

DOCA SNAP-4 Service Guide

Table of contents

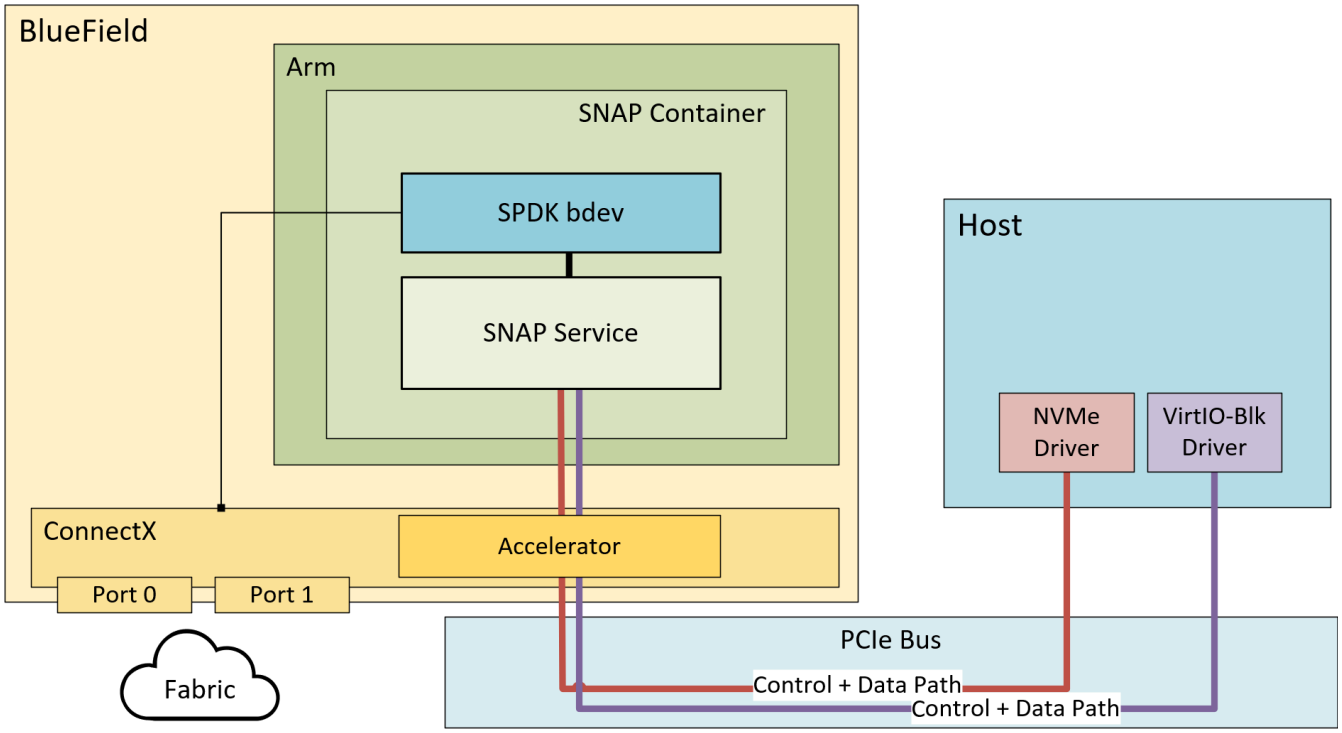
SNAP-4 Service Release Notes 142

This guide provides instructions on using the DOCA SNAP-4 service on top of the NVIDIA® BlueField®-3 DPU.

Introduction

NVIDIA® BlueField® SNAP and virtio-blk SNAP (storage-defined network accelerated processing) technology enables hardware-accelerated virtualization of local storage. NVMe/virtio-blk SNAP presents networked storage as a local block-storage device (e.g., SSD) emulating a local drive on the PCIe bus. The host OS or hypervisor uses its standard storage driver, unaware that communication is done, not with a physical drive, but with NVMe/virtio-blk SNAP framework. Any logic may be applied to the I/O requests or to the data via the NVMe/virtio-blk SNAP framework prior to redirecting the request and/or data over a fabric-based network to remote or local storage targets.

NVMe/virtio-blk SNAP is based on the NVIDIA® BlueField® DPU family technology and combines unique software-defined hardware-accelerated storage virtualization with the advanced networking and programmability capabilities of the DPU. NVMe/virtio-blk SNAP together with the BlueField DPU enable a world of applications addressing storage and networking efficiency and performance.



The traffic arriving from the host towards the emulated PCIe device is redirected to its matching storage controller opened on the `mlnx_snap` service.

The controller implements the device specification and may expose backend device accordingly (in this use case SPDK is used as the storage stack that exposes backend devices). When a command is received, the controller executes it.

Admin commands are mostly answered immediately, while I/O commands are redirected to the backend device for processing.

The request-handling pipeline is completely asynchronous, and the workload is distributed across all Arm cores (allocated to SPDK application) to achieve the best performance.

The following are key concepts for SNAP:

- Full flexibility in fabric/transport/protocol (e.g. NVMe-oF/iSCSI/other, RDMA/TCP, ETH/IB)
- NVMe and virtio-blk emulation support
- Programmability
- Easy data manipulation
- Allowing zero-copy DMA from the remote storage to the host
- Using Arm cores for data path

Note

BlueField SNAP for NVIDIA® BlueField®-2 DPU is licensed software. Users must purchase a license per BlueField-2 DPU to use them.

NVIDIA® BlueField®-3 DPU does not have license requirements to run BlueField SNAP.

SNAP as Container

In this approach, the container could be downloaded from NVIDIA NGC and could be easily deployed on the DPU.

The yaml file includes SNAP binaries aligned with the latest `spdk.nvda` version. In this case, the SNAP sources are not available, and it is not possible to modify SNAP to support

different SPDK versions (SNAP as an SDK package should be used for that).

> **Note**
>
> SNAP 4.x is not pre- installed on the BFB but can be downloaded manually on demand .

For instructions on how to install the SNAP container, please see "SNAP Container Deployment".

SNAP as a Package

The SNAP development package (custom) is intended for those wishing to customize the SNAP service to their environment, usually to work with a proprietary bdev and not with the `spdk.nvda` version. This allows users to gain full access to the service code and the lib headers which enables them to compile their changes.

SNAP Emulation Lib

This includes the protocols libraries and the interaction with the firmware/hardware (PRM) as well as:

- Plain shared objects (`*.so`)
- Static archives (`*.a`)
- pkgconfig definitions (`*.pc`)
- Include files (`*.h`)

SNAP Service Sources

This includes the following managers:

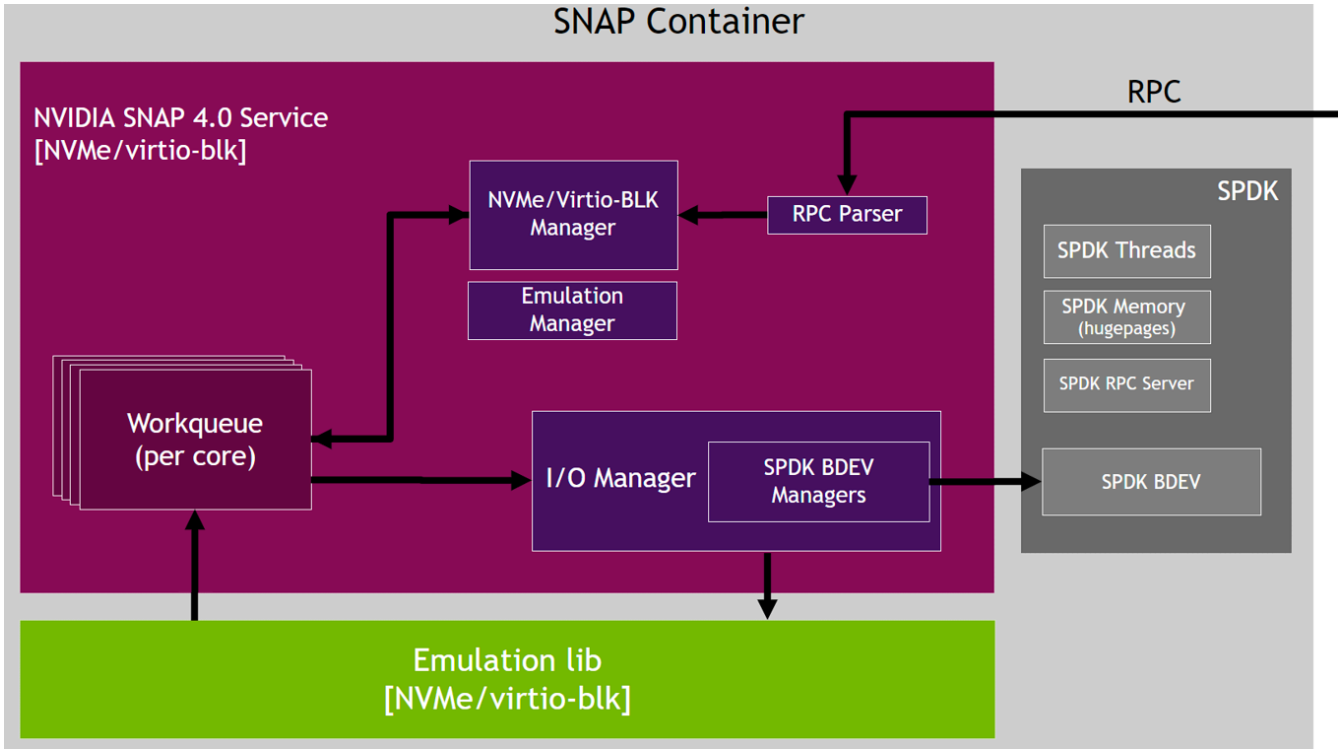
- Emulation device managers:
    - Emulation manager – manages the device emulations, function discovery, and function events
    - Hotplug manager – manages the device emulations hotplug and hot-unplug
    - Config manager – handles common configurations and RCPs (which are not protocol-specific)
- Service infrastructure managers:
    - Memory manager – handles the SNAP mempool which is used to copy into the Arm memory when zero-copy between the host and the remote target is not used
    - Thread manager – handles the SPDK threads
- Protocol specific control path managers:
    - NVMe manager – handles the NVMe subsystem, NVMe controller and Namespace functionalities
    - VBLK manager – handles the virtio-blk controller functionalities
- IO manager:
    - Implements the IO path for regular and optimized flows (RDMA ZC and TCP XLIO ZC)
    - Handles the bdev creation and functionalities

SNAP Service Dependencies

SNAP service depends on the following libraries:

