,dst=de:d0:a3:63:0b:30),eth_type(0x0800),ipv4(frag=no),
packets:878, bytes:122802, used:0.440s,
actions:set(tunnel(tun_id=0x64,src=1.1.1.1,dst=1.1.1.2,ttl=64,flags(key))),2
```

```
tunnel(tun_id=0x64,src=1.1.1.1,dst=1.1.1.2,flags(+key)),recirc_id(0),in_port(2),eth(src=de:d0:a3:63:0b:30,dst
packets:995, bytes:97510, used:0.440s, actions:3
```

Note

For the host PF, in order for GRE to work properly with the default 1500 MTU, follow these steps.

1. Disable host PF as the port owner from Arm (see section "Zero-trust Mode"). Run:

   ```
   $ mlxprivhost -d /dev/mst/mt41682_pciconf0 --disable_port_owner r
   ```

2. The MTU of the end points (`pf0hpf` in the example above) of the GRE tunnel must be smaller than the MTU of the tunnel interfaces (`p0`) to account for the size of the GRE headers. For example, you can set the MTU of P0 to 2000.

GENEVE Tunneling Offload

GENEVE tunnels are created on the Arm side and attached to the OVS. GENEVE decapsulation/encapsulation behavior is similar to normal GENEVE behavior, including over `hw_offload=true`.

To allow GENEVE encapsulation, the uplink representor (`p0`) must have an MTU value at least 50 bytes greater than that of the host PF/VF.

Please refer to "Configuring Uplink MTU" for more information.

Configuring GENEVE Tunnel

1. Consider `p0` to be the local GENEVE tunnel interface. `p0` should not be attached to any OVS bridge.
2. Create an OVS bridge, `br0`, with a GENEVE tunnel interface, `gnv0`. Run:

```
ovs-vsctl add-port br0 gnv0 -- set interface gnv0 type=geneve options:local_ip=1.1.1.1
options:remote_ip=1.1.1.2 options:key=100
```

3. Add `pf0hpf` to `br0`.

```
ovs-vsctl add-port br0 pf0hpf
```

4. At this point, any network traffic sent or received by the host's PF0 undergoes GENEVE processing inside the BlueField OS.

Options are supported for GENEVE. For example, you may add option `0xea55` to tunnel metadata, run:

```
ovs-ofctl add-tlv-map geneve_br "{class=0xffff,type=0x0,len=4}->tun_metadata0"
ovs-ofctl add-flow geneve_br ip,actions="set_field:0xea55->tun_metadata0",normal
```

Note

For the host PF, in order for GENEVE to work properly with the default 1500 MTU, follow these steps.

1. Disable host PF as the port owner from Arm (see section "Zero-trust Mode"). Run:

```
$ mlxprivhost -d /dev/mst/mt41682_pciconf0 --disable_port_owner r
```

2. The MTU of the end points (pf0hpf in the example above) of the GENEVE tunnel must be smaller than the MTU of the tunnel interfaces (p0) to account for the size of the GENEVE headers. For example, you can set the MTU of P0 to 2000.

Using TC Interface to Configure Offload Rules

Offloading rules can also be added directly, and not just through OVS, using the tc utility. To enable TC ingress on all the representors (i.e., uplink, PF, and VF).

```
$ tc qdisc add dev p0 ingress
$ tc qdisc add dev pf0hpf ingress
$ tc qdisc add dev pf0vf0 ingress
```

L2 Rules Example

The rule below drops all packets matching the given source and destination MAC addresses.

```
$ tc filter add dev pf0hpf protocol ip parent ffff: \
                    flower \
                            skip_sw \
                            dst_mac e4:11:22:11:4a:51 \
                            src_mac e4:11:22:11:4a:50 \
                     action drop
```

VLAN Rules Example

The following rules push VLAN ID 100 to packets sent from VF0 to the wire (and forward it through the uplink representor) and strip the VLAN when sending the packet to the VF.

```
$ tc filter add dev pf0vf0 protocol 802.1Q parent ffff: \
                        flower \
                               skip_sw \
                               dst_mac e4:11:22:11:4a:51 \
                               src_mac e4:11:22:11:4a:50 \
                        action vlan push id 100 \
                        action mirred egress redirect dev p0

$ tc filter add dev p0 protocol 802.1Q parent ffff: \
                        flower \
                              skip_sw \
                              dst_mac e4:11:22:11:4a:51 \
                              src_mac e4:11:22:11:4a:50 \
                              vlan_ethtype 0x800 \
                              vlan_id 100 \
                              vlan_prio 0 \
                       action vlan pop \
                       action mirred egress redirect dev pf0vf0
```

VXLAN Encap/Decap Example

```
$ tc filter add dev pf0vf0 protocol 0x806 parent ffff: \
                        flower \
                               skip_sw \
                               dst_mac e4:11:22:11:4a:51 \
                               src_mac e4:11:22:11:4a:50 \
                        action tunnel_key set \
                        src_ip 20.1.12.1 \
                        dst_ip 20.1.11.1 \
                        id 100 \
                        action mirred egress redirect dev vxlan100

$ tc filter add dev vxlan100 protocol 0x806 parent ffff: \
                        flower \
                              skip_sw \
                              dst_mac e4:11:22:11:4a:51 \
                              src_mac e4:11:22:11:4a:50 \
                              enc_src_ip 20.1.11.1 \
                              enc_dst_ip 20.1.12.1 \
                              enc_key_id 100 \
                              enc_dst_port 4789 \
                          action tunnel_key unset \
                          action mirred egress redirect dev pf0vf0
```

VirtIO Acceleration Through Hardware vDPA

For configuration procedure, please refer to the MLNX_OFED documentation under OVS Offload Using ASAP^2 Direct > VirtIO Acceleration through Hardware vDPA.

Configuring Uplink MTU

To configure the port MTU while operating in DPU mode, users must restrict the external host port ownership by issuing the following command on the BlueField:

```
mlxprivhost -d /dev/mst/<pciconf0 device> r --disable_port_owner
```

Server cold reboot is required for this restriction to take effect.

Once the host is restricted, the port MTU is configured by changing the MTU of the uplink representor (`p0` or `p1`).

Link Aggregation

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.

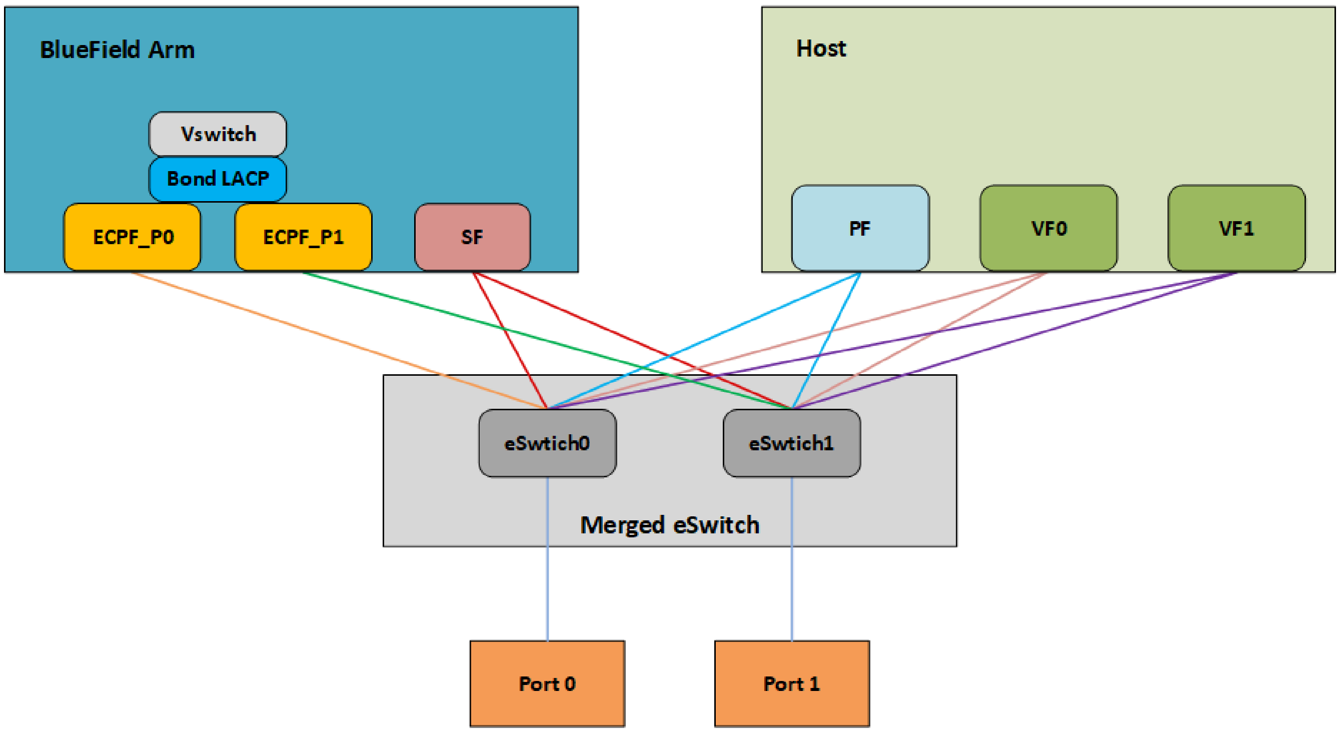
Note

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 following diagram 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:

```
LAG_HASH_MODE="no"
```

3. Perform BlueField system reboot for the mlxconfig settings to take effect.

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 BlueField system reboot for the mlxconfig settings to take effect.

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 BlueField system reboot for the `mlxconfig` settings to take effect.

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

   Note

While LAG is being configured (starting with the next step), traffic cannot pass through the physical ports.

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
#
#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) &
```

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:

```
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.

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:

       ```
       systemctl stop openibd
       ```
2. Delete the LAG OVS bridge on the Arm side. Run:

   ```
   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 BlueField system reboot for the `mlxconfig` settings to take effect.