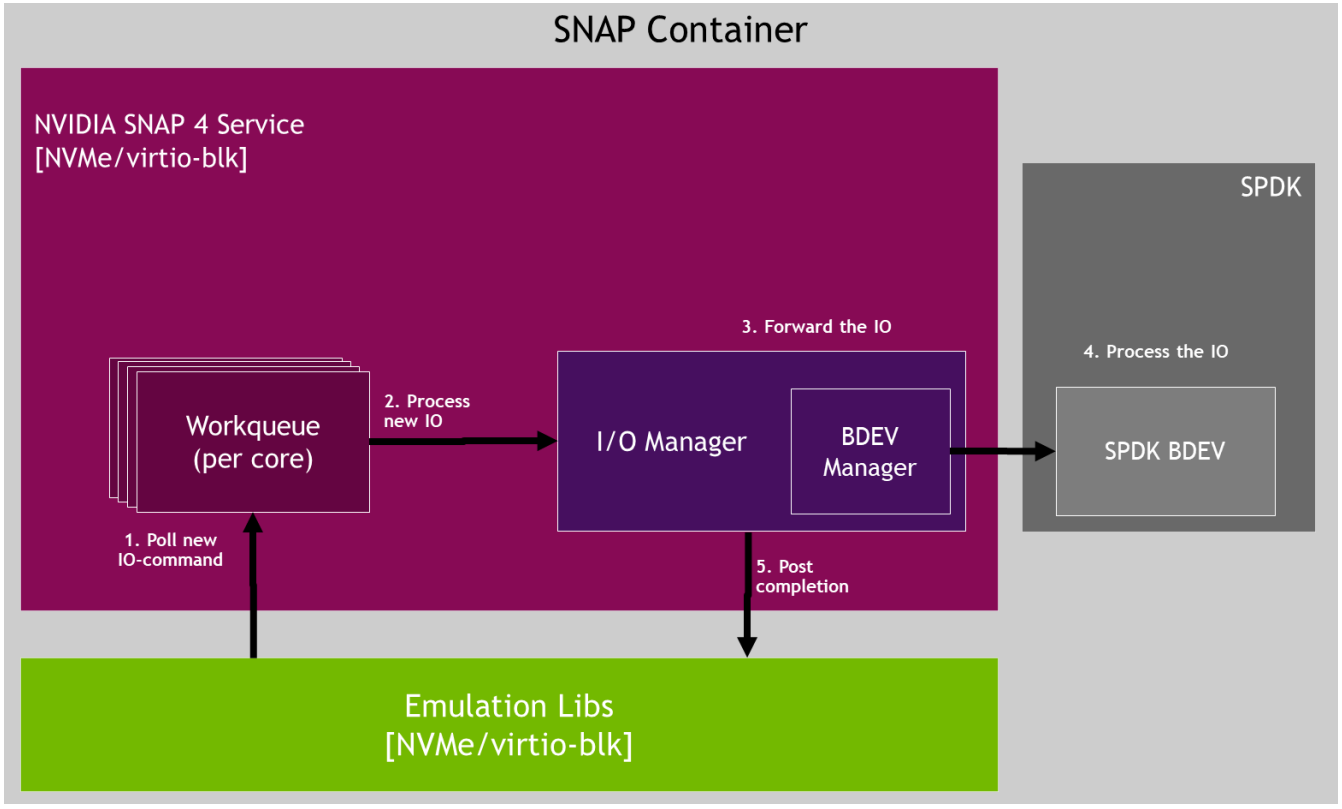
- SPDK – depends on the bdev and the SPDK resources, such as SPDK threads, SPDK memory, and SPDK RPC service
- XLIO (for NVMeTCP acceleration)

SNAP Service Flows

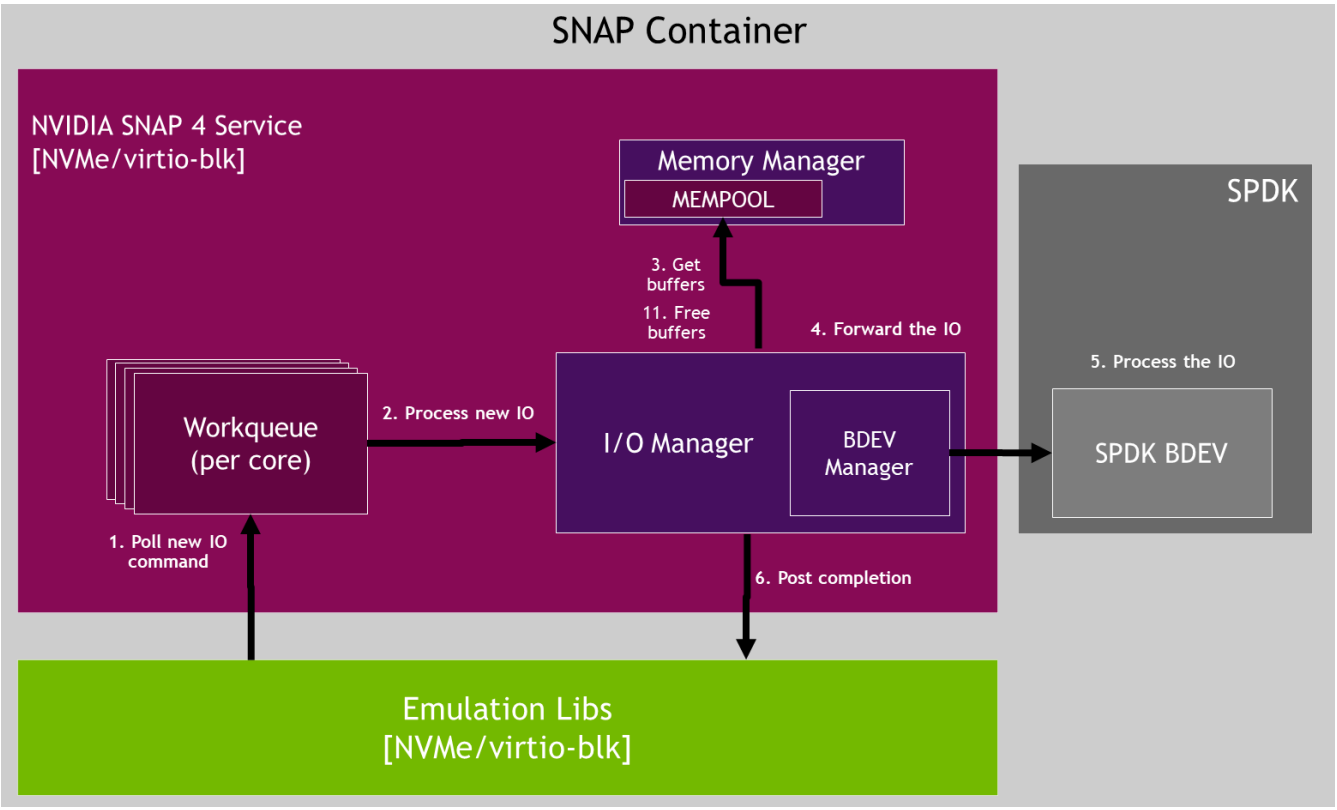


IO Flows

Example of RDMA zero-copy read/write IO flow:



Example of RDMA non-zero-copy read IO flow:



Data Path Providers

SNAP facilitates user-configurable providers to assist in offloading data-path applications from the host. These include: Device emulation, IO-intensive operations, and DMA operations.

Data Path Providers	Description	Support
DPA provider	DPA (data path accelerator) is a cluster of multi-core and multi-execution-unit RISC-V processors embedded within the BlueField.	NVMe, Virtio-blk
DPU provider	Hybrid mode that handles the data-path applications from the host using the BlueField CPU and the DPA hardware. This mode improves IO latency and reduces SNAP downtime during crash recovery .	Virtio-blk

> **Note**
>
> DPA is the default provider in SNAP for NVMe and virtio-blk.
>
> To switch provider, use the environment variable `VIRTIO_EMU_PROVIDER=dpu` to modify the the variable on the YAML. Refer to the "SNAP Environment Variables" page for more information.

SNAP Deployment

This section describes how to deploy SNAP as a container.

> **Note**
>
> SNAP does not come pre-installed with the BFB.

Installing Full DOCA Image on DPU

To install NVIDIA® BlueField®-3 BFB:

```
[host] sudo bfb-install --rshim <rshimN> --bfb <image_path.bfb>
```

For more information, please refer to section "Installing Full DOCA Image on DPU" in the *DOCA Installation Guide for Linux*.

Firmware Installation

```
[dpu] sudo /opt/mellanox/mlnx-fw-updater/mlnx_fw_updater.pl --
force-fw-update
```

For more information, please refer to section "Upgrading Firmware" in the *DOCA Installation Guide for Linux*.

Firmware Configuration

> **Note**
>
> FW configuration may expose new emulated PCI functions, which can be later used by the host's OS. As such, user must make sure all exposed PCI functions (static/hotplug PFs, VFs) are backed by a supporting SNAP SW configuration, otherwise these functions will remain malfunctioning and host behavior will be undefined.

1. Clear the firmware config before implementing the required configuration:

   ```
   [dpu] mst start
   [dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 reset
   ```

2. Review the firmware configuration:

   ```
   [dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 query
   ```

   Output example:

```
mlxconfig -d /dev/mst/mt41692_pciconf0 -e query | grep NVME
Configurations:                                          Default
Current         Next Boot
*        NVME_EMULATION_ENABLE                           False(0)
True(1)         True(1)
*        NVME_EMULATION_NUM_VF                           0
125             125
*        NVME_EMULATION_NUM_PF                           1
2               2
         NVME_EMULATION_VENDOR_ID                        5555
5555            5555
         NVME_EMULATION_DEVICE_ID                        24577
24577           24577
         NVME_EMULATION_CLASS_CODE                       67586
67586           67586
         NVME_EMULATION_REVISION_ID                      0
0               0
         NVME_EMULATION_SUBSYSTEM_VENDOR_ID              0
0               0
```

Where the output provides 5 columns:

- Non-default configuration marker (*)
- Firmware configuration name
- Default firmware value
- Current firmware value
- Firmware value after reboot – shows a configuration update which is pending system reboot

3. To enable storage emulation options, the first DPU must be set to work in internal CPU model:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s
INTERNAL_CPU_MODEL=1
```

4. To enable the firmware config with virtio-blk emulation PF:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s
VIRTIO_BLK_EMULATION_ENABLE=1 VIRTIO_BLK_EMULATION_NUM_PF=1
```

5. To enable the firmware config with NVMe emulation PF:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s
NVME_EMULATION_ENABLE=1 NVME_EMULATION_NUM_PF=1
```

Note

For a complete list of the SNAP firmware configuration options, refer to appendix "DPU Firmware Configuration".

Note

Power cycle is required to apply firmware configuration changes.

RDMA/RoCE Firmware Configuration

RoCE communication is blocked for BlueField OS's default interfaces (named ECPFs, typically `mlx5_0` and `mlx5_1`). If RoCE traffic is required, additional network functions

must be added, scalable functions (or SFs), which do support RoCE transport.

To enable RDMA/RoCE:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s PER_PF_NUM_SF=1
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s PF_SF_BAR_SIZE=8
PF_TOTAL_SF=2
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0.1 s PF_SF_BAR_SIZE=8
PF_TOTAL_SF=2
```

Note

This is not required when working over TCP or RDMA over InfiniBand.

SR-IOV Firmware Configuration

SNAP supports up to 512 total VFs on NVMe and up to 2000 total VFs on virtio-blk. The VFs may be spread between up to 4 virtio-blk PFs or 2 NVMe PFs.

Info

The following examples are for reference. For complete details on parameter ranges, refer to appendix "DPU Firmware Configuration".

- Common example:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s SRIOV_EN=1
PER_PF_NUM_SF=1 LINK_TYPE_P1=2 LINK_TYPE_P2=2 PF_TOTAL_SF=1
```

```
PF_SF_BAR_SIZE=8 TX_SCHEDULER_BURST=15
```

> **Note**
>
> When using 64KB pagesize OS, `PF_SF_BAR_SIZE=10` (instead of 8) should be configured.

- Virtio-blk 250 VFs example (2 queue per VF):

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s
VIRTIO_BLK_EMULATION_ENABLE=1 VIRTIO_BLK_EMULATION_NUM_VF=125
VIRTIO_BLK_EMULATION_NUM_PF=2 VIRTIO_BLK_EMULATION_NUM_MSIX=2
VIRTIO_BLK_EMULATION_NUM_VF_MSIX=2
```

- Virtio-blk 1000 VFs example (2 queue per VF):

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s
VIRTIO_BLK_EMULATION_ENABLE=1 VIRTIO_BLK_EMULATION_NUM_VF=250
VIRTIO_BLK_EMULATION_NUM_PF=4 VIRTIO_BLK_EMULATION_NUM_MSIX=2
VIRTIO_NET_EMULATION_ENABLE=0 NUM_OF_VFS=0
PCI_SWITCH_EMULATION_ENABLE=0
VIRTIO_BLK_EMULATION_NUM_VF_MSIX=2
```

- NVMe 250 VFs example (2 IO-queue per VF):

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s
NVME_EMULATION_ENABLE=1 NVME_EMULATION_NUM_VF=125
```

```
NVME_EMULATION_NUM_PF=2 NVME_EMULATION_NUM_MSIX=2
NVME_EMULATION_NUM_VF_MSIX=2
```

Hot-plug Firmware Configuration

Once enabling PCIe switch emulation, BlueField can support up to 31 hotplug NVMe/Virtio-blk functions. "`PCI_SWITCH_EMULATION_NUM_PORT`-1" hot-plugged PCIe functions. These slots are shared among all DPU users and applications and may hold hot-plugged devices of type NVMe, virtio-blk, virtio-fs, or others (e.g., virtio-net).

To enable PCIe switch emulation and determine the number of hot-plugged ports to be used:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s
PCI_SWITCH_EMULATION_ENABLE=1 PCI_SWITCH_EMULATION_NUM_PORT=32
```

`PCI_SWITCH_EMULATION_NUM_PORT` equals 1 + the number of hot-plugged PCIe functions.

For additional information regarding hot plugging a device, refer to section "Hotplugged PCIe Functions Management".

Note

Hotplug is not guaranteed to work on AMD machines.

Note

Enabling PCI_SWITCH_EMULATION_ENABLE could potentially impact SR-IOV capabilities on Intel and AMD machines.

Note

Currently, hotplug PFs do not support SR-IOV.

UEFI Firmware Configuration

To use the storage emulation as a boot device, it is recommended to use the DPU's embedded UEFI expansion ROM drivers to be used by the UEFI instead of the original vendor's BIOS ones.

To enable UEFI drivers:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s
EXP_ROM_VIRTIO_BLK_UEFI_x86_ENABLE=1
EXP_ROM_NVME_UEFI_x86_ENABLE=1
```

DPU Configurations

DPA Core Mask

The d ata path accelerator (DPA) is a cluster of 16 cores with 16 execution units (EUs) per core.

Note

Only EUs 0-170 are available for SNAP.

SNAP supports reservation of DPA EUs for NVMe or virtio-blk controllers. By default, all available EUs, 0-170, are shared between NVMe, virtio-blk, and other DPA applications on the system (e.g., virtio-net).

To assign specific set of EUs, set the following environment variable:

- For NVMe:

```
dpa_nvme_core_mask=0x<EU_mask>
```

- For virtio-blk DPU mode:

```
dpa_helper_core_mask=0x<EU_mask>
```

- For virtio-blk DPA mode:

```
dpa_virtq_split_core_mask=0x<EU_mask>
```

The core mask must contain valid hexadecimal digits (it is parsed right to left). For example, `dpa_virtq_split_core_mask=0xff00` sets 8 EUs (i.e., EUs 8-16).

Note

There is a hardware limit of 128 queues (threads) per DPA EU.

SNAP Container Deployment

SNAP container is available on the DOCA SNAP NVIDIA NGC catalog page.

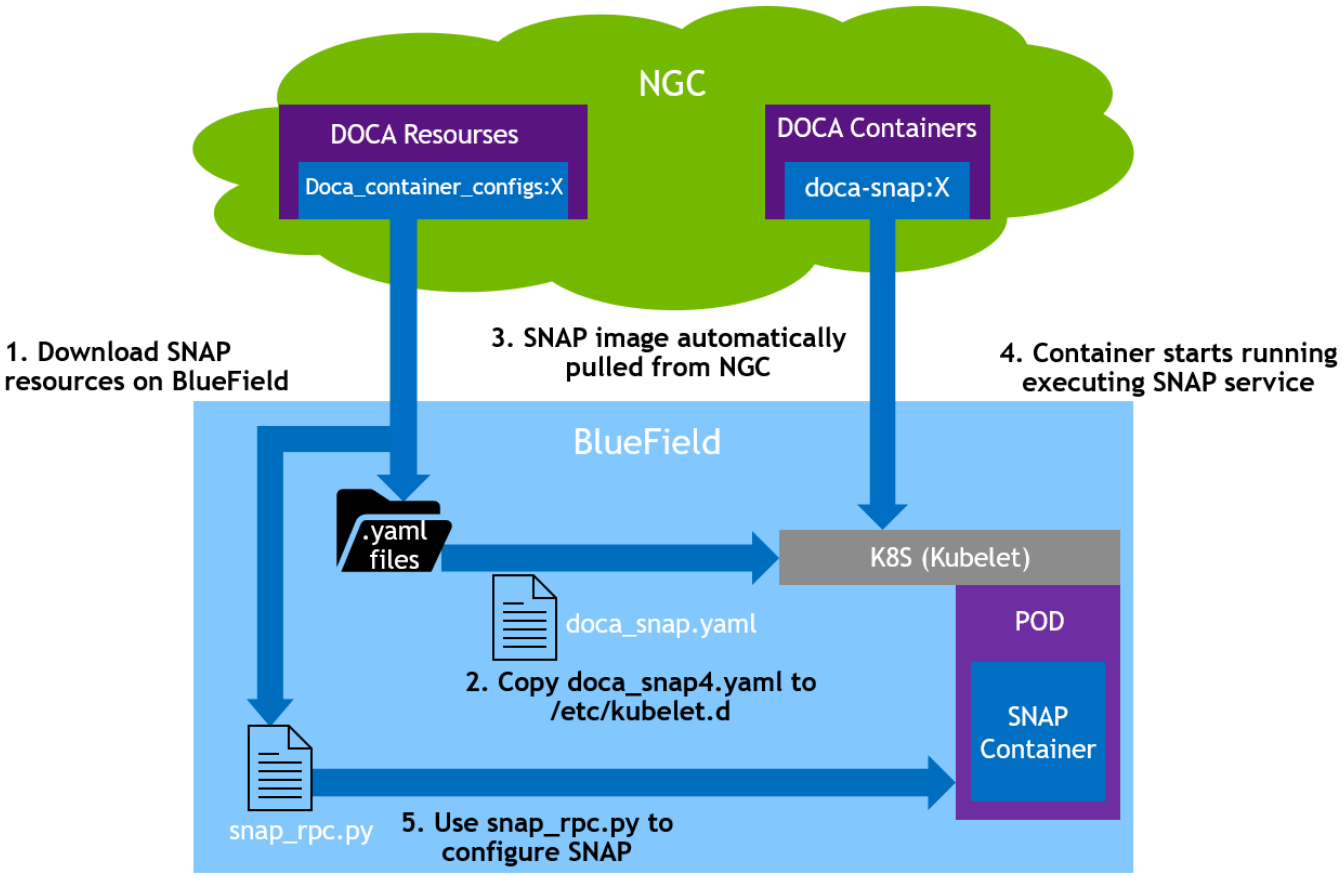
SNAP container deployment on top of the BlueField DPU requires the following sequence:

1. Setup preparation and SNAP resource download for container deployment. See section "Preparation Steps" for details.

2. Adjust the `doca_snap.yaml` for advanced configuration if needed according to section "Adjusting YAML Configuration".

3. Deploy the container. The image is automatically pulled from NGC. See section "Spawning SNAP Container" for details.

The following is an example of the SNAP container setup.



Preparation Steps

Step 1: Allocate Hugepages

Generic

Allocate `4GiB` hugepages for the SNAP container according to the DPU OS's `Hugepagesize` value:

1. Query the `Hugepagesize` value:

   ```
   [dpu] grep Hugepagesize /proc/meminfo
   ```

   In Ubuntu, the value should be 2048KB. In CentOS 8.x, the value should be 524288KB.

2. Append the following line to the end of the `/etc/sysctl.conf` file:

   - For Ubuntu or CentOS 7.x setups (i.e., `Hugepagesize` = 2048 kB):

     ```
     vm.nr_hugepages = 2048
     ```

   - For CentOS 8.x setups (i.e., `Hugepagesize` = 524288 kB):

     ```
     vm.nr_hugepages = 8
     ```

3. Run the following:

   ```
   [dpu] sysctl --system
   ```

Note

If live upgrade is utilized in this deployment, it is necessary to allocate twice the amount of resources listed above for the upgraded

container.

> **Warning**
>
> If other applications are running concurrently within the setup and are consuming hugepages, make sure to allocate additional hugepages beyond the amount described in this section for those applications.
>
> When deploying SNAP with a high scale of connections (i.e., disks 500 or more), the default allocation of hugepages (4GiB) becomes insufficient. This shortage of hugepages can be identified through error messages in the SNAP and SPDK layers. These error messages typically indicate failures in creating or modifying QPs or other objects.

Step 2: Create nvda_snap Folder

The folder `/etc/nvda_snap` is used by the container for automatic configuration after deployment.

Downloading YAML Configuration

The `.yaml` file configuration for the SNAP container is `doca_snap.yaml`. The download command of the `.yaml` file can be found on the DOCA SNAP NGC page.

> **Note**
>
> Internet connectivity is necessary for downloading SNAP resources.
> To deploy the container on DPUs without Internet connectivity, refer

to appendix "Deploying Container on Setups Without Internet Connectivity".

Adjusting YAML Configuration

The `.yaml` file can easily be edited for advanced configuration.

- The SNAP `.yaml` file is configured by default to support Ubuntu setups (i.e., `Hugepagesize` = 2048 kB) by using hugepages-2Mi.

  To support other setups, edit the hugepages section according to the DPU OS's relevant `Hugepagesize` value. For example, to support CentOS 8.x configure `Hugepagesize` to 512MB:

  ```
  limits:
     hugepages-512Mi: "<number-of-hugepages>Gi"
  ```

  > **Note**
  >
  > When deploying SNAP with a large number of controllers (500 or more), the default allocation of hugepages (4GB) may become insufficient. This shortage of hugepages can be identified through error messages, typically indicate failures in creating or modifying QPs or other objects. In these cases, more hugepages needed.

- The following example edits the `.yaml` file to request 16 CPU cores for the SNAP container and 4Gi memory and 4Gi hugepages:

  ```
  resources:
    requests:
      memory: "2Gi"
  ```

```
        hugepages-2Mi: "4Gi"
        cpu: "8"
      limits:
        memory: "4Gi"
        hugepages-2Mi: "4Gi"
        cpu: "16"
env:
  - name: APP_ARGS
    value: "-m 0xffff"
```

Note

If all BlueField-3 cores are requested, the user must verify no other containers are in conflict over the CPU resources.

Note

When running the Virtio-fs service with a large number of cores, it is necessary to increase the number of IO buffers in SPDK. For example, to run with 16 cores, the size of the large IO buffer pool must be set to at least 4095. This can be configured by adding the RPC command `iobuf_set_options --large-pool-count 4095` to `spdk_rpc_init.conf` under `/etc/nvda_snap`. Depending on the scale and SPDK subsystems in use other SPDK configuration parameters may need to be adjusted. Refer to SPDK documentation for more details.

- To automatically configure SNAP container upon deployment, edit the files below according to the use case. During bring-up, SNAP will forward the content of these files into the appropriate RPC script, whether SPDK RPCs or SNAP RPCs. Ensure that the required RPCs for your use case are included.

1. Add `spdk_rpc_init.conf` file under `/etc/nvda_snap/`. The file includes the required SPDK RPCs. File example:

```
bdev_malloc_create 64 512
```

2. Add `snap_rpc_init.conf` file under `/etc/nvda_snap/`. The file includes the required SPDK RPCs.

   Virtio-blk file example:

```
virtio_blk_controller_create --pf_id 0 --bdev Malloc0
```

   NVMe file example:

```
nvme_subsystem_create --nqn nqn.2022-10.io.nvda.nvme:0
nvme_namespace_create -b Malloc0 -n 1 --nqn nqn.2022-
10.io.nvda.nvme:0 --uuid 16dab065-ddc9-8a7a-108e-
9a489254a839
nvme_controller_create --nqn nqn.2022-10.io.nvda.nvme:0
--ctrl NVMeCtrl1 --pf_id 0 --suspended
nvme_controller_attach_ns -c NVMeCtrl1 -n 1
nvme_controller_resume -c NVMeCtrl1
```

3. Edit the `.yaml` file accordingly (uncomment):

```
env:
  - name: SPDK_RPC_INIT_CONF
    value: "/etc/nvda_snap/spdk_rpc_init.conf"
  - name: SNAP_RPC_INIT_CONF
```

```
        value: "/etc/nvda_snap/snap_rpc_init.conf"
```

> **Note**
>
> It is user responsibility to make sure SNAP configuration matches firmware configuration. That is, an emulated controller must be opened on all existing (static/hotplug) emulated PCIe functions (either through automatic or manual configuration). A PCIe function without a supporting controller is considered malfunctioned, and host behavior with it is anomalous.

Spawning SNAP Container

Run the Kubernetes tool:

```
[dpu] systemctl restart containerd
[dpu] systemctl restart kubelet
[dpu] systemctl enable kubelet
[dpu] systemctl enable containerd
```

Copy the updated `doca_snap.yaml` file to the `/etc/kubelet.d` directory.