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:

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

3. Make sure NVME emulation is disabled:

```
$ 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:

```
$ 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:

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

3. Bring the interfaces up. Run:

```
$ 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:

```
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

When performing driver reload (`openibd restart`) or reboot, you must remove bond configuration from NetworkManager, and to reapply the configurations after the driver is fully up.

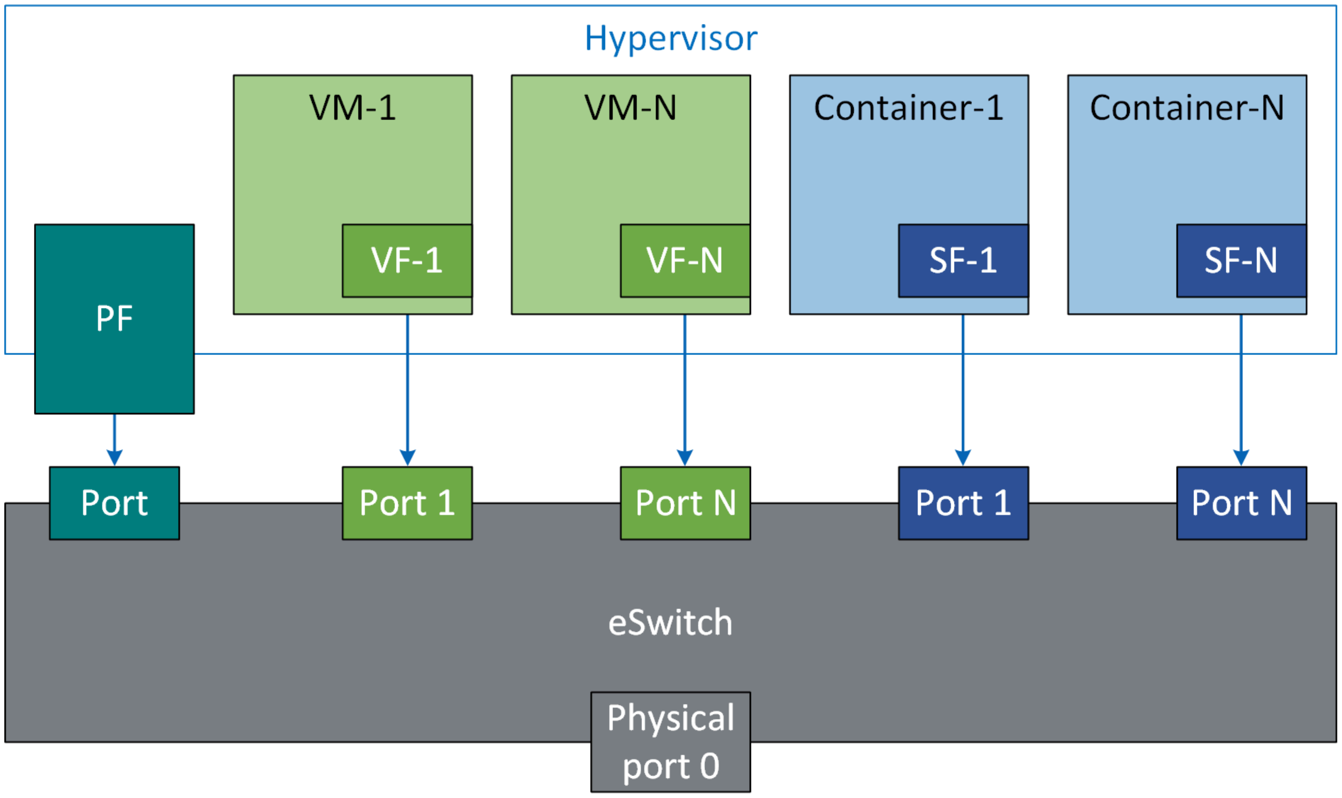
Removing LAG Configuration on Multi-host

Refer to section "Removing LAG Configuration".

Scalable Functions

A scalable function (SF) is a lightweight function that has a parent PCIe function on which it is deployed. An mlx5 SF has its own function capabilities and its own resources. This means that an SF has its own dedicated queues (txq, rxq, cq, eq) which are neither shared nor stolen from the parent PCIe function.

No special support is needed from system BIOS to use SFs. SFs co-exist with PCIe SR-IOV virtual functions. SFs do not require enabling PCIe SR-IOV.



Scalable Function Configuration

The following procedure offers a guide on using scalable functions with upstream Linux kernel.

Device Configuration

1. Make sure your firmware version supports SFs (20.30.1004 and above).
2. Enable SF support in device. Run:

```
$ 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
```

3. Perform BlueField system reboot for the `mlxconfig` settings to take effect.

Mandatory Kernel Configuration on Host

Support for Linux kernel mlx5 SFs must be enabled as it is disabled by default.

The following two Kconfig flags must be enabled.

- MLX5_ESWITCH
- MLX5_SF

Software Control and Commands

SFs use a 4-step process as follows:

- Create
- Configure
- Deploy
- Use

SFs are managed using mlxdevm tool. It is located under directory `/opt/mellanox/iproute2/sbin/mlxdevm`.

1. Display the physical (i.e. uplink) port of the PF. Run:

```
$ devlink port show
pci/0000:03:00.0/65535: type eth netdev p0 flavour physical port 0 splittable false
```

2. Add an SF. Run:

```
$ mlxdevm port add pci/0000:03:00.0 flavour pcisf pfnum 0 sfnum 88
pci/0000:03:00.0/229409: type eth netdev eth0 flavour pcisf controller 0 pfnum 0 sfnum 88
  function:
    hw_addr 00:00:00:00:00:00 state inactive opstate detached trust off
```

Note

An added SF is still not usable for the end-user application. It can only be used after configuration and activation.

Note

SF number ≥1 000 is reserved for the virtio-net controller.

When an SF is added on the external controller (e.g., BlueField) users must supply the controller number. In a single host BlueField case, there is only one controller starting with controller number 1.

Example of adding an SF for PF0 of external controller 1:

```
$ mlxdevm port add pci/0000:03:00.0 flavour pcisf pfnum 0 sfnum 88 controller 1
pci/0000:03:00.0/32768: type eth netdev eth6 flavour pcisf controller 1 pfnum 0 sfnum 88
splittable false
  function:
    hw_addr 00:00:00:00:00:00 state inactive opstate detached
```

3. Show the newly added devlink port by its port index or its representor device.

```
$ mlxdevm port show en3f0pf0sf88
pci/0000:03:00.0/229409: type eth netdev en3f0pf0sf88 flavour pcisf controller 0 pfnum 0 sfnum 88
  function:
    hw_addr 00:00:00:00:00:00 state inactive opstate detached trust off
```

Or:

```
$ mlxdevm port show pci/0000:03:00.0/229409
pci/0000:03:00.0/229409: type eth netdev en3f0pf0sf88 flavour pcisf controller 0 pfnum 0 sfnum 88
  function:
    hw_addr 00:00:00:00:00:00 state inactive opstate detached trust off
```

4. Set the MAC address of the SF. Run:

```
$ mlxdevm port function set pci/0000:03:00.0/229409 hw_addr 00:00:00:00:88:88
```

5. Set SF as trusted (optional). Run:

```
$ mlxdevm port function set pci/0000:03:00.0/229409 trust on
pci/0000:03:00.0/229409: type eth netdev en3f0pf0sf88 flavour pcisf controller 0 pfnum 0 sfnum 88
  function:
    hw_addr 00:00:00:00:88:88 state inactive opstate detached trust on
```

Note

A trusted function has additional privileges like the ability to update steering database.

6. Configure OVS. Run:

```
$ systemctl start openvswitch
$ ovs-vsctl add-br network1
$ ovs-vsctl add-port network1 ens3f0npf0sf88
$ ip link set dev ens3f0npf0sf88 up
```

7. Activate the SF. Run:

```
$ mlxdevm port function set pci/0000:03:00.0/229409 state active
```

Activating the SF results in creating an auxiliary device and initiating driver load sequence for netdevice, RDMA, and VDPA devices. Once the operational state is marked as attached, a driver is attached to this SF and device loading begins.

Note

An application interested in using the SF netdevice and rdma device must monitor the RDMA and netdevices either through udev monitor or poll the sysfs hierarchy of the SF's auxiliary device.

8. By default, SF is attached to the configuration driver `mlx5_core.sf_cfg`. Users must unbind an SF from the configuration and bind it to the `mlx5_core.sf` driver to make use of it. Run:

```
$ echo mlx5_core.sf.4 > /sys/bus/auxiliary/devices/mlx5_core.sf.4/driver/unbind
$ echo mlx5_core.sf.4 > /sys/bus/auxiliary/drivers/mlx5_core.sf/bind
```

9. View the new state of the SF. Run:

```
$ mlxdevm port show en3f0pf0sf88 -jp
{
    "port": {
        "pci/0000:03:00.0/229409": {
            "type": "eth",
            "netdev": "en3f0pf0sf88",
            "flavour": "pcisf",
            "controller": 0,
            "pfnum": 0,
            "sfnum": 88,
            "function": {
                "hw_addr": "00:00:00:00:88:88",
                "state": "active",
                "opstate": "detached",
                "trust": "on"
            }
        }
    }
}
```

10. View the auxiliary device of the SF. Run:

```
$ cat /sys/bus/auxiliary/devices/mlx5_core.sf.4/sfnum
88
```

There can be hundreds of auxiliary SF devices on the auxiliary bus. Each SF's auxiliary device contains a unique sfnum and PCI information.