Kubelet automatically pulls the container image from NGC described in the YAML file and spawns a pod executing the container.

```
cp doca_snap.yaml /etc/kubelet.d/
```

The SNAP service starts initialization immediately, which may take a few seconds. To verify SNAP is running:

- Look for the message "SNAP Service running successfully" in the log
- Send `spdk_rpc.py spdk_get_version` to confirm whether SNAP is operational or still initializing

Debug and Log

View currently active pods, and their IDs (it might take up to 20 seconds for the pod to start):

```
crictl pods
```

Example output:

```
POD ID              CREATED               STATE         NAME
0379ac2c4f34c       About a minute ago    Ready         snap
```

View currently active containers, and their IDs:

```
crictl ps
```

View existing containers and their ID:

```
crictl ps -a
```

Examine the logs of a given container (SNAP logs):

```
crictl logs <container_id>
```

Examine the kubelet logs if something does not work as expected:

```
journalctl -u kubelet
```

The container log file is saved automatically by Kubelet under `/var/log/containers`.

Refer to section "RPC Log History" for more logging information.

Stop, Start, Restart SNAP Container

SNAP binaries are deployed within a Docker container as SNAP service, which is managed as a supervisorctl service. Supervisorctl provides a layer of control and configuration for various deployment options.

- In the event of a SNAP crash or restart, supervisorctl detects the action and waits for the exited process to release its resources. It then deploys a new SNAP process within the same container, which initiates a recovery flow to replace the terminated process.
- In the event of a container crash or restart, kubeletclt detects the action and waits for the exited container to release its resources. It then deploys a new container with a new SNAP process, which initiates a recovery flow to replace the terminated process.

Note

After containers crash or exit, the kubelet restarts them with an exponential back-off delay (10s, 20s, 40s, etc.) which is capped at five minutes. Once a container has run for 10 minutes without an issue, the kubelet resets the restart back-off timer for that container.

Restarting the SNAP service without restarting the container helps avoid the occurrence of back-off delays.

Different SNAP Termination Options

Container Termination

- To kill the container, remove the `.yaml` file form `/etc/kubelet.d/`. To start the container, `cp` the `.yaml` file back to the same path:

```
cp doca_snap.yaml /etc/kubelet.d/
```

- To restart the container (with sig-term) using `crictl`, use the `-t` (timeout) option:

```
crictl stop -t 10 <container-id>
```

SNAP Process Termination

To restart the SNAP service without restarting the entire container, user can either use the `supervictl` tool to restart the SNAP service or terminate the SNAP service process on the DPU. Different signals correspond to different termination behaviors. For example:

- Restart sends SIGTERM

```
crictl exec -it $(crictl ps -s running -q --name snap)
supervisorctl restart snap
```

- pkill sends SIGKILL

```
pkill -9 -f snap
```

> **Note**
>
> SNAP service termination may take time as it releases all allocated resources. The duration depends on the scale of the use case and any other applications sharing resources with SNAP.

SNAP Process Fast Restart

The duration can be improved by configuring supervisorctl to give the exited SNAP process a shorter or zero termination time when using `supervisorctl restart snap` .

This causes the new process to start up while the old process's resources are still being freed by the kernel.

The user must ensure that the hugepage allocation is sufficient to accommodate both processes running in parallel. To modify the time SNAP takes to exit, the user should use the relevant environment variable SUPERVISOR_STOPWAITSECS.

- To restart the SNAP service without restarting the container, kill the SNAP service process on the DPU. Different signals can be used for different termination options. For example:

  ```
  pkill -9 -f snap
  ```

SNAP Source Package Deployment

System Preparation

Allocate `4GiB` hugepages for the SNAP container according to the DPU OS's `Hugepagesize` value:

1. Query the `Hugepagesize` value:

   ```
   [dpu] grep Hugepagesize /proc/meminfo
   ```

   In Ubuntu, the value should be 2048KB. In CentOS 8.x, the value should be 524288KB.

2. Append the following line to the end of the `/etc/sysctl.conf` file:

   - For Ubuntu or CentOS 7.x setups (i.e., `Hugepagesize` = 2048 kB):

     ```
     vm.nr_hugepages = 2048
     ```

   - For CentOS 8.x setups (i.e., `Hugepagesize` = 524288 kB):

     ```
     vm.nr_hugepages = 8
     ```

3. Run the following:

   ```
   [dpu] sysctl --system
   ```

Note

If live upgrade is utilized in this deployment, it is necessary to allocate twice the amount of resources listed above for the upgraded container.

⚠ Warning

If other applications are running concurrently within the setup and are consuming hugepages, make sure to allocate additional hugepages beyond the amount described in this section for those applications.

When deploying SNAP with a high scale of connections (i.e., disks 500 or more), the default allocation of hugepages (4GiB) becomes insufficient. This shortage of hugepages can be identified through error messages in the SNAP and SPDK layers. These error messages typically indicate failures in creating or modifying QPs or other objects.

Installing SNAP Source Package

Download the SNAP source package provided by NVIDIA on BlueField (BF).

Install the package:

- For Ubuntu, run:

```
dpkg -i snap-sources_<version>_arm64.*
```

- For CentOS, run:

```
rpm -i snap-sources_<version>_arm64.*
```

Build, Compile, and Install Sources

> **Note**
>
> To build SNAP with a custom SPDK, see section "Replace the BFB SPDK".

1. Move to the sources folder. Run:

```
cd /opt/nvidia/nvda_snap/src/
```

2. Build the sources. Run:

```
meson /tmp/build
```

3. Compile the sources. Run:

```
meson compile -C /tmp/build
```

4. Install the sources. Run:

```
meson install -C /tmp/build
```

Configure SNAP Environment Variables

To config the environment variables of SNAP, run:

```
source
/opt/nvidia/nvda_snap/src/scripts/set_environment_variables.sh
```

Run SNAP Service

```
/opt/nvidia/nvda_snap/bin/snap_service
```

Replace the BFB SPDK (Optional)

Start with installing SPDK.

> **Note**
>
> For legacy SPDK versions (e.g., SPDK 19.04) see appendix "Install Legacy SPDK".

To build SNAP with a custom SPDK, instead of following the basic build steps, perform the following:

1. Move to the sources folder. Run:

   ```
   cd /opt/nvidia/nvda_snap/src/
   ```

2. Build the sources with spdk-compat enabled and provide the path to the custom SPDK. Run:

```
meson setup /tmp/build -Denable-spdk-compat=true -
Dsnap_spdk_prefix=</path/to/custom/spdk>
```

3. Compile the sources. Run:

```
meson compile -C /tmp/build
```

4. Install the sources. Run:

```
meson install -C /tmp/build
```

5. Configure SNAP env variables and run SNAP service as explained in section "Configure SNAP Environment Variables" and "Run SNAP Service".

Build with Debug Prints Enabled (Optional)

Instead of the basic build steps, perform the following:

1. Move to the sources folder. Run:

```
cd /opt/nvidia/nvda_snap/src/
```

2. Build the sources with `buildtype=debug`. Run:

```
meson --buildtype=debug /tmp/build
```

3. Compile the sources. Run:

```
meson compile -C /tmp/build
```

4. Install the sources. Run:

```
meson install -C /tmp/build
```

5. Configure SNAP env variables and run SNAP service as explained in section "Configure SNAP Environment Variables" and "Run SNAP Service".

Automate SNAP Configuration (Optional)

The script `run_snap.sh` automates SNAP deployment. Users must modify the following files to align with their setup. The configuration files are in the same format explained in the "adjust the YAML section". If different directories are utilized by the user, edits must be made to `run_snap.sh` accordingly:

1. Edit SNAP env variables in:

```
/opt/nvidia/nvda_snap/bin/set_environment_variables.sh
```

2. Edit SPDK initialization RPCs calls:

```
/opt/nvidia/nvda_snap/bin/spdk_rpc_init.conf
```

3. Edit SNAP initialization RPCs calls:

```
/opt/nvidia/nvda_snap/bin/snap_rpc_init.conf
```

Run the script:

```
/opt/nvidia/nvda_snap/bin/run_snap.sh
```

SNAP Environment Variables

Supported Environment Variables

Name	Description	Default
`SNAP_RDMA_ZCOPY_ENABLE`	Enable/disable RDMA zero-copy transport type. For more info refer to section "Zero Copy (SNAP-direct)".	1 (enabled)
`NVME_BDEV_RESET_ENABLE`	It is recommended that namespaces discovered from the same remote target are not shared by different PCIe emulations. If it is desirable to do that, users should set the variable `NVME_BDEV_RESET_ENABLE` to `0`. **⚠ Warning** By doing so, the user must ensure that SPDK bdev always completes IOs (either with success or failure) in a reasonable time. Otherwise, the system may stall until all IOs return.	1 (enabled)
`VBLK_RECOVERY_SHM`	Enable/disable virtio-blk recovery using shared memory files. This allows recovering without using `--force_in_order`.	1 (enabled)

Name	Description	Default
`SNAP_EMULATION_MANAGER`	The name of the RDMA device configured to have emulation management capabilities. If the variable is not defined (default), SNAP searches through all available devices to find the emulation manager (which may slow down initialization process). Unless configured otherwise, SNAP selects the first ECPF (i.e., "mlx5_0") as the emulation manager.	NULL (not configured)
`VIRTIO_EMU_PROVIDER`	Configure the Virtio-blk data path Provider - options: "dpu" and "dpa".	"dpu"
`IO_CORE_DISTRIBUTION_POLICY`	Support for IO/core Multiplexer (MP) capabilities in DPA provider for NVMe. For more info refer to section "NVMe DPA multiplexer (MP) data path Provider" - options "weighted_rr", "none"	"none" (disabled)
`SUPERVISOR_STOPWAITSECS`	SNAP process termination time when using `supervisorctl restart snap`	10 (sec)

YAML Configuration

To change the SNAP environment variables add the following to the `doca_snap.yaml` and continue from section "Adjusting YAML Configuration".

```
env:
      - name: VARIABLE_NAME
        value: "VALUE"
```

For example:

```
env:
```

```
      - name: SNAP_RDMA_ZCOPY_ENABLE
        value: "1"
```

Source Package Configuration

To change the SNAP environment variables:

1. Add/modify the configuration under `scripts/set_environment_variables.sh` .
2. Rerun:

   ```
   source scripts/set_environment_variables.sh
   ```

3. Rerun SNAP.

SNAP RPC Commands

Remote procedure call (RPC) protocol is used to control the SNAP service. NVMe/virtio-blk SNAP, like other standard SPDK applications, supports JSON-based RPC protocol commands to control any resources and create, delete, query, or modify commands easily from CLI.

SNAP supports all standard SPDK RPC commands in addition to an extended SNAP-specific command set. SPDK standard commands are executed by the `spdk_rpc.py` tool while the SNAP-specific command set extension is executed by the `snap_rpc.py` tool.

Full `spdk_rpc.py` command set documentation can be found in the SPDK official documentation site.

Full `snap_rpc.py` extended commands are detailed further down in this chapter.

Using JSON-based RPC Protocol

The JSON-based RPC protocol can be used via the `snap_rpc.py` script that is installed inside the SNAP container and can be accessed using `crictl` tool.

> **Info**
>
> The SNAP container is CRI-compatible.

- To query the active container ID:

```
crictl ps -s running -q --name snap
```

- To post RPCs to the container using `crictl`:

```
crictl exec <container-id> snap_rpc.py <RPC-method>
```

  For example:

```
crictl exec 0379ac2c4f34c snap_rpc.py emulation_function_list
```

  In addition, an alias can be used:

```
alias snap_rpc.py="crictl ps -s running -q --name snap |
xargs -I{} crictl exec -i {} snap_rpc.py "
alias spdk_rpc.py="crictl ps -s running -q --name snap |
xargs -I{} crictl exec -i {} spdk_rpc.py "
```

- To open a bash shell to the container that can be used to post RPCs:

```
crictl exec -it <container-id> bash
```

Log Management

snap_log_level_set

SNAP allows dynamically changing the log level of the logger backend using the `snap_log_level_set`. Any log under the requested level is shown.

Parameter	Mandatory?	Type	Description
`level`	Yes	Number	Log level • 0 – Critical • 1 – Error • 2 – Warning • 3 – Info • 4 – Debug • 5 – Trace

PCIe Function Management

Emulated PCIe functions are managed through IB devices called emulation managers. Emulation managers are ordinary IB devices with special privileges to control PCIe communication and device emulations towards the host OS.

SNAP queries an emulation manager that supports the requested set of capabilities.

The emulation manager holds a list of the emulated PCIe functions it controls. PCIe functions may be approached later in 3 ways:

- `vuid` – recommended as it is guaranteed to remain constant (see appendix "PCIe BDF to VUID Translation" for details)

- `vhca_id`
- Function index (i.e., `pf_id` or `vf_id`)

emulation_function_list

`emulation_function_list` lists all existing functions.

The following is an example response for the `emulation_function_list` command:

```
[
  {
    "hotplugged": true,
    "hotplug state": "POWER_ON",
    "emulation_type": "VBLK",
    "pf_index": 0,
    "pci_bdf": "87:00.0",
    "vhca_id": 5,
    "vuid": "MT2306XZ009TVBLKS1D0F0",
    "ctrl_id": "VblkCtrl1",
    "num_vfs": 0,
    "vfs": []
  }
]
```

> **Note**
>
> Use `-a` or `--all`, to show all inactive VF functions.

SNAP supports 2 types of PCIe functions:

- Static functions – PCIe functions configured at the firmware configuration stage (physical and virtual). Refer to appendix "DPU Firmware Configuration" for additional

information.

- Hot-pluggable functions – PCIe functions configured dynamically at runtime. Users can add detachable functions. Refer to section "Hot-pluggable PCIe Functions Management" for additional information.

Hot-pluggable PCIe Functions Management

Hotplug PCIe functions are configured dynamically at runtime using RPCs. Once a new PCIe function is hot plugged, it appears in the host's PCIe device list and remains persistent until explicitly unplugged or the system undergoes a cold reboot. Importantly, this persistence continues even if the SNAP process terminates. Therefore, it is advised not to include hotplug/hotunplug actions in automatic initialization scripts (e.g., `snap_rpc_init.conf`).

> **Note**
>
> Hotplug PFs do not support SR-IOV.

Two-step PCIe Hotplug

The following RPC commands are used to dynamically add or remove PCIe PFs (i.e., hot-plugged functions) in the DPU application.

Once a PCIe function is created (via `virtio_blk_function_create`), it is accessible and manageable within the DPU application but is not immediately visible to the host OS/kernel. This differs from the legacy API, where creation and host exposure occurs simultaneously. Instead, exposing or hiding PCIe functions to the host OS is managed by separate RPC commands (`virtio_blk_controller_hotplug` and `virtio_blk_controller_hotunplug`). After hot unplugging, the function can be safely removed from the DPU (using `virtio_blk_function_destroy`).

A key advantage of this approach is the ability to pre-configure a controller on the function, enabling it to serve the host driver as soon as it is exposed. In fact, users must

create a controller to use the `virtio_blk_controller_hotplug` API, which is required to make the function visible to the host OS.

Command	Description
virtio_blk_function_create	Create a new virtio-blk emulation function
virtio_blk_controller_hotplug	Exposes (hot plugs) the emulation function to the host OS
virtio_blk_controller_hotunplug	Removes (hot unplugs) the emulation function from the host OS
virtio_blk_function_destroy	Delete an existing virtio-blk emulation function

virtio_blk_function_create

Create a new virtio-blk emulation function.

Command parameters:

Parameter	Mandatory?	Type	Description
`manager`	No	String	Emulation manager to manage hotplug function (unused)

virtio_blk_function_destroy

Delete an existing virtio-blk emulation function.

Command parameters:

Parameter	Mandatory?	Type	Description
`vuid`	Yes	String	Identifier of the hotplugged function to delete

virtio_blk_controller_hotplug

Exposes (hot plugs) the emulation function to the host OS.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller to expose to the host OS
`wait_for_done`	No	Bool	Block until host discovers and acknowledges the new command
`timeout`	No	int	Time (in msecs) to wait until giving up. Only valid when `wait_for_done` is used.

virtio_blk_controller_hotunplug

Removes (hot unplugs) the emulation function from the host OS.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller to expose to the host OS
`wait_for_done`	No	Bool	Block until host identifies and removes the function

Note

The non-legacy API is not supported yet for NVMe protocol.

Note

When not using `wait_for_done` approach, it is the user's responsibility to verify host identifies the new hotplugged function.

This can be done by querying the `pci_hotplug_state` parameter in `emulation_function_list` RPC output.

Two-step PCIe Hotplug/Unplug Example

```
# Bringup
spdk_rpc.py bdev_nvme_attach_controller -b nvme0 -t rdma -a
1.1.1.1 -f ipv4 -s 4420 -n nqn.2022-10.io.nvda.nvme:swx-storage
snap_rpc.py virtio_blk_function_create
snap_rpc.py virtio_blk_controller_create --vuid
MT2114X12200VBLKS1D0F0 --bdev nvme0n1
snap_rpc.py virtio_blk_controller_hotplug -c VblkCtrl1

# Cleanup
snap_rpc.py virtio_blk_controller_hotunplug -c VblkCtrl1
snap_rpc.py virtio_blk_controller_destroy -c VblkCtrl1
snap_rpc.py virtio_blk_function_destroy --vuid
MT2114X12200VBLKS1D0F0
spdk_rpc.py bdev_nvme_detach_controller nvme0
```

(Deprecated) Legacy API

Hotplug Legacy Commands

The following commands hot plug a new PCIe function to the system.

After a new PCIe function is plugged, it is immediately shown on the host's PCIe devices list until it is either explicitly unplugged or the system goes through a cold reboot. Therefore, it is user responsibility to open a controller instance to manage the new function immediately after a function's creation. Keeping a hotplugged function without a matching controller to manage may cause anomalous behavior on the host OS driver.

Command	Description
virtio_blk_emulation_device_attach	Attach virtio-blk emulation function
nvme_emulation_device_attach	Attach NVMe emulation function

virtio_blk_emulation_device_attach

Attach virtio-blk emulation function.

Command parameters:

Parameter	Mandatory?	Type	Description
`id`	No	Number	Device ID
`vid`	No	Number	Vendor ID
`ssid`	No	Number	Subsystem device ID
`ssvid`	No	Number	Subsystem vendor ID
`revid`	No	Number	Revision ID
`class_code`	No	Number	Class code
`num_msix`	No	Number	MSI-X table size
`total_vf`	No	Number	Maximal number of VFs allowed

Parameter	Mandatory?	Type	Description
`bdev`	No	String	Block device to use as backend
`num_queues`	No	Number	Number of IO queues (default 1, range 1-62). **Note** The actual number of queues is limited by the number of queues supported by the hardware. **Tip** It is recommended that the number of MSIX be greater than the number of IO queues (1 is used for the config interrupt).
`queue_depth`	No	Number	Queue depth (default 256, range 1-256) **Note** It is only possible to modify the queue depth if the driver is not loaded.
`transitional_device`	No	Boolean	Transitional device support. See section "Virtio-blk Transitional Device Support" for more details.
`dbg_bdev_type`	No	Boolean	N/A – not supported

nvme_emulation_device_attach

Attach NVMe emulation function.

Command parameters:

Parameter	Mandatory?	Type	Description
`id`	No	Number	Device ID
`vid`	No	Number	Vendor ID
`ssid`	No	Number	Subsystem device ID
`ssvid`	No	Number	Subsystem vendor ID
`revid`	No	Number	Revision ID
`class_code`	No	Number	Class code
`num_msix`	No	Number	MSI-X table size
`total_vf`	No	Number	Maximal number of VFs allowed
`num_queues`	No	Number	Number of IO queues (default 31, range 1-31). **Note**

Parameter	Mandatory?	Type	Description
			The actual number of queues is limited by the number of queues supported by the hardware. **Tip** It is recommended that the number of MSIX be greater than the number of IO queues (1 is used for the config interrupt).
`version`	No	String	Specification version (currently only `1.4` is supported)

Hot Unplug Legacy Commands

The following commands hot-unplug a PCIe function from the system in 2 steps:

	Command	Description
1	emulation_device_detach_prepare	Prepare emulation function to be detached
2	emulation_device_detach	Detach emulation function

emulation_device_detach_prepare

This is the first step for detaching an emulation device. It prepares the system to detach a hot plugged emulation function. In case of success, the host's hotplug device state changes and you may safely proceed to the `emulation_device_detach` command.

The controller attached to the emulation function must be created and active when executing this command.

Command parameters:

Parameter	Mandatory?	Type	Description
`vhca_id`	No	Number	vHCA ID of PCIe function
`vuid`	No	String	PCIe device VUID
`ctrl`	No	String	Controller ID

> **Note**
>
> At least one identifier must be provided to describe the PCIe function to be detached.

emulation_device_detach

This is the second step which completes detaching of the hotplugged emulation function. If the detach preparation times out, you may perform a surprise unplug using `--force` with the command.

> **Note**
>
> The driver must be unprobed, otherwise errors may occur.

Command parameters:

Parameter	Mandatory?	Type	Description
`vhca_id`	No	Number	vHCA ID of PCIe function
`vuid`	no	String	PCIe device VUID
`force`	No	Boolean	Detach with failed preparation

> **Note**
>
> At least one identifier must be provided to describe the PCIe function to be detached.

Virtio-blk Hot Plug/Unplug Example

```
// Bringup
spdk_rpc.py bdev_nvme_attach_controller -b nvme0 -t rdma -a
1.1.1.1 -f ipv4 -s 4420 -n nqn.2022-10.io.nvda.nvme:swx-storage
snap_rpc.py virtio_blk_emulation_device_attach
snap_rpc.py virtio_blk_controller_create --vuid
MT2114X12200VBLKS1D0F0 --bdev nvme0n1

// Cleanup
snap_rpc.py emulation_device_detach_prepare --vuid
MT2114X12200VBLKS1D0F0
snap_rpc.py virtio_blk_controller_destroy -c VblkCtrl1
snap_rpc.py emulation_device_detach --vuid MT2114X12200VBLKS1D0F0
spdk_rpc.py bdev_nvme_detach_controller nvme0
```

(Deprecated) SPDK Bdev Management

The following RPCs are deprecated and are no longer supported:

- `spdk_bdev_create`
- `spdk_bdev_destroy`

- `bdev_list`

These RPCs were optional. If not performed, SNAP would automatically generate SNAP block devices (bdevs).

Virtio-blk Emulation Management

Virtio-blk emulation is a storage protocol belonging to the virtio family of devices. These devices are found in virtual environments yet by design look like physical devices to the user within the virtual machine.

Each virtio-blk device (e.g., virtio-blk PCIe entry) exposed to the host, whether it is PF or VF, must be backed by a virtio-blk controller.

> **Note**
>
> Virtio-blk limitations:
>
> - Probing a virtio-blk driver on the host without an already functioning virtio-blk controller may cause the host to hang until such controller is opened successfully (no timeout mechanism exists).
> - Upon creation of a virtio-blk controller, a backend device must already exist.

Virtio-blk Emulation Management Commands

Command	Description
virtio_blk_controller_create	Create new virtio-blk SNAP controller
virtio_blk_controller_destroy	Destroy virtio-blk SNAP controller
virtio_blk_controller_suspend	Suspend virtio-blk SNAP controller
virtio_blk_controller_resume	Resume virtio-blk SNAP controller

Command	Description
virtio_blk_controller_bdev_attach	Attach bdev to virtio-blk SNAP controller
virtio_blk_controller_bdev_detach	Detach bdev from virtio-blk SNAP controller
virtio_blk_controller_list	Virtio-blk SNAP controller list
virtio_blk_controller_modify	Virtio-blk controller parameters modification
virtio_blk_controller_dbg_io_stats_get	Get virtio-blk SNAP controller IO stats
virtio_blk_controller_dbg_debug_stats_get	Get virtio-blk SNAP controller debug stats
virtio_blk_controller_state_save	Save state of the suspended virtio-blk SNAP controller
virtio_blk_controller_state_restore	Restore state of the suspended virtio-blk SNAP controller
virtio_blk_controller_vfs_msix_reclaim	Reclaim virtio-blk SNAP controller VFs MSIX for the free MSIX pool. Valid only for PFs.

virtio_blk_controller_create

Create a new SNAP-based virtio-blk controller over a specific PCIe function on the host. To specify the PCIe function to open a controller upon must be provided as described in section "PCIe Function Management":

1. `vuid` (recommended as it is guaranteed to remain constant).
2. `vhca_id.`
3. Function index – `pf_id`, `vf_id`.