11. View the parent PCI device of the SF. Run:

```
$ readlink /sys/bus/auxiliary/devices/mlx5_core.sf.1
```

```
../../../devices/pci0000:00/0000:00:00.0/0000:01:00.0/0000:02:00.0/0000:03:00.0/mlx5_core.sf.1
```

12. View the devlink instance of the SF device. Run:

```
$ devlink dev show
$ devlink dev show auxiliary/mlx5_core.sf.4
```

13. View the port and netdevice associated with the SF. Run:

```
$ devlink port show auxiliary/mlx5_core.sf.4/1
auxiliary/mlx5_core.sf.4/1: type eth netdev enp3s0f0s88 flavour virtual port 0 splittable false
```

14. View the RDMA device for the SF. Run:

```
$ rdma dev show
$ ls /sys/bus/auxiliary/devices/mlx5_core.sf.4/infiniband/
```

15. Deactivate SF. Run:

```
$ mlxdevm port function set pci/0000:03:00.0/229409 state inactive
```

    Deactivating the SF triggers driver unload in the host system. Once SF is deactivated, its operational state changes to "detached". An orchestration system should poll for the operational state to be changed to "detached" before deleting the SF. This ensures a graceful hot-unplug.

16. Delete SF. Run:

```
$ mlxdevm port del pci/0000:03:00.0/229409
```

Finally, once the state is "inactive" and the operational state is "detached" the user can safely delete the SF. For faster provisioning, a user can reconfigure and active the SF again without deletion.

RDMA Stack Support on Host and Arm System

Full RDMA stack is pre-installed on the Arm Linux system. RDMA, whether RoCE or InfiniBand, is supported on NVIDIA® BlueField® networking platforms (DPUs or SuperNICs) in the configurations listed below.

Separate Host Mode

RoCE is supported from both the host and Arm system.

InfiniBand is supported on the host system.

Embedded CPU Mode

RDMA Support on Host

To use RoCE on a host system's PCIe PF, OVS hardware offloads must be enabled on the Arm system.

RoCE is not supported by connection tracking offload. Please refer to "Configuring Connection Tracking Offload" for a workaround for it.

RDMA Support on Arm

RoCE is unsupported on the Arm system on the PCIe PF. However, RoCE is fully supported using scalable function as explained under "Scalable Functions". Scalable functions are created by default, allowing RoCE traffic without further configuration.

InfiniBand is supported on the Arm system on the PCIe PF in this mode.

Controlling Host PF and VF Parameters

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 firmware 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:

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

Querying Configuration

To query the current configuration, run:

```
$ 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 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. `pf0hpf` and `pf1hpf`) 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:

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

To reactivate host networking PFs:

- For single-port BlueFields, run:

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

- For dual-port BlueFields, run:

  ```
  mlxconfig -d /dev/mst/mt41686_pciconf0 s NUM_OF_PF=2
  ```

Note

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

Note

Perform BlueField system reboot for the `mlxconfig` settings to take effect.

DPDK on BlueField

Please refer to "NVIDIA BlueField Board Support Package" in the DPDK documentation.

BlueField SNAP

NVIDIA® BlueField® SNAP (Software-defined Network Accelerated Processing) technology enables hardware-accelerated virtualization of NVMe storage. BlueField SNAP presents networked storage as a local NVMe SSD, emulating an NVMe drive on the PCIe bus. The host OS/Hypervisor makes use of its standard NVMe-driver unaware that the communication is terminated, not by a physical drive, but by the BlueField SNAP. Any logic may be applied to the data via the BlueField SNAP framework and transmitted over the network, on either Ethernet or InfiniBand protocol, to a storage target.

BlueField SNAP combines unique hardware-accelerated storage virtualization with the advanced networking and programmability capabilities of NVIDIA® BlueField® networking platforms (DPU or SuperNIC). BlueField SNAP together with the DPU enable a world of applications addressing storage and networking efficiency and performance.

To enable BlueField SNAP, please refer to the NVIDIA BlueField-3 SNAP for NVMe and Virtio-blk documentation.

BlueField SR-IOV

The NVIDIA® BlueField® SR-IOV solution is based on asymmetric VF and enables per-ECPF and per PF control over number of VF allocation .

ECPF VFs are intended to be used in switchdev mode. Like SFs and host VFs, ECPF VFs have a representor. Representor naming for ECPF VFs start after the host VFs. For example, if the host has 32 VFs enabled, then the host VF representors are named `pf0vf0-pf0vf31`, and the Arm representors continue at `pf0vf32` onward.

To enable BlueField SR-IOV, apply the following configuration in the BlueField OS:

```
mlxconfig -d 03:00.0 -y s PF_NUM_OF_VF_VALID=1
```

Note

Once `PF_NUM_OF_VF_VALID` is set, the `NUM_OF_VFS` mlxconfig option is not relevant and the user must set `PF_NUM_OF_VF` for each host and EC function. It is recommended for the number of VFs for each ECPF and each host PF be the same.

The BlueField should now support setting asymmetric VF configuration per port.

The following are examples for configuring the number of VFs per port:

1. In the BlueField, issue the following commands to configure 32 VFs per port:

```
bf> mlxconfig -d 03:00.0 -y s PF_NUM_OF_VF=32
bf> mlxconfig -d 03:00.1 -y s PF_NUM_OF_VF=32
```

Note

The BlueField ECPF driver in the BlueField's Arm OS limits the number of VFs it supports to 32 per port.

2. In the host OS, issue the following commands to configure up to 126 VFs per port:

```
host> mlxconfig -d 03:00.0 -y s PF_NUM_OF_VF=126
host> mlxconfig -d 03:00.1 -y s PF_NUM_OF_VF=126
```

3. Perform a BlueField system reboot for the `mlxconfig` settings to take effect.

4. Create ECPF VFs:

```
echo 1 > /sys/class/net/p0/device/sriov_numvfs
```

Note

BlueField SR-IOV VFs do not support the following legacy SRIOV functionalities:

- Virtual switch tagging (VF VLAN)
- Spoof check
- VF trust
- VF rate

Compression Acceleration

NVIDIA® BlueField® networking platforms (DPUs or SuperNIC) support high-speed compression acceleration. This feature allows the host to offload multiple compression/decompression jobs to BlueField.

Compress-class operations are supported in parallel to the net, vDPA, and RegEx class operations.

Configuring Compression Acceleration

The compression application can run either from the host or Arm.

For more information, please refer to:

- The DPDK community documentation about compression
- The mlx5 support documentation

Public Key Acceleration

NVIDIA® BlueField® networking platforms (DPUs or SuperNICs) 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.

PKA Prerequisites

- The BlueField PKA software is intended for BlueField products with HW accelerated crypto capabilities. To verify whether your BlueField chip has crypto capabilities, look for CPU flags `aes`, `sha1`, and `sha2` in the BlueField OS. For example:

```
# lscpu
...
Flags: fp asimd evtstrm aes pmull sha1 sha2 crc32 cpuid
```

- BlueField bootloader must enable SMMU support to benefit from the full hardware and software capabilities. SMMU support may be enabled in UEFI menu through system configuration options.

PKA Use Cases

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 the Mellanox PKA GitHub.

IPsec Functionality

Transparent IPsec Encryption and Decryption

NVIDIA® BlueField® networking platforms (DPU or SuperNICs) can offload IPsec operations transparently from the host CPU. This means that the host does not need to be aware that network traffic is encrypted before hitting the wire or decrypted after coming off the wire. IPsec operations can be run on BlueField in software on the Arm cores or in the accelerator block.

IPsec Hardware Offload: Crypto Offload

IPsec hardware crypto offload, also known as IPsec inline offload or IPsec aware offload, enables the user to offload IPsec crypto encryption and decryption operations to the hardware, leaving the encapsulation/decapsulation task to the software.

Please refer to the MLNX_OFED documentation under Features Overview and Configuration > Ethernet Network > IPsec Crypto Offload for more information on enabling and configuring this feature.

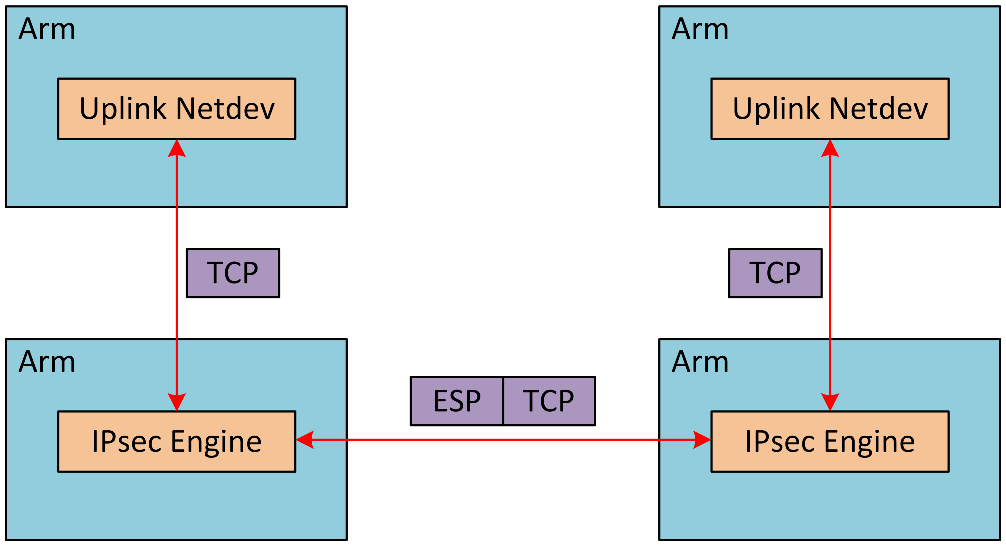
Please note that to use IPsec crypto offload with OVS, you must disable hardware offloads.

IPsec Hardware Offload: Packet Offload

Note

IPSec packet offload is only supported on Ubuntu BlueField kernel 5.15

IPsec packet offload offloads both IPsec crypto and IPsec encapsulation to the hardware. IPsec packet offload is configured on the Arm via the uplink netdev. The following figure illustrates IPsec packet offload operation in hardware.



Enabling IPsec Packet Offload

Explicitly enable IPsec packet offload on the Arm cores before setting up offload-aware IPsec tunnels .

Note

If an OVS VXLAN tunnel configuration already exists, stop `openvswitch` service prior to performing the steps below and restart the service afterwards.

Explicitly enable IPsec full offload on the Arm cores.

1. Set `IPSEC_FULL_OFFLOAD="yes"` in `/etc/mellanox/mlnx-bf.conf` .
2. Restart IB driver (rebooting also works). Run:

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

Note

If mlx-regex is running:

1. Disable mlx-regex:

   ```
   systemctl stop mlx-regex
   ```

2. Restart IB driver according to the command above.

3. Re-enable mlx-regex after the restart has finished:

   ```
   systemctl restart mlx-regex
   ```

Note

To revert IPsec full offload mode, redo the procedure from step 1, only difference is to set IPSEC_FULL_OFFLOAD="no" in /etc/mellanox/mlnx-bf.conf.

Note

To use IPsec packet packet with strongSwan, refer to section "IPsec Packet Offload strongSwan Support".

To configure IPsec rules, please follow the instructions in MLNX_OFED documentation under Features Overview and Configuration > Ethernet Network > IPsec Crypto Offload >

Configuring Security Associations for IPsec Offloads but, use "offload packet" to achieve IPsec Packet offload.

Configuring IPsec Rules with iproute2

Note

If you are working directly with the `ip xfrm` tool, you must use the `/opt/mellanox/iproute2/sbin/ip` to benefit from IPsec packet offload support.

The following example configures IPsec packet offload rules with local address 192.168.1.64 and remote address 192.168.1.65:

```
ip xfrm state add src 192.168.1.64/24 dst 192.168.1.65/24 proto esp spi 0x4834535d reqid 0x4834535d
mode transport aead 'rfc4106(gcm(aes))'
0xc57f6f084ebf8c6a71dd9a053c2e03b94c658a9bf00dd25780e73948931d10d08058a27c 128 offload
packet dev p0 dir out sel src 192.168.1.64 dst 192.168.1.65
ip xfrm state add src 192.168.1.65/24 dst 192.168.1.64/24 proto esp spi 0x2be60844 reqid 0x2be60844
mode transport aead 'rfc4106(gcm(aes))'
0xacca06b66489011d3c1c21f1a36d925cf7449d3aeaa6fe534446c3a8f8bd5f5fdc266589 128 offload
packet dev p0 dir in sel src 192.168.1.65 dst 192.168.1.64
sudo ip xfrm policy add src 192.168.1.64 dst 192.168.1.65 offload packet dev p0 dir out tmpl src
192.168.1.64/24 dst 192.168.1.65/24 proto esp reqid 0x4834535d mode transport
sudo ip xfrm policy add src 192.168.1.65 dst 192.168.1.64 offload packet dev p0 dir in tmpl src
192.168.1.65/24 dst 192.168.1.64/24 proto esp reqid 0x2be60844 mode transport
```

Note

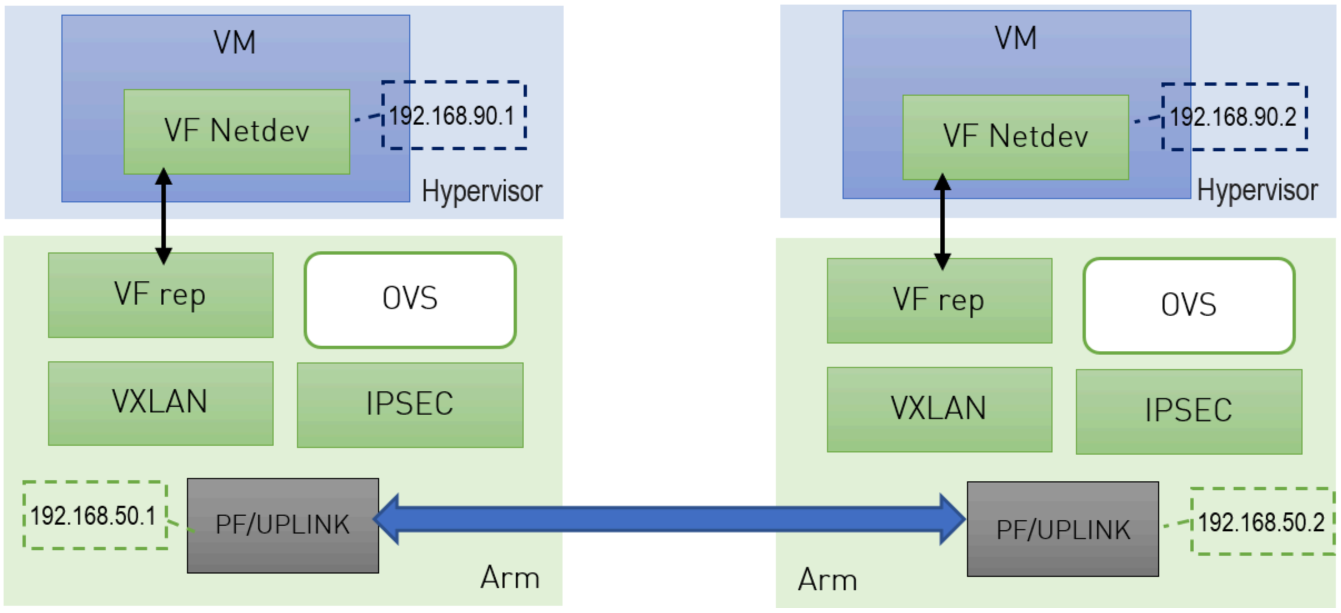
The numbers used by the `spi`, `reqid`, or `aead` algorithms are random. These same numbers are also used in the configuration of peer Arm.

Do not confuse these numbers with source and destination IPs. The connection may fail if they are not consistent.

IPsec Packet Offload strongSwan Support

BlueField supports configuring IPsec rules using strongSwan 5.9.10—appears as 5.9.10bf in the BFB which is based on upstream 5.9.10 version—which supports new fields in the swanctl.conf file.

The following figure illustrates an example with two BlueField devices , Left and Right, operating with a secured VXLAN channel .



Support for strongSwan IPsec packet HW offload requires using VXLAN together with IPSec as shown here .

1. Follow the procedure under section "Enabling IPsec Packet Offload".
2. Follow the procedure under section "VXLAN Tunneling Offload" to configure VXLAN on Arm.

Note

Make sure the MTU of the PF used by VXLAN is at least 50 bytes larger than VXLAN-REP MTU.

3. Enable tc offloading. Run:

```
ethtool -K <PF> hw-tc-offload on
```

Note

Do not add the PF itself using "ovs-vsctl add-port" to the OVS.

Setting IPSec Packet Offload Using strongSwan

strongSwan configures IPSec HW packet offload using a new value added to its configuration file `swanctl.conf` (as of strongSwan version 5.9.10).

The file should be placed under "sysconfdir" which by default can be found at `/etc/swanctl/swanctl.conf`.

The terms Left (BFL) and Right (BFR) are used to identify the two nodes that communicate (corresponding with the figure under section "IPsec Packet Offload strongSwan Support").

In this example, 192.168.50.1 is used for the left PF uplink and 192.168.50.2 for the right PF uplink.

```
connections {
  BFL-BFR {
    local_addrs  = 192.168.50.1
    remote_addrs = 192.168.50.2

    local {
      auth = psk
      id = host1
```

```
        }
        remote {
          auth = psk
          id = host2
        }
                children {
          bf-out {
            local_ts = 192.168.50.1/24 [udp]
            remote_ts = 192.168.50.2/24 [udp/4789]
            esp_proposals = aes128gcm128-x25519-esn
            mode = transport
            policies_fwd_out = yes
            hw_offload = packet
          }
          bf-in {
            local_ts = 192.168.50.1/24 [udp/4789]
            remote_ts = 192.168.50.2/24 [udp]
            esp_proposals = aes128gcm128-x25519-esn
            mode = transport
            policies_fwd_out = yes
            hw_offload = packet
          }
        }
        version = 2
        mobike = no
        reauth_time = 0
        proposals = aes128-sha256-x25519
    }
}