The mapping for `pci_index` can be queried by running `emulation_function_list`.

Command parameters:

Parameter	Mandatory?	Type	Description
`vuid`	No	String	PCIe device VUID
`vhca_id`	No	Number	vHCA ID of PCIe function
`pf_id`	No	Number	PCIe PF index to start emulation on
`vf_id`	No	Number	PCIe VF index to start emulation on (if the controller is meant to be opened on a VF)
`pci_bdf`	No	String	PCIe device BDF
`ctrl`	No	String	Controller ID
`num_queues`	No	Number	Number of IO queues (default 1, range 1-64). **Tip** It is recommended that the number of MSIX be greater than the number of IO queues (1 is used for the config interrupt). Based on effective `num_msix` value (can be queried from `virtio_blk_controller_list` RPC), it can be later aligned using `virtio_blk_controller_modify` RPC).

Parameter	Mandatory?	Type	Description
`queue_size`	No	Number	Queue depth (default 256, range 1-256)
`size_max`	No	Number	Maximal SGE data transfer size (default 65536, range 1–`MAX_UINT16`)
`seg_max`	No	Number	Maximal SGE list length (default 32, range 1-`queue_depth`)
`bdev`	No	String	SNAP SPDK block device to use as backend
`vblk_id`	No	String	Serial number for the controller
`admin_q`	No	0/1	Enables live migration and NVIDIA vDPA
`dynamic_msix`	No	0/1	Dynamic MSIX for SR-IOV VFs on this PF. Only valid for PFs.
`vf_num_msix`	No	Number	Control the number of MSIX tables to associate with this controller. Valid only for VFs (whose parent PF controller is created using the `--dynamic_msix` option) and only when the dynamic MSIX management feature is enabled. Must be an even number ⩾ 2. **Note** This field is mandatory when the VF's MSIX is reclaimed using `virtio_blk_controller_vfs_msix_reclaim`

Parameter	Mandatory?	Type	Description
			or released using `--release_msix` on `virtio_blk_controller_destroy`.
`force_in_order`	No	0/1	Support virtio-blk crash recovery. Enabling this parameter to 1 may impact virtio-blk performance (default is 0). For more information, refer to section "Virtio-blk Crash Recovery".
`indirect_desc`	No	0/1	Enables indirect descriptors support for the controller's virt-queues. **Note** When using the virtio-blk kernel driver, if indirect descriptors are enabled, it is always used by the driver. Using indirect descriptors for all IO traffic patterns may hurt performance in most cases.
`read_only`	No	0/1	Creates read only virtio-blk controller.
`suspended`	No	0/1	Creates controller in suspended state.
`live_update_listener`	No	0/1	Creates controller with the ability to listen for live update notifications via IPC.

Parameter	Mandatory?	Type	Description
`dbg_bdev_type`	No	0/1	N/A – not supported
`dbg_local_optimized`	No	0/1	N/A – not supported

Example response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": "VblkCtrl1"
}
```

virtio_blk_controller_destroy

Destroy a previously created virtio-blk controller. The controller can be uniquely identified by the controller's name as acquired from `virtio_blk_controller_create()`.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name
force	No	Boolean	Force destroying VF controller for SR-IOV

virtio_blk_controller_suspend

While suspended, the controller stops receiving new requests from the host driver and only finishes handling of requests already in flight. All suspended requests (if any) are processed after resume.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name

virtio_blk_controller_resume

After the controller stops receiving new requests from the host driver (i.e., is suspended) and only finishes handling of requests already in flight, the resume command will resume the handling of IOs by the controller.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name

virtio_blk_controller_bdev_attach

Attach the specified bdev into virtIO-blk SNAP controller. It is possible to change the serial ID (using the `vblk_id` parameter) if a new bdev is attached.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name
`bdev`	Yes	String	Block device name
`vblk_id`	No	String	Serial number for controller

virtio_blk_controller_bdev_detach

You may replace the bdev for virtio-blk controller. First, you should detach bdev from the controller. When bdev is detached, the controller stops receiving new requests from the host driver (i.e., is suspended) and finishes handling requests already in flight only.

At this point, you may attach a new bdev or destroy the controller.

When a new bdev is attached, the controller resumes handling all outstanding I/Os.

> **Note**
>
> The block size cannot be changed if the driver is loaded.
>
> bdev may be replaced with a different block size if the driver is not loaded.

> **Note**
>
> A controller with no bdev attached to it is considered a temporary state, in which the controller is not fully operational, and may not respond to some actions requested by the driver.
>
> If there is no imminent intention to call `virtio_blk_controller_bdev_attach`, it is advised to attach a `none` bdev instead. For example:
>
> ```
> snap_rpc.py virtio_blk_controller_bdev_attach -c
> VblkCtrl1 --bdev none --dbg_bdev_type null
> ```

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name

virtio_blk_controller_list

List virtio-blk SNAP controller.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	No	String	Controller name

Example response:

```
{
  "ctrl_id": "VblkCtrl2",
  "vhca_id": 38,
  "num_queues": 4,
  "queue_size": 256,
  "seg_max": 32,
  "size_max": 65536,
  "bdev": "Nvme1",
  "plugged": true,
  "indirect_desc": true,
  "num_msix": 2,
  "min configurable num_msix": 2,
  "max configurable num_msix": 32
}
```

virtio_blk_controller_modify

This function allows user to modify some of the controller's parameters in real-time, after it was already created.

Modifications can only be done when the emulated function is in idle state - thus there is no driver communicating with it.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	No	String	Controller Name
num_queues	No	int	Number of queues for the controller
num_msix	No	int	Number of MSIX to be used for a controller. Relevant only for VF controllers (when dynamic MSIX feature is enabled).

> **Note**
>
> Standard virtio-blk kernel driver currently does not support PCI FLR.
> As such,

virtio_blk_controller_dbg_io_stats_get

Debug counters are per-controller I/O stats that can help knowing the I/O distribution between different queues of the controller and the total I/O received on the controller.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name

Example response:

```
"ctrl_id": "VblkCtrl2",
```

```
"queues": [
  {
    "queue_id": 0,
    "core_id": 0,
    "read_io_count": 19987068,
    "write_io_count": 6319931,
    "flush_io_count": 0
  },
  {
    "queue_id": 1,
    "core_id": 1,
    "read_io_count": 9769556,
    "write_io_count": 3180098,
    "flush_io_count": 0
  }
],
"read_io_count": 29756624,
"write_io_count": 9500029,
"flush_io_count": 0
    }
```


virtio_blk_controller_dbg_debug_stats_get

Debug counters are per-controller debug statistics that can help knowing the controller and queues health and status.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name

Example response:


```
{
```

```
"ctrl_id": "VblkCtrl1",
"queues": [
  {
    "qid": 0,
    "state": "RUNNING",
    "hw_available_index": 6,
    "sw_available_index": 6,
    "hw_used_index": 6,
    "sw_used_index": 6,
    "hw_received_descs": 13,
    "hw_completed_descs": 13
  },
  {
    "qid": 1,
    "state": "RUNNING",
    "hw_available_index": 2,
    "sw_available_index": 2,
    "hw_used_index": 2,
    "sw_used_index": 2,
    "hw_received_descs": 6,
    "hw_completed_descs": 6
  },
  {
    "qid": 2,
    "state": "RUNNING",
    "hw_available_index": 0,
    "sw_available_index": 0,
    "hw_used_index": 0,
    "sw_used_index": 0,
    "hw_received_descs": 4,
    "hw_completed_descs": 4
  },
  {
    "qid": 3,
    "state": "RUNNING",
    "hw_available_index": 0,
```

```
      "sw_available_index": 0,
      "hw_used_index": 0,
      "sw_used_index": 0,
      "hw_received_descs": 3,
      "hw_completed_descs": 3
    }
  ]
}
```

virtio_blk_controller_state_save

Save the state of the suspended virtio-blk SNAP controller.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name
`file_name`	Yes	String	Filename to save state to

virtio_blk_controller_state_restore

Restore the state of the suspended virtio-blk SNAP controller.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name
`file_name`	Yes	String	Filename to save state to

virtio_blk_controller_vfs_msix_reclaim

Reclaim virtio-blk SNAP controller VFs MSIX back to the free MSIX pool. Valid only for PFs.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name

Virtio-blk Configuration Examples

Virtio-blk Configuration for Single Controller

```
spdk_rpc.py bdev_nvme_attach_controller -b nvme0 -t rdma -a
1.1.1.1 -f ipv4 -s 4420 -n nqn.2022-10.io.nvda.nvme:swx-storage
snap_rpc.py virtio_blk_controller_create --vuid
MT2114X12200VBLKS1D0F0 --bdev nvme0n1
```

Virtio-blk Cleanup for Single Controller

```
snap_rpc.py virtio_blk_controller_destroy -c VblkCtrl1
spdk_rpc.py bdev_nvme_detach_controller nvme0
```

Virtio-blk Dynamic Configuration For 125 VFs

1. Update the firmware configuration as described section "SR-IOV Firmware Configuration".

2. Reboot the host.

3. Run:

```
[dpu] spdk_rpc.py bdev_nvme_attach_controller -b nvme0 -t
rdma -a 1.1.1.1 -f ipv4 -s 4420 -n nqn.2022-
10.io.nvda.nvme:swx-storage
[dpu] snap_rpc.py virtio_blk_controller_create --vuid
MT2114X12200VBLKS1D0F0

[host] modprobe -v virtio-pci && modprobe -v virtio-blk
[host] echo 125 >
/sys/bus/pci/devices/0000:86:00.3/sriov_numvfs

[dpu] for i in `seq 0 124`; do snap_rpc.py
virtio_blk_controller_create --pf_id 0 --vf_id $i --bdev
nvme0n1; done;
```

Note

When SR-IOV is enabled, it is recommended to destroy virtio-blk controllers on VFs using the following and not the `virito_blk_controller_destroy` RPC command:

```
[host] echo 0 >
/sys/bus/pci/devices/0000:86:00.3/sriov_numvfs
```

To destroy a single virtio-blk controller, run:

```
[dpu] ./snap_rpc.py -t 1000
virtio_blk_controller_destroy -c VblkCtrl5 –f
```

Virtio-blk Suspend, Resume Example

```
[host] // Run fio
[dpu] snap_rpc.py virtio_blk_controller_suspend -c VBLKCtrl1
[host] // IOs will get suspended
[dpu] snap_rpc.py virtio_blk_controller_resume -c VBLKCtrl1
[host] // fio will resume sending IOs
```

Virtio-blk Bdev Attach, Detach Example

```
[host] // Run fio
[dpu] snap_rpc.py virtio_blk_controller_bdev_detach -c VBLKCtrl1
[host] // Bdev will be detached and IOs will get suspended
[dpu] snap_rpc.py virtio_blk_controller_bdev_attach -c VBLKCtrl1
--bdev null2
[host] // The null2 bdev will be attached into controller and fio
will resume sending IOs
```

Notes

- Virtio-blk protocol controller supports one backend device only
- Virtio-blk protocol does not support administration commands to add backends. Thus, all backend attributes are communicated to the host virtio-blk driver over PCIe BAR and must be accessible during driver probing. Therefore, backends can only be changed once the PCIe function is not in use by any host storage driver.

NVMe Emulation Management

NVMe Subsystem

The NVMe subsystem as described in the NVMe specification is a logical entity which encapsulates sets of NVMe backends (or namespaces) and connections (or controllers). NVMe subsystems are extremely useful when working with multiple NVMe controllers especially when using NVMe VFs. Each NVMe subsystem is defined by its serial number (SN), model number (MN), and qualified name (NQN) after creation.

The RPCs listed in this section control the creation and destruction of NVMe subsystems.

NVMe Namespace

NVMe namespaces are the representors of a continuous range of LBAs in the local/remote storage. Each namespace must be linked to a subsystem and have a unique identifier (NSID) across the entire NVMe subsystem (e.g., 2 namespaces cannot share the same NSID even if they are linked to different controllers).

After creation, NVMe namespaces can be attached to a controller.

> **Note**
>
> SNAP does not currently support shared namespaces between different controllers. So, each namespace should be attached to a single controller.

The SNAP application uses an SPDK block device framework as a backend for its NVMe namespaces. Therefore, they should be configured in advance. For more information about SPDK block devices, see SPDK bdev documentation and Appendix SPDK Configuration.

NVMe Controller

Each NVMe device (e.g., NVMe PCIe entry) exposed to the host, whether it is a PF or VF, must be backed by NVMe controller, which is responsible for all protocol communication with the host's driver.

Every new NVMe controller must also be linked to an NVMe subsystem. After creation, NVMe controllers can be addressed using either their name (e.g., "Nvmectrl1") or both their subsystem NQN and controller ID.

Attaching NVMe Namespace to NVMe Controller

After creating an NVMe controller and an NVMe namespace under the same subsystem, the following method is used to attach the namespace to the controller.

NVMe Emulation Management Command

Command	Description
nvme_subsystem_create	Create NVMe subsystem
nvme_subsystem_destroy	Destroy NVMe subsystem
nvme_subsystem_list	NVMe subsystem list
nvme_namespace_create	Create NVMe namespace
nvme_namespace_destroy	Destroy NVMe namespace
nvme_controller_suspend	Suspend NVMe controller
nvme_controller_resume	Resume NVMe controller
nvme_controller_snapshot_get	Take snapshot of NVMe controller to a file
nvme_namespace_list	NVMe namespace list
nvme_controller_create	Create new NVMe controller
nvme_controller_destroy	Destroy NVMe controller
nvme_controller_list	NVMe controller list
nvme_controller_modify	NVMe controller parameters modification
nvme_controller_attach_ns	Attach NVMe namespace to controller
nvme_controller_detach_ns	Detach NVMe namespace from controller

Command	Description
nvme_controller_vfs_msix_reclaim	Reclaim NVMe SNAP controller VFs MSIX back to free MSIX pool. Valid only for PFs.
nvme_controller_dbg_io_stats_get	Get NVMe controller IO debug stats

nvme_subsystem_create

Create a new NVMe subsystem to be controlled by one or more NVMe SNAP controllers. An NVMe subsystem includes one or more controllers, zero or more namespaces, and one or more ports. An NVMe subsystem may include a non-volatile memory storage medium and an interface between the controller(s) in the NVMe subsystem and non-volatile memory storage medium.

Command parameters:

Parameter	Mandatory?	Type	Description
`nqn`	Yes	String	Subsystem qualified name
`serial_number`	No	String	Subsystem serial number
`model_number`	No	String	Subsystem model number
`nn`	No	Number	Maximal namespace ID allowed in the subsystem (default 0xFFFFFFFE; range 1-0xFFFFFFFE)
`mnan`	No	Number	Maximal number of namespaces allowed in the subsystem (default 1024; range 1-0xFFFFFFFE)

Example request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "nvme_subsystem_create",
  "params": {
```

```
    "nqn": "nqn.2022-10.io.nvda.nvme:0"
  }
}
```

nvme_subsystem_destroy

Destroy (previously created) NVMe SNAP subsystem.

Command parameters:

Parameter	Mandatory?	Type	Description
`nqn`	Yes	String	Subsystem qualified name
`force`	No	Bool	Force the deletion of all the controllers and namespaces under the subsystem

nvme_subsystem_list

List NVMe subsystems.

nvme_namespace_create

Create new NVMe namespaces that represent a continuous range of LBAs in the previously configured bdev. Each namespace must be linked to a subsystem and have a unique identifier (NSID) across the entire NVMe subsystem.

Command parameters:

Parameter	Mandatory?	Type	Description
`nqn`	Yes	String	Subsystem qualified name

Parameter	Mandatory?	Type	Description
`bdev_name`	Yes	String	Block device to use as backend
`nsid`	Yes	Number	Namespace ID
`uuid`	No	Number	Namespace UUID **Note** To safely detach/attach namespaces, the UUID should be provided to force the UUID to remain persistent.
`dbg_bdev_type`	No	string	Bdev plugin to be attached with the controller. Refer to section "Bdev" for more information.

nvme_namespace_destroy

Destroy a previously created NVMe namespaces.

Command parameters:

Parameter	Mandatory?	Type	Description
`nqn`	Yes	String	Subsystem qualified name
`nsid`	Yes	Number	Namespace ID

nvme_namespace_list

List NVMe SNAP namespaces.

Command parameters:

Parameter	Mandatory?	Type	Description
`nqn`	No	String	Subsystem qualified name

nvme_controller_create

Create a new SNAP-based NVMe blk controller over a specific PCIe function on the host.

To specify the PCIe function to open the controller upon, `pci_index` must be provided.

The mapping for `pci_index` can be queried by running `emulation_function_list`.

Command parameters:

Parameter	Mandatory?	Type	Description
`nqn`	Yes	String	Subsystem qualified name
`vuid`	No	Number	VUID of PCIe function
`pf_id`	No	Number	PCIe PF index to start emulation on
`vf_id`	No	Number	PCIe VF index to start emulation on (if the controller is destined to be opened on a VF)
`pci_bdf`	No	String	PCIe BDF to start emulation on
`vhca_id`	No	Number	vHCA ID of PCIe function

Parameter	Mandatory?	Type	Description
`ctrl`	No	Number	Controller ID
`num_queues`	No	Number	Number of IO queues (default 1, range 1-31). **Note** The actual number of queues is limited by the number of queues supported by the hardware. **Tip** It is recommended for the number of MSIX to match be greater than the number of IO queues.
`mdts`	No	Number	MDTS (default 7, range 1-7)
`fw_slots`	No	Number	Maximum number firmware slots (default 4)
`write_zeroes`	No	0/1	Enable the `write_zeroes` optional NVMe command
`compare`	No	0/1	Set the value of the `compare` support bit in the controller
`compare_write`	No	0/1	Set the value of the `compare_write` support bit in the controller **Note**

Parameter	Mandatory?	Type	Description
			During crash recovery, all compare and write commands are expected to fail.
`deallocate_dsm`	No	0/1	Set the value of the `dsm` (dataset management) support bit in the controller. The only `dsm` request currently supported is `deallocate`.
`suspended`	No	0/1	Open the controller in suspended state (requires an additional call to `nvme_controller_resume` before it becomes active) **Note** This is required if NVMe recovery is expected or when creating the controller when the driver is already loaded. Therefore, it is advisable to use it in all scenarios. To resume the controller after attaching namespaces, use `nvme_controller_resume`.
`snapshot`	No	String	Create a controller out of a snapshot file path. Snapshot is previously taken using `nvme_controller_snapshot_get`.
`dynamic_msix`	No	0/1	Enable dynamic MSIX management for the controller (default 0). Applies only for PFs.
`vf_num_msix`	No	Number	Control the number of MSIX tables to associate with this controller. Valid only for VFs (whose parent PF controller is created using the `--dynamic_msix` option) and only when the dynamic MSIX management feature is enabled. **Note**

Parameter	Mandatory?	Type	Description
			This field is mandatory when the VF's MSIX is reclaimed using `nvme_controller_vfs_msix_reclaim` or released using `--release_msix` on `nvme_controller_destroy`.
`admin_only`	No	0/1	Creates NVMe controller with admin queues only (i.e., without IO queues)
`quirks`	No	Number	Bitmask to support buggy drivers which are non-compliant per NVMe specification. • Bit 0 – send "Namespace Attribute Changed" async event, even though it is disabled by the driver during "Set Features" command • Bit 1 – keep sending "Namespace Attribute Changed" async events, even when "Changed Namespace List" Get Log Page has not arrived from driver • Bit 2 – reserved • Bit 3 – force-enable "Namespace Management capability" NVMe OACS even though it is not supported by the controller • Bit 4 - Disable Scatter-Gather Lists support. For more details, see section "OS Issues".

Note

If not set, the SNAP NVMe controller supports an optional NVMe command only if all the namespaces attached to it when loading the driver support it. To bypass this feature, you may explicitly set the NVMe optional command support bit by using its corresponding flag.

For example, a controller created with `--compare 0` would not support the optional `compare` NVMe command regardless of its

attached namespaces.

Example request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "nvme_controller_create",
  "params": {
    "nqn": "nqn.2022-10.io.nvda.nvme:0",
    "pf_id": 0,
    "num_queues": 8,
  }
}
```

nvme_controller_destroy

Destroy a previously created NVMe controller. The controller can be uniquely identified by a controller name as acquired from `nvme_controller_create`.

Command parameters:

Parameter	Mandatory ?	Type	Description
`ctrl`	Yes	String	Controller name
`release_msix`	No	1/0	Release MSIX back to free pool. Applies only for VFs.

nvme_controller_suspend

While suspended, the controller stops handling new requests from the host driver. All pending requests (if any) will be processed after resume.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name
`timeout_ms`	No	Number	Suspend timeout **Note** If IOs are pending in the bdev layer (or in the remote target), the operation fails and resumes after this timeout. If `timeout_ms` is not provided, the operation waits until the IOs complete without a timeout on the SNAP layer.
`force`	No	0/1	Force suspend even when there are inflight I/Os
`admin_only`	No	0/1	Suspend only the admin queue
`live_update_notifier`	No	0/1	Send a live update notification via IPC

nvme_controller_resume

The resume command continues the (previously-suspended) controller's handling of new requests sent by the driver. If the controller is created in suspended mode, resume is also used to start initial communication with host driver.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name
`live_update`	No	0/1	Live update resume

nvme_controller_snapshot_get

Take a snapshot of the current state of the controller and dump it into a file. This file may be used to create a controller based on this snapshot. For the snapshot to be consistent, users should call this function only when the controller is suspended (see `nvme_controller_suspend` RPC).

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name
`filename`	Yes	String	File path

nvme_controller_vfs_msix_reclaim

Reclaims all VFs MSIX back to the PF's free MSIX pool.

This function can only be applied on PFs and can only be run when SR-IOV is not set on host side (i.e., `sriov_numvfs = 0`).

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name

nvme_controller_list

Provide a list of all active (created) NVMe controllers with their characteristics.

Command parameters:

Parameter	Mandatory?	Type	Description
`nqn`	No	String	Subsystem qualified name
`ctrl`	No	String	Only search for a specific controller

nvme_controller_modify

This function allows user to modify some of the controller's parameters in real-time, after it was already created.

Modifications can only be done when the emulated function is in idle state - thus there is no driver communicating with it.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	No	String	Controller Name
num_queues	No	int	Number of queues for the controller
num_msix	No	int	Number of MSIX to be used for a controller. Relevant only for VF controllers (when dynamic MSIX feature is enabled).

nvme_controller_attach_ns

Attach a previously created NVMe namespace to given NVMe controller under the same subsystem.

The result in the response object returns `true` for success and `false` for failure.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name
`nsid`	Yes	Number	Namespace ID

nvme_controller_detach_ns

Detach a previously attached namespace with a given NSID from the NVMe controller.

The result in the response object returns `true` for success and `false` for failure.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name
`nsid`	Yes	Number	Namespace ID

nvme_controller_dbg_io_stats_get

The result in the response object returns `true` for success and `false` for failure.

Command parameters:

Parameter	Mandatory?	Type	Description
`ctrl`	Yes	String	Controller name

```
"ctrl_id": "NVMeCtrl2",
"queues": [
  {
    "queue_id": 0,
    "core_id": 0,
    "read_io_count": 19987068,
```

```
    "write_io_count": 6319931,
    "flush_io_count": 0
  },
  {
    "queue_id": 1,
    "core_id": 1,
    "read_io_count": 9769556,
    "write_io_count": 3180098,
    "flush_io_count": 0
  }
],
"read_io_count": 29756624,
"write_io_count": 9500029,
"flush_io_count": 0
    }
```

NVMe Configuration Examples

NVMe Configuration for Single Controller

On the DPU:

```
spdk_rpc.py bdev_nvme_attach_controller -b nvme0 -t rdma -a
1.1.1.1 -f ipv4 -s 4420 -n nqn.2022-10.io.nvda.nvme:swx-storage
snap_rpc.py nvme_subsystem_create --nqn nqn.2022-
10.io.nvda.nvme:0
snap_rpc.py nvme_namespace_create -b nvme0n1 -n 1 --nqn nqn.2022-
10.io.nvda.nvme:0 --uuid 263826ad-19a3-4feb-bc25-4bc81ee7749e
snap_rpc.py nvme_controller_create --nqn nqn.2022-
10.io.nvda.nvme:0 --pf_id 0 --suspended
snap_rpc.py nvme_controller_attach_ns -c NVMeCtrl1 -n 1
snap_rpc.py nvme_controller_resume -c NVMeCtrl1
```

Note

It is necessary to create a controller in a suspended state. Afterward, the namespaces can be attached, and only then should the controller be resumed using the `nvme_controller_resume` RPC.

Note

To safely detach/attach namespaces, the UUID must be provided to force the UUID to remain persistent.

NVMe Cleanup for Single Controller

```
snap_rpc.py nvme_controller_detach_ns -c NVMeCtrl2 -n 1
snap_rpc.py nvme_controller_destroy -c NVMeCtrl2
snap_rpc.py nvme_namespace_destroy -n 1 --nqn nqn.2022-
10.io.nvda.nvme:0
snap_rpc.py nvme_subsystem_destroy --nqn nqn.2022-
10.io.nvda.nvme:0
spdk_rpc.py bdev_nvme_detach_controller nvme0
```

NVMe and Hotplug Cleanup for Single Controller

```
snap_rpc.py nvme_controller_detach_ns -c NVMeCtrl1 -n 1
snap_rpc.py emulation_device_detach_prepare --vuid
MT2114X12200VBLKS1D0F0
```

```
snap_rpc.py nvme_controller_destroy -c NVMeCtrl1
snap_rpc.py emulation_device_detach --vuid MT2114X12200VBLKS1D0F0
snap_rpc.py nvme_namespace_destroy -n 1 --nqn nqn.2022-
10.io.nvda.nvme:0
snap_rpc.py nvme_subsystem_destroy --nqn nqn.2022-
10.io.nvda.nvme:0
spdk_rpc.py bdev_nvme_detach_controller nvme0
```

NVMe Configuration for 125 VFs SR-IOV

1. Update the firmware configuration as described section "SR-IOV Firmware Configuration".

2. Reboot the host.

3. Create a dummy controller on the parent PF:

```
[dpu] # snap_rpc.py nvme_subsystem_create --nqn nqn.2022-
10.io.nvda.nvme:0
[dpu] # snap_rpc.py nvme_controller_create --nqn nqn.2022-
10.io.nvda.nvme:0 --ctrl NVMeCtrl1 --pf_id 0 --admin_only
```

4. Create 125 Bdevs (Remote or Local), 125 NSs and 125 controllers:

```
[dpu] for i in `seq 0 124`; do \
   # spdk_rpc.py bdev_null_create null$((i+1)) 64 512;
   # snap_rpc.py nvme_namespace_create -b null$((i+1)) -n
$((i+1)) --nqn nqn.2022-10.io.nvda.nvme:0 --uuid 3d9c3b54-
5c31-410a-b4f0-7cf2afd9e$((i+100));
   # snap_rpc.py nvme_controller_create --nqn nqn.2022-
10.io.nvda.nvme:0 --ctrl NVMeCtrl$((i+2)) --pf_id 0 --vf_id
$i --suspended;
```

```
   # snap_rpc.py nvme_controller_attach_ns -c
NVMeCtrl$((i+2)) -n $((i+1));
   # snap_rpc.py nvme_controller_resume -c NVMeCtrl$(i+2);
done
```

5. Load the driver and configure VFs:

```
[host] # modprobe -v nvme
[host] # echo 125 >
/sys/bus/pci/devices/0000\:25\:00.2/sriov_numvfs
```

Environment Variable Management

snap_global_param_list

`snap_global_param_list` lists all existing environment variables.

The following is an example response for the `snap_global_param_list` command:

```
[
  "SNAP_ENABLE_POLL_SKIP       : set     : 0  ",
  "SNAP_POLL_CYCLE_SIZE        : not set : 16 ",
  "SNAP_RPC_LOG_ENABLE         : set     : 1  ",
  "SNAP_MEMPOOL_SIZE_MB        : set     : 1024",
  "SNAP_MEMPOOL_4K_BUFFS_PER_CORE : not set : 1024",
  "SNAP_RDMA_ZCOPY_ENABLE      : set     : 1  ",
  "SNAP_TCP_XLIO_ENABLE        : not set : 1  ",
  "SNAP_TCP_XLIO_TX_ZCOPY      : not set : 1  ",
  "MLX5_SHUT_UP_BF             : not set : 0  ",
  "SNAP_SHARED_RX_CQ           : not set : 1  ",
```

```
  "SNAP_SHARED_TX_CQ           : not set : 1   ",
...
```

Advanced Features

RPC Log History

RPC log history (enabled by default) records all the RPC requests (from `snap_rpc.py` and `spdk_rpc.py`) sent to the SNAP application and the RPC response for each RPC requests in a dedicated log file, `/var/log/snap-log/rpc-log`. This file is visible outside the container (i.e., the log file's path on the DPU is `/var/log/snap-log/rpc-log` as well).

The `SNAP_RPC_LOG_ENABLE` env can be used to enable (`1`) or disable (`0`) this feature.

Info

RPC log history is supported with SPDK version spdk23.01.2-12 and above.

Warning

When RPC log history is enabled, the SNAP application writes (in append mode) RPC request and response message to `/var/log/snap-log/rpc-log` constantly. Pay attention to the size of this file. If it gets too large, delete the file on the DPU before launching the SNAP pod.

SR-IOV

The configuration of SR-IOV depends on the kernel version and requires careful consideration to ensure optimal device visibility and system stability.

For deployments with multiple virtual devices, autoprobe must be disabled to ensure reliable device discovery by setting `sriov_drivers_autoprobe=0` in `/sys/bus/pci/devices/<BDF>/`. Failing to do this may cause the following issues:

- Incomplete device visibility
- Missing virtual disks
- Potential system hangs during device initialization
- Unreliable behavior with large numbers of VFs (>100)

Configurations required for large-scale deployments:

1. Disable autoprobe:

   ```
   echo 0 > /sys/bus/pci/devices/<BDF>/sriov_drivers_autoprobe
   ```

2. Manually bind the VFs as required using your specific driver binding tools (e.g., `driverctl` or `bind/unbind` in sysfs).

For small-scale deployments (with fewer than 100 VFs), you can use the `sriov_totalvfs` sysfs entry to set the number of VFs:

```
echo <number_of_vfs> > /sys/bus/pci/devices/<BDF>/sriov_totalvfs
```

Note

Applying this configuration for larger deployments.

Note

After completing the SR-IOV configuration, no disk will be exposed in the hypervisor by default. The disk will only appear within the VM after the associated PCIe VF is assigned to the VM using the virtualization manager (e.g., libvirt, VMware, etc.). If you need to use the device directly from the hypervisor, manually bind the PCIe VF to the desired driver.

Note

Hot-plug PFs do not support SR-IOV.

Info

It is recommended to add `pci=assign-busses` to the boot command line when creating more than 127 VFs.

00000196-6a22-dea4-abb7-ef62aead0003

Zero Copy (SNAP-direct)

Note

Zero-copy is supported on SPDK 21.07 and higher.

SNAP-direct allows SNAP applications to transfer data directly from the host memory to remote storage without using any staging buffer inside the DPU.

SNAP enables the feature according to the SPDK BDEV configuration only when working against an SPDK NVMe-oF RDMA block device.

To enable zero copy, set the environment variable (as it is enabled by default):

```
SNAP_RDMA_ZCOPY_ENABLE=1
```

For more info refer to the section SNAP Environment Variables.

NVMe/TCP XLIO Zero Copy

NVMe/TCP Zero Copy is implemented as a custom `NVDA_TCP` transport in SPDK NVMe initiator, and it is based on a new XLIO socket layer implementation.

The implementation is different for Tx and Rx:

- The NVMe/TCP Tx Zero Copy is similar between RDMA and TCP in that the data is sent from the host memory directly to the wire without an intermediate copy to Arm memory
- The NVMe/TCP Rx Zero Copy allows achieving partial zero copy on the Rx flow by eliminating copy from socket buffers (XLIO) to application buffers (SNAP). But data still must be DMA'ed from Arm to host memory.

To enable NVMe/TCP Zero Copy, use `SPDK v22.05.nvda --with-xlio` (`v22.05.nvda` or higher).

Note

For more information about XLIO including limitations and bug fixes, refer to the *NVIDIA Accelerated IO (XLIO) Documentation*.

To enable SNAP TCP XLIO Zero Copy:

1. SNAP container: Set the environment variables and resources in the YAML file:

```
resources:
  requests:
    memory: "4Gi"
    cpu: "8"
  limits:
    hugepages-2Mi: "4Gi"
    memory: "6Gi"
    cpu: "16"

## Set according to the local setup
env:
  - name: APP_ARGS
    value: "--wait-for-rpc"
  - name: SPDK_XLIO_PATH
    value: "/usr/lib/libxlio.so"
```

2. SNAP sources: Set the environment variables and resources in the relevant scripts

    1. In `run_snap.sh`, edit the `APP_ARGS` variable to use the SPDK command line argument `--wait-for-rpc`:

    ```
    APP_ARGS="--wait-for-rpc"
    ```

    2. In `set_environment_variables.sh`, uncomment the `SPDK_XLIO_PATH` environment variable:

```
export SPDK_XLIO_PATH="/usr/lib/libxlio.so"
```

Note

NVMe/TCP XLIO requires a BlueField Arm OS hugepage size of 4Gi. For information on configuring the hugepages, refer to sections "Step 1: Allocate Hugepages" and "Adjusting YAML Configuration".

At high scale, it is required to use the global variable `XLIO_RX_BUFS=4096` even though it leads to high memory consumption. Using `XLIO_RX_BUFS=1024` requires lower memory consumption but limits the ability to scale the workload.

Info

For more info refer to the section "SNAP Environment Variables".

Tip

It is recommended to configure NVMe/TCP XLIO with the transport ack timeout option increased to 12.

```
[dpu] spdk_rpc.py bdev_nvme_set_options --
transport-ack-timeout 12
```

Other `bdev_nvme` options may be adjusted according to requirements.

Expose an NVMe-oF subsystem with one namespace by using a TCP transport type on the remote SPDK target.

```
[dpu] spdk_rpc.py sock_set_default_impl -i xlio
[dpu] spdk_rpc.py framework_start_init
[dpu] spdk_rpc.py bdev_nvme_set_options --transport-ack-timeout
12
[dpu] spdk_rpc.py bdev_nvme_attach_controller -b nvme0 -t
nvda_tcp -a 3.3.3.3 -f ipv4 -s 4420 -n nqn.2023-01.io.nvmet
[dpu] snap_rpc.py nvme_subsystem_create --nqn nqn.2023-
01.com.nvda:nvme:0
[dpu] snap_rpc.py nvme_namespace_create -b nvme0n1 -n 1 --nqn
nqn. 2023-01.com.nvda:nvme:0 --uuid 16dab065-ddc9-8a7a-108e-
9a489254a839
[dpu] snap_rpc.py nvme_controller_create --nqn nqn.2023-
01.com.nvda:nvme:0 --ctrl NVMeCtrl1 --pf_id 0 --suspended --
num_queues 16
[dpu] snap_rpc.py nvme_controller_attach_ns -c NVMeCtrl1 -n 1
[dpu] snap_rpc.py nvme_controller_resume -c NVMeCtrl1 -n 1

[host] modprobe -v nvme
[host] fio --filename /dev/nvme0n1 --rw randrw --name=test-randrw
--ioengine=libaio --iodepth=64 --bs=4k --direct=1 --numjobs=1 --
runtime=63 --time_based --group_reporting --verify=md5
```

Info

For more information on XLIO, please refer to XLIO documentation.

Encryption

The SPDK version that comes with SNAP supports hardware encryption/decryption offload. To enable AES/XTS, follow the instructions under section "Modifying SF Trust Level to Enable Encryption".

Zero Copy (SNAP-direct) with Encryption

SNAP offers support for zero copy with encryption for `bdev_nvme` with an RDMA transport.

> **Note**
>
> If another `bdev_nvme` transport or base bdev other than NVMe is used, then zero copy flow is not supported, and additional DMA operations from the host to the BlueField Arm are performed.

> **Info**
>
> Refer to section "SPDK Crypto Example" to see how to configure zero copy flow with AES_XTS offload.

Command	Description
`mlx5_scan_accel_module`	Accepts a list of devices to be used for the crypto operation
`accel_crypto_key_create`	Creates a crypto key
`bdev_nvme_attach