secrets {
    ike-BF {
        id-host1 = host1
        id-host2 = host2
        secret = 0sv+NkxY9LLZvwj4qCC2o/gGrWDF2d21jL
    }
}
```

Note

BFB installation will place two example swanctl.conf files for both Left and Right nodes (BFL.swanctl.conf and BFR.swanctl.conf respectively) in the strongSwan conf.d directory. Please move one of them manually to the other machine and edit it according to your configuration.

Note that:

- "hw_offload = packet" is responsible for configuring IPsec packet offload
- Packet offload support has been added to the existing hw_offload field and preserves backward compatibility.

  For your reference:

Value	Description
no	Do not configure HW offload
crypto	Configure crypto HW offload if supported by the kernel and hardware, fail if not supported
yes	Same as crypto (considered legacy)
packet	Configure packet HW offload if supported by the kernel and hardware, fail if not supported
auto	Configure packet HW offload if supported by the kernel and hardware, do not fail (perform fallback to crypto or no as necessary)

Note

Whenever the value of hw_offload is changed, strongSwan configuration must be reloaded.

- [udp/4789] is crucial for instructing strongSwan to IPSec only VXLAN communication

ⓘ **Note**

Packet HW offload can only be done on what is streamed over VXLAN.

Mind the following limitations:

Field	Limitation
reauth_time	Ignored if set
rekey_time	Do not use. Ignored if set.
rekey_bytes	Do not use. Not supported and will fail if it is set.
rekey_packets	Use for rekeying

Running strongSwan Example

Notes:

- IPsec daemons are started by systemd `strongswan.service`, users must avoid using `strongswan-starter.service` as it is a legacy service and using both services at the same time leads to anomalous behavior
- Use `systemctl [start | stop | restart]` to control IPsec daemons through `strongswan.service`. For example, to restart, the command `systemctl restart strongswan.service` will effectively do the same thing as `ipsec restart`.

⚠ **Warning**

Do **not** use `ipsec` script to restart/stop/start.

If you are using the `ipsec` script, then, in order to restart or start the daemons, `openssl.cnf.orig` must be copied to openssl.cnf before performing `ipsec restart` or `ipsec start`. Then `openssl.cnf.mlnx` can be copied to openssl.cnf after restart or start. Failing to do so can result in errors since openssl.cnf.mlnx allows IPsec PK and RNG hardware offload via the OpenSSL plugin.

- On Ubuntu/Debian/Yocto, openssl.cnf* can be found under /etc/ssl/
- On CentOS, openssl.cnf* can be found under /etc/pki/tls/

- The strongSwan package installs openssl.cnf config files to enable hardware offload of PK and RNG operations via the OpenSSL plugin
- The OpenSSL dynamic engine is used to carry out the offload to hardware. OpenSSL dynamic engine ID is "pka".

Procedure:

1. Perform the following on Left and Right devices (corresponding with the figure under section "IPsec Packet Offload strongSwan Support").

```
# systemctl start strongswan.service
# swanctl --load-all
```

   The following should appear.

```
Starting strongSwan 5.9.10bf IPsec [starter]...
no files found matching '/etc/ipsec.d/*.conf'
# deprecated keyword 'plutodebug' in config setup
# deprecated keyword 'virtual_private' in config setup
loaded ike secret 'ike-BF'
no authorities found, 0 unloaded
no pools found, 0 unloaded
loaded connection 'BFL-BFR'
successfully loaded 1 connections, 0 unloaded
```

2. Perform the actual connection **on one side only** (client, Left in this case).

```
# swanctl -i --child bf-in bf-out
```

The following should appear.

```
[IKE] initiating IKE_SA BFL-BFR[1] to 192.168.50.2
[ENC] generating IKE_SA_INIT request 0 [ SA KE No N(NATD_S_IP) N(NATD_D_IP) N(FRAG_SUP)
N(HASH_ALG) N(REDIR_SUP) ]
[NET] sending packet: from 192.168.50.1[500] to 192.168.50.2[500] (240 bytes)
[NET] received packet: from 192.168.50.2[500] to 192.168.50.1[500] (273 bytes)
[ENC] parsed IKE_SA_INIT response 0 [ SA KE No N(NATD_S_IP) N(NATD_D_IP) CERTREQ
N(FRAG_SUP) N(HASH_ALG) N(CHDLESS_SUP) N(MULT_AUTH) ]
[CFG] selected proposal:
IKE:AES_CBC_128/HMAC_SHA2_256_128/PRF_HMAC_SHA2_256/CURVE_25519
[IKE] received 1 cert requests for an unknown ca
[IKE] authentication of 'host1' (myself) with pre-shared key
[IKE] establishing CHILD_SA bf{1}
[ENC] generating IKE_AUTH request 1 [ IDi N(INIT_CONTACT) IDr AUTH N(USE_TRANSP) SA TSi TSr
N(MULT_AUTH) N(EAP_ONLY) N(MSG_ID_SYN_SUP) ]
[NET] sending packet: from 192.168.50.1[500] to 192.168.50.2[500] (256 bytes)
[NET] received packet: from 192.168.50.2[500] to 192.168.50.1[500] (224 bytes)
[ENC] parsed IKE_AUTH response 1 [ IDr AUTH N(USE_TRANSP) SA TSi TSr N(AUTH_LFT) ]
[IKE] authentication of 'host2' with pre-shared key successful
[IKE] IKE_SA BFL-BFR[1] established between 192.168.50.1[host1]...192.168.50.2[host2]
[IKE] scheduling reauthentication in 10027s
[IKE] maximum IKE_SA lifetime 11107s
[CFG] selected proposal: ESP:AES_GCM_16_128/NO_EXT_SEQ
[IKE] CHILD_SA bf{1} established with SPIs ce543905_i c60e98a2_o and TS 192.168.50.1/32 ===
192.168.50.2/32
initiate completed successfully
```

You may now send encrypted data over the HOST VF interface (192.168.70.[1|2]) configured for VXLAN.

Building strongSwan

Do this only if you want to build your own BFB and would like to rebuild strongSwan.

1. Install dependencies mentioned here. libgmp-dev is missing from that list, so make sure to install that as well.
2. Git clone https://github.com/Mellanox/strongswan.git.
3. Git checkout BF-5.9.10. This branch is based on the official strongSwan 5.9.10 branch with added packaging and support for DOCA IPsec plugin (check the NVIDIA DOCA IPsec Security Gateway Application Guide for more information regarding the strongSwan DOCA plugin).
4. Run `autogen.sh` within the strongSwan repo.
5. Run the following:

```
configure --enable-openssl --disable-random --prefix=/usr/local --sysconfdir=/etc --enable-systemd
make
make install
```

Note:

- `--enable-systemd` enables the systemd service for strongSwan present inside the GitHub repo (see step 3) at `init/systemd-starter/strongswan.service.in`.
- When building strongSwan on your own, the `openssl.cnf.mlnx` file, required for PK and RNG HW offload via OpenSSL plugin, is not installed. It must be copied over manually from github repo inside the openssl-conf directory. See section "Running Strongswan Example" for important notes.

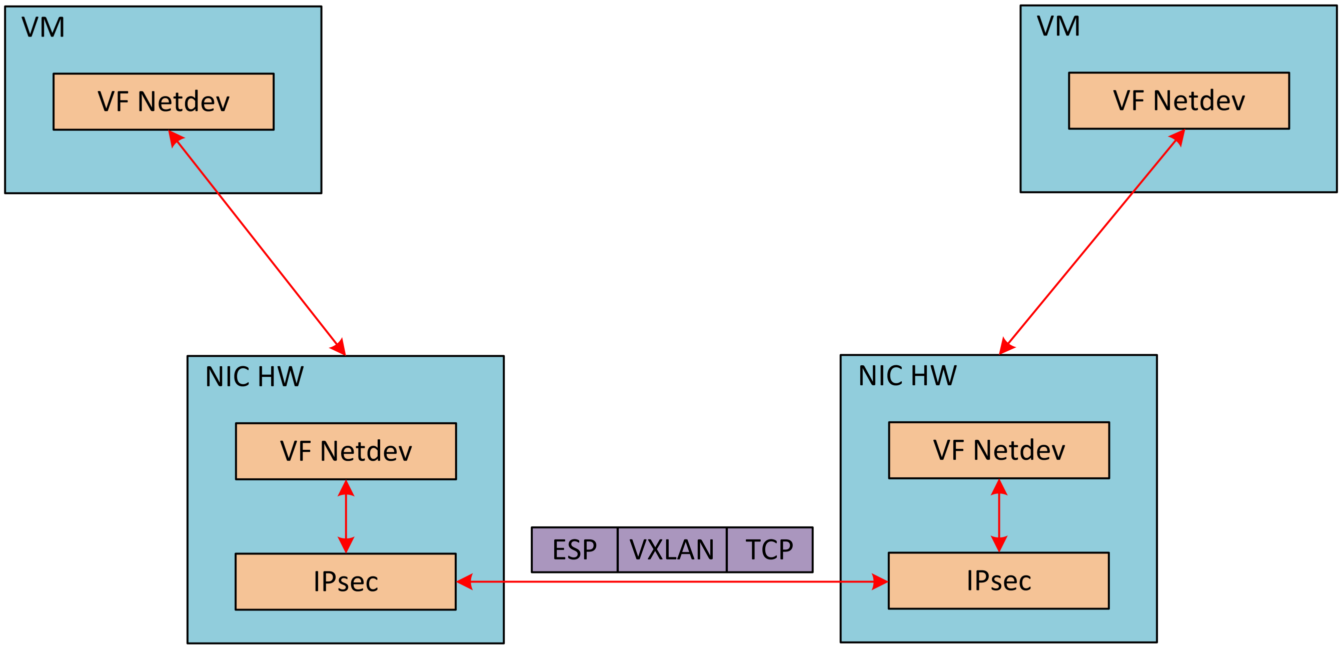
Note

The `openssl.cnf.mlnx` file references PKA engine shared objects. libpka (version 1.3 or later) and openssl (version 1.1.1) must be installed for this to work.

IPsec Packet Offload and OVS Offload

IPsec packet offload configuration works with and is transparent to OVS offload. This means all packets from OVS offload are encrypted by IPsec rules.

The following figure illustrates the interaction between IPsec packet offload and OVS VXLAN offload.



Note

OVS offload and IPsec IPv6 do not work together.

OVS IPsec

To start the service, run:

```
systemctl start openvswitch-ipsec.service
```

Refer to section "Enabling IPsec Packet Offload" for information to prepare the IPsec packet offload environment.

Configuring IPsec Tunnel

For the sake of example, if you want to build an IPsec tunnel between two hosts with the following external IP addresses:

- host1 – 1.1.1.1
- host2 – 1.1.1.2

You have to first make sure host1 and host2 can ping each other via these external IPs.

This example will set up some variables on both hosts, set ip1 and ip2:

```
# ip1=1.1.1.1
# ip2=1.1.1.2
REP=eth5
PF=p0
```

1. Set up OVS bridges in both hosts.
    1. On Arm_1:

        ```
        ovs-vsctl add-br ovs-br
        ovs-vsctl add-port ovs-br $REP
        ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
        ```

    2. On Arm_2:

        ```
        ovs-vsctl add-br ovs-br
        ```

```
ovs-vsctl add-port ovs-br $REP
ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
```

Note

Configuring `other_config:hw-offload=true` sets IPsec packet offload. Setting it to `false` sets software IPsec. Make sure that IPsec devlink's mode is set back to `none` for software IPsec.

2. Set up IPsec tunnel. Three authentication methods are possible. Follow the steps relevant for the method that works best for your environment.

Note

Do not try to use more than 1 authentication method.

Note

After the IPsec tunnel is set up, strongSwan configuration will be automatically done.

3. Make sure the MTU of the PF used by tunnel is at least 50 bytes larger than VXLAN-REP MTU.

    1. Disable host PF as the port owner from Arm (see section "Zero-trust Mode"). Run:

```
$ mlxprivhost -d /dev/mst/mt41682_pciconf0 --disable_port_owner r
```

2. The MTU of the end points (`pf0hpf` in the example above) of the tunnel must be smaller than the MTU of the tunnel interfaces (`p0`) to account for the size of the tunnel headers. For example, you can set the MTU of P0 to 2000.

Authentication Methods

Using Pre-shared Key

Note

The following example uses `tun type=gre` and `dst_port=1723`. Depending on your configuration, `tun type` can be `vxlan` or `geneve` with `dst_port` 4789 or 6081 respectively.

Note

The following example uses `ovs-br` as the bridge name. However, this value can be any string you have chosen to create the bridge previously.

1. On `Arm_1`, run:

```
# ovs-vsctl add-port ovs-br tun -- \
        set interface tun type=gre \
                options:local_ip=$ip1 \
                options:remote_ip=$ip2 \
                options:key=100 \
                options:dst_port=1723 \
                options:psk=swordfish
```

2. On Arm_2, run:

```
# ovs-vsctl add-port ovs-br tun -- \
        set interface tun type=gre \
                options:local_ip=$ip2 \
                options:remote_ip=$ip1 \
                options:key=100 \
                options:dst_port=1723 \
                options:psk=swordfish
```

Using Self-signed Certificate

1. Generate self-signed certificates in both host1 and host2, then copy the certificate of host1 to host2, and the certificate of host2 to host1.

2. Move both host1-cert.pem and host2-cert.pem to /etc/swanctl/x509/, if on Ubuntu, or /etc/strongswan/swanctl/x509/, if on CentOS.

3. Move the local private key to /etc/swanctl/private, if on Ubuntu, or /etc/strongswan/swanctl/private, if on CentOS. For example, for host1:

```
mv host1-privkey.pem /etc/swanctl/private
```

4. Set up OVS other_config on both sides.

    1. On Arm_1:

    ```
    # ovs-vsctl set Open_vSwitch . other_config:certificate=/etc/swanctl/x509/host1-cert.pem \
      other_config:private_key=/etc/swanctl/private/host1-privkey.pem
    ```

    2. On Arm_2:

```
# ovs-vsctl set Open_vSwitch . other_config:certificate=/etc/swanctl/x509/host2-cert.pem \
  other_config:private_key=/etc/swanctl/private/host2-privkey.pem
```

5. Set up the tunnel.
    1. On `Arm_1`:

```
# ovs-vsctl add-port ovs-br vxlanp0 -- set interface vxlanp0 type=vxlan
options:local_ip=$ip1 \
  options:remote_ip=$ip2 options:key=100 options:dst_port=4789 \
  options:remote_cert=/etc/swanctl/x509/host2-cert.pem
# service openvswitch-switch restart
```

    2. On `Arm_2`:

```
# ovs-vsctl add-port ovs-br vxlanp0 -- set interface vxlanp0 type=vxlan
options:local_ip=$ip2 \
  options:remote_ip=$ip1 options:key=100 options:dst_port=4789 \
  options:remote_cert=/etc/swanctl/x509/host1-cert.pem
# service openvswitch-switch restart
```

Using CA-signed Certificate

1. For this method, you need all the certificates and the requests to be in the same directory during the certificate generating and signing. This example refers to this directory as `certsworkspace`.
    1. On `Arm_1`:

```
# ovs-pki init --force
# cp /var/lib/openvswitch/pki/controllerca/cacert.pem <path_to>/certsworkspace
# ovs-pki req -u host1
# ovs-pki sign host1 switch
```

2. On Arm_2:

```
# ovs-pki init --force
# cp /var/lib/openvswitch/pki/controllerca/cacert.pem <path_to>/certsworkspace
# ovs-pki req -u host2
# ovs-pki sign host2 switch
```

2. Move both host1-cert.pem and host2-cert.pem to /etc/ swanctl/x509/, if on Ubuntu, or /etc/strongswan/swanctl/x509/, if on CentOS.

3. Move the local private key to /etc/swanctl/private, if on Ubuntu, or /etc/strongswan/swanctl/private, if on CentOS. For example, for host1:

```
mv host1-privkey.pem /etc/swanctl/private
```

4. Copy cacert.pem to the x509ca directory under /etc/swanctl/x509ca/, if on Ubuntu, or /etc/strongswan/swanctl/x509ca/, if on CentOS.

5. Set up OVS other_config on both sides.

   1. On Arm_1:

```
# ovs-vsctl set Open_vSwitch . \
      other_config:certificate=/etc/strongswan/swanctl/x509/host1.pem \
      other_config:private_key=/etc/strongswan/swanctl/private/host1-privkey.pem \
      other_config:ca_cert=/etc/strongswan/swanctl/x509ca/cacert.pem
```

   2. On Arm_2:

```
# ovs-vsctl set Open_vSwitch . \
      other_config:certificate=/etc/strongswan/swanctl/x509/host2.pem \
      other_config:private_key=/etc/strongswan/swanctl/private/host2-privkey.pem \
        other_config:ca_cert=/etc/strongswan/swanctl/x509ca/cacert.pem
```

6. Set up the tunnel:

1.

   1. On Arm_1:

```
# ovs-vsctl add-port ovs-br vxlanp0 -- set interface vxlanp0 type=vxlan
options:local_ip=$ip1 \
options:remote_ip=$ip2 options:key=100 options:dst_port=4789 \
options:remote_name=host2
 #service openvswitch-switch restart
```

   2. On Arm_2:

```
# ovs-vsctl add-port ovs-br vxlanp0 -- set interface vxlanp0 type=vxlan
options:local_ip=$ip2 \
options:remote_ip=$ip1 options:key=100 options:dst_port=4789 \
options:remote_name=host1
#service openvswitch-switch restart
```

Ensuring IPsec is Configured

Use `/opt/mellanox/iproute2/sbin/ip xfrm state show`. You should be able to see IPsec states with the keyword `in mode packet`.

Troubleshooting

For troubleshooting information, refer to Open vSwitch's official documentation.

fTPM over OP-TEE

Note

fTMP over OP-TEE is supported on NVIDIA® BlueField®-3 DPUs and higher only on host OS Ubuntu 22.04 or Oracle Linux.

The Trusted Computing Group (TCG) is responsible for the specifications governing the trusted platform module (TPM). In many systems, the TPM provides integrity measurements, health checks and authentication services.

Attributes of a TPM:

- Support for bulk (symmetric) encryption in the platform
- High quality random numbers
- Cryptographic services
- Protected persistent store for small amounts of data, sticky bits, monotonic counters, and extendible registers
- Protected pseudo-persistent store for unlimited amounts of keys and data
- Extensive choice of authorization methods to access protected keys and data
- Platform identities
- Support for platform privacy
- Signing and verifying digital signatures
- Certifying the properties of keys and data

- Auditing the usage of keys and data

With TPM 2.0., the TCG creates a library specification describing all the commands or features that could be implemented and may be necessary in servers, laptops, or embedded systems. Each platform can select the features needed and the level of security or assurance required. This flexibility allows the newest TPMs to be applied to many embedded applications.

Firmware TPM (fTPM) is implemented in protected software. The code runs on the main CPU so that a separate chip is not required. While running like any other program, the code is in a protected execution environment called a trusted execution environment (TEE) which is separate from the rest of the programs running on the CPU. By doing this, secrets (e.g., private keys perhaps needed by the TPM but should not be accessed by others) can be kept in the TEE creating a more secure environment.

Info

fTPM provides similar functionality to a chip-based TPM, but does not require extra hardware. It complies with the official TCG reference implementation of the TPM 2.0 specification . The source code of this implementation is located here.

Info

fTPM f ully supports TPM2 Tools and the TCG TPM2 Software Stack (TSS).

Characteristics of an fTPM:

- Emulated TPM using an isolated hardware environment
- Executes in an open-source trusted execution environment (OP-TEE)

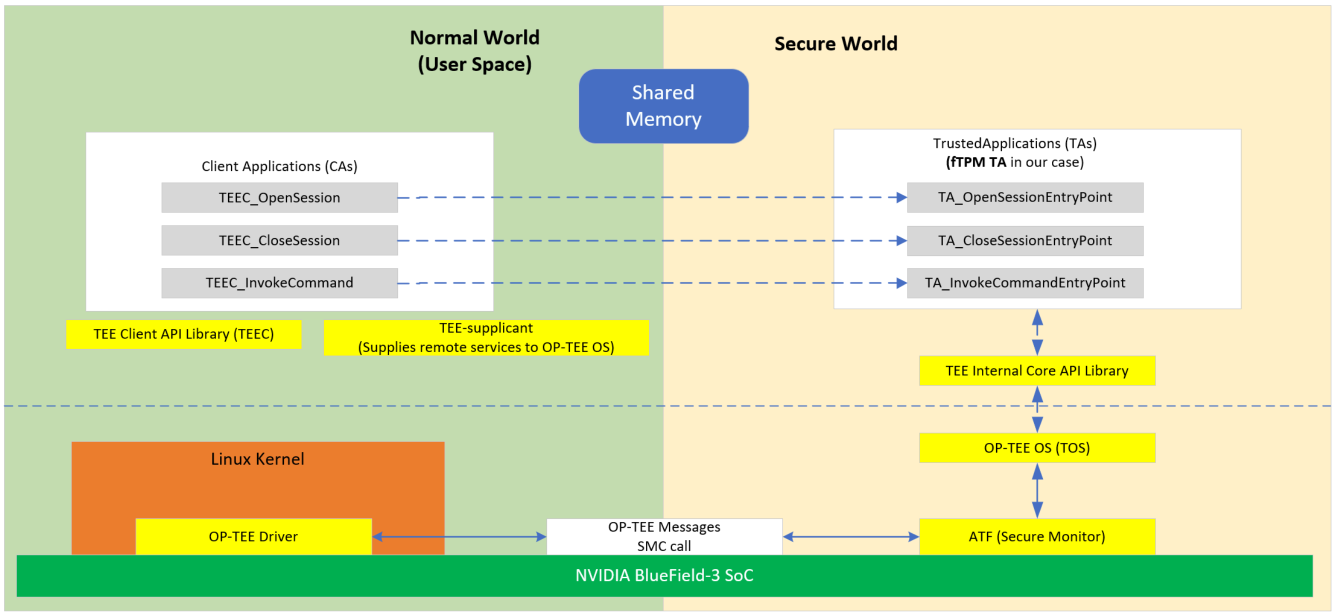
- fTPM trusted application (TA) is part of the OP-TEE binary. This allows early access on bootup, runs only in secure DRAM.

Info

Currently, the only TA supported is fTPM.

- fTPM is not a task waiting to be woken up. It only executes when TPM primitives are forwarded to it from the user space. It is guaranteed shielded execution via the TEE OS and, when invoked via the TEE Dispatcher, runs to completion.

The fTPM TA is the only TA BlueField-3 currently supports. Any TA loaded by OP-TEE must be signed (signing done externally) and then authenticated by OP-TEE before being allowed to load and execute.

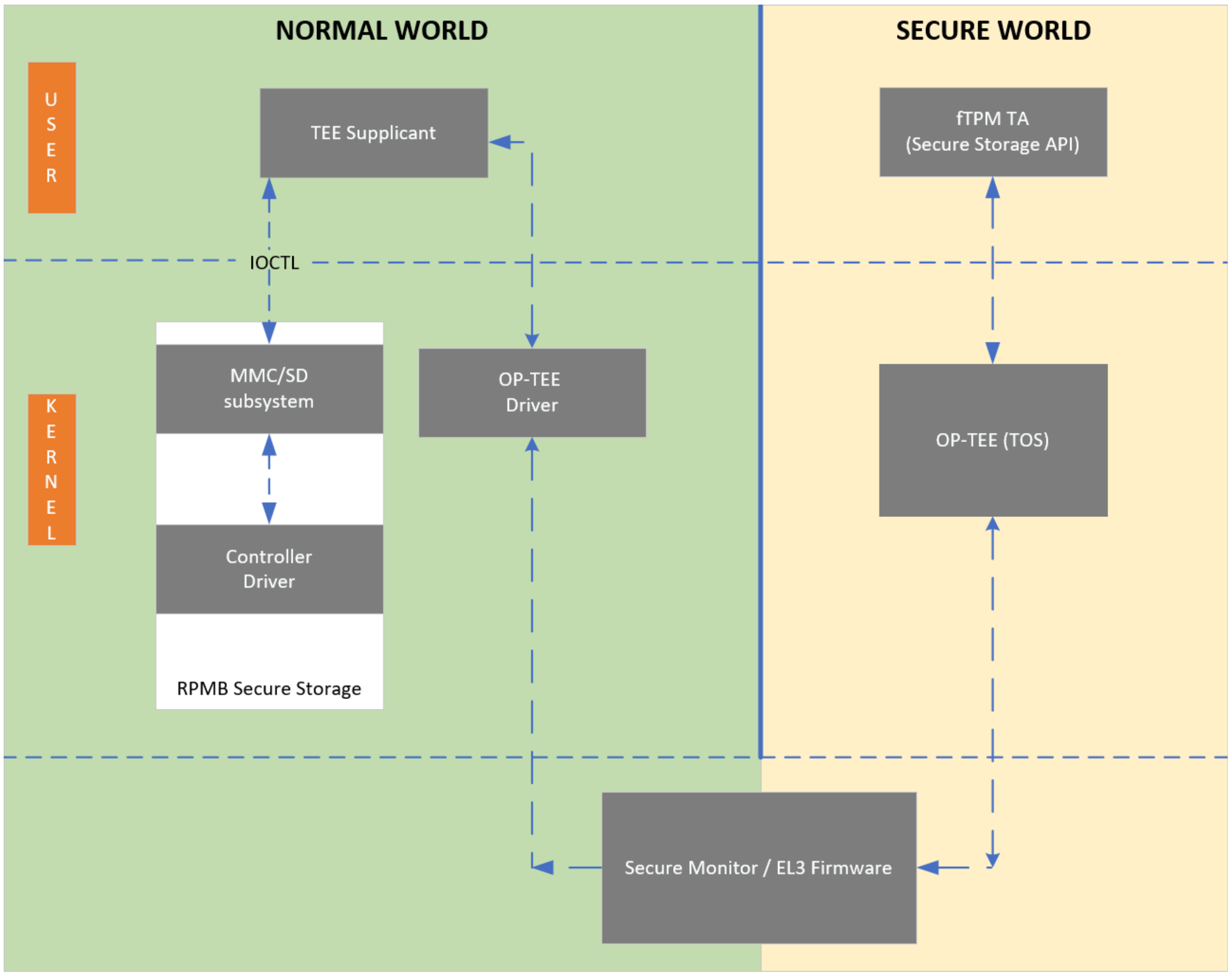


A replay-protected memory block (RPMB) is provided as a means for a system to store data to the specific memory area in an authenticated and replay-protected manner, making it readable and writable only after a successful authentication read/write accesses. The RPMB is a dedicated partition available on the eMMC, which makes it possible to store and retrieve data with integrity and authenticity support. A signed access to an RPMB is supported by first programming authentication key information to the eMMC memory (shared secret). The RPMB authentication key is programmed into BlueField at manufacturing time.

Info

RPMB features a 4MB partition secure storage for BlueField-3.

There is no eMMC controller driver in OP-TEE. All device operations have to go through the normal world via the TEE-supplicant daemon, which relies on the Linux kernel's ioctl interface to access the device. All writes to the RPMB are atomic, authenticated, and encrypted. The RPMB partition stores data in an authenticated, replay-protected manner, making it a perfect complement to fTPM for storing and protecting data.



Enabling OP-TEE on BlueField-3

Enable OP-TEE in the UEFI menu:

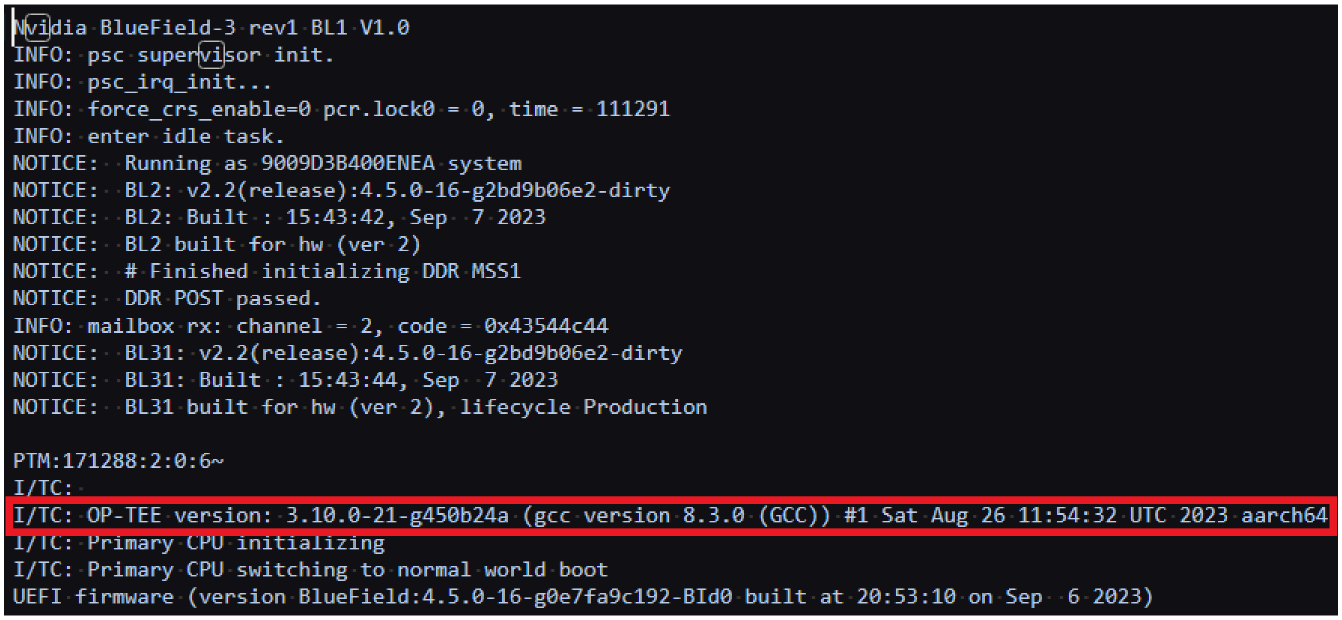
1. ESC into the UEFI on BlueField boot.
2. Navigate to Device Manager > System Configuration.
3. Check "Enable OP-TEE".
4. Save the change and reset/reboot.
5. Upon reboot OP-TEE is enabled.

Note

OP-TEE is essentially dormant (does not have an OS scheduler) and reacts to external inputs.

Verifying BlueField-3 is Running OP-TEE

Users can see the OP-TEE version during BlueField-3 boot:



```
Nvidia BlueField-3 rev1 BL1 V1.0
INFO: psc supervisor init.
INFO: psc_irq_init...
INFO: force_crs_enable=0 pcr.lock0 = 0, time = 111291
INFO: enter idle task.
NOTICE:  Running as 9009D3B400ENEA system
NOTICE:  BL2: v2.2(release):4.5.0-16-g2bd9b06e2-dirty
NOTICE:  BL2: Built : 15:43:42, Sep  7 2023
NOTICE:  BL2 built for hw (ver 2)
NOTICE:  # Finished initializing DDR MSS1
NOTICE:  DDR POST passed.
INFO: mailbox rx: channel = 2, code = 0x43544c44
NOTICE:  BL31: v2.2(release):4.5.0-16-g2bd9b06e2-dirty
NOTICE:  BL31: Built : 15:43:44, Sep  7 2023
NOTICE:  BL31 built for hw (ver 2), lifecycle Production

PTM:171288:2:0:6~
I/TC:
I/TC: OP-TEE version: 3.10.0-21-g450b24a (gcc version 8.3.0 (GCC)) #1 Sat Aug 26 11:54:32 UTC 2023 aarch64
I/TC: Primary CPU initializing
I/TC: Primary CPU switching to normal world boot
UEFI firmware (version BlueField:4.5.0-16-g0e7fa9c192-BId0 built at 20:53:10 on Sep  6 2023)
```

The following indicators should all be present if fTPM over OP-TEE is enabled:

- Check "dmesg" for the OP-TEE driver initializing

```
root@localhost ~]# dmesg | grep tee
[    5.646578] optee: probing for conduit method.
[    5.653282] optee: revision 3.10 (450b24ac)
[    5.653991] optee: initialized driver
```

- Verify that the following kernel modules are loaded (running):

```
[root@localhost ~]# lsmod | grep tee
tpm_ftpm_tee           16384  0
optee                  49152  1
tee                    49152  3 optee,tpm_ftpm_tee
```

- Verify that the proper devices are created/available (4 in total):

```
[root@localhost ~]# ls -l /dev/tee*
crw------- 1 root root 234,  0 Sep  8 18:24 /dev/tee0
crw------- 1 root root 234, 16 Sep  8 18:24 /dev/teepriv0

[root@localhost ~]# ls -l /dev/tpm*
crw-rw---- 1 tss root  10,   224 Sep  8 18:24 /dev/tpm0
crw-rw---- 1 tss tss  252, 65536 Sep  8 18:24 /dev/tpmrm0
```

- Verify that the required processes are running (3 in total):

```
[root@localhost ~]# ps axu | grep tee
root       707  0.0  0.0  76208  1372 ?        Ssl  14:42   0:00 /usr/sbin/tee-supplicant
root       715  0.0  0.0      0     0 ?        I<   14:42   0:00 [optee_bus_scan]

[root@localhost ~]# ps axu | grep tpm
root       124  0.0  0.0      0     0 ?        I<   18:24   0:00 [tpm_dev_wq]
```

QoS Configuration

Note

To learn more about port QoS configuration, refer to this community post.

Warning

When working in Embedded Host mode, using mlnx_qos on both the host and Arm will result with undefined behavior. Users must only use mlnx_qos from the Arm. After changing the QoS settings from Arm, users must restart the mlx5 driver on host.

Note

When configuring QoS using DCBX, the lldpad service from the NVIDIA® BlueField® networking platform's (DPU or SuperNIC) side must be disabled if the configurations are not done using tools other than lldpad.

This section explains how to configure QoS group and settings using devlink located under /opt/mellanox/iproute2/sbin/. It is applicable to host PF/VF and Arm side SFs. The following uses VF as example.

The settings of a QoS group include creating/deleting a QoS group and modifying its tx_max and tx_share values. The settings of VF QoS include modifying its tx_max and tx_share values, assigning a VF to a QoS group, and unassigning a VF from a QoS group. This section focuses on the configuration syntax.

Please refer to section "Limit and Bandwidth Share Per VF" in the MLNX_OFED User Manual for detailed explanation on vPort QoS behaviors.

devlink port function rate add

	devlink port function rate add <DEV>/<GROUP_NAME> Adds a QoS group.	
Syntax Description	DEV/GROUP_NAME	Specifies group name in string format
Example	This command adds a new QoS group named 12_group under device pci/0000:03:00.0: `devlink port function rate add pci/0000:03:00.0/12_group`	
Notes		

devlink port function rate del

	devlink port function rate del <DEV>/<GROUP_NAME> Deletes a QoS group.	
Syntax Description	DEV/GROUP_NAME	Specifies group name in string format
Example	This command deletes QoS group 12_group from device pci/0000:03:00.0: `devlink port function rate del pci/0000:03:00.0/12_group`	
Notes		

devlink port function rate set tx_max tx_share

	devlink port function rate set {<DEV>/<GROUP_NAME> | <DEV>/<PORT_INDEX>} tx_max <TX_MAX> [tx_share <TX_SHARE>]

	Sets tx_max and tx_share for QoS group or devlink port.	
Syntax Description	DEV/GROUP_NAME	Specifies the group name to operate on
	DEV/PORT_INDEX	Specifies the devlink port to operate on
	TX_MAX	tx_max bandwidth in MB/s
	TX_SHARE	tx_share bandwidth in MB/s

Example	This command sets tx_max to 2000MB/s and tx_share to 500MB/s for the 12_group QoS group:
	`devlink port function rate set pci/0000:03:00.0/12_group tx_max 2000MBps tx_share 500MBps`
	This command sets tx_max to 2000MB/s and tx_share to 500MB/s for the VF represented by port index 196609:
	`devlink port function rate set pci/0000:03:00.0/196609 tx_max 200MBps tx_share 50MBps`
	This command displays a mapping between VF devlink ports and netdev names:
	`$ devlink port`
	In the output of this command, VFs are indicated by flavour pcivf.	

Notes		

devlink port function rate set parent

	devlink port function rate set <DEV>/<PORT_INDEX> {parent <PARENT_GROUP_NAME>} Assigns devlink port to a QoS group.	
Syntax Description	DEV/PORT_INDEX	Specifies the devlink port to operate on
	PARENT_GROUP_NAME	parent group name in string format
Example	This command assigns this function to the QoS group 12_group:	

	```devlink port function rate set pci/0000:03:00.0/196609 parent 12_group```
Notes	

devlink port function rate set noparent

	devlink port function rate set <DEV>/<PORT_INDEX> noparent Ungroups a devlink port.	
Syntax Description	DEV/PORT_INDEX	Specifies the devlink port to operate on
Example	This command ungroups this function: ```devlink port function rate set pci/0000:03:00.0/196609 noparent```	
Notes		

devlink port function rate show

	devlink port function rate show [<DEV>/<GROUP_NAME> | <DEV>/<PORT_INDEX>] Displays QoS information QoS group or devlink port.	
Syntax Description	DEV/GROUP_NAME	Specifies the group name to display
	DEV/PORT_INDEX	Specifies the devlink port to display

Example	This command displays the QoS info of all QoS groups and devlink ports on the system:
	```
	devlink port function rate show
	pci/0000:03:00.0/12_group type node tx_max 2000MBps tx_share 500MBps
	pci/0000:03:00.0/196609 type leaf tx_max 200MBps tx_share 50MBps parent 12_group
	```
	This command displays QoS info of `12_group`:
	```
	devlink port function rate show pci/0000:03:00.0/12_group
	pci/0000:03:00.0/12_group type node tx_max 2000MBps tx_share 500MBps
	```

Notes	If a QoS group name or devlink port are not specified, all QoS groups and devlink ports are displayed.

Virtio-net Emulated Devices

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

Shared RQ Mode

When creating 1 send queue (SQ) and 1 receive queue (RQ), each representor consumes ~3MB memory per single channel. Scaling this to the desired 1024 representors (SFs and/or VFs) would require ~3GB worth of memory for single channel. A major chunk of the 3MB is contributed by RQ allocation (receive buffers and SKBs). Therefore, to make efficient use of memory, shared RQ mode is implemented so PF/VF/SF representors share receive queues owned by the uplink representor.

The feature is enabled by default. To disable it:

1. Edit the field ALLOW_SHARED_RQ in /etc/mellanox/mlnx-bf.conf as follows:

   ```
   ALLOW_SHARED_RQ="no"
   ```

2. Restart the driver. Run:

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

To connect from the host to NVIDIA® BlueField® networking platform (DPU or SuperNIC) in shared RQ mode, please refer to section Verifying Connection from Host to BlueField.

Note

PF/VF representor to PF/VF communication on the host is not possible.

The following behavior is observed in shared RQ mode:

- It is expected to see a `0` in the `rx_bytes` and `rx_packets` and valid `vport_rx_packets` and `vport_rx_bytes` after running traffic. Example output:

```
# ethtool -S pf0hpf
NIC statistics:
     rx_packets: 0
     rx_bytes: 0
     tx_packets: 66946
     tx_bytes: 8786869
     vport_rx_packets: 546093
     vport_rx_bytes: 321100036
     vport_tx_packets: 549449
     vport_tx_bytes: 321679548
```

- Ethtool usage – in this mode, it is not possible to change/set the ring or coalesce parameters for the RX side using ethtool. Changing channels also only affects the TX side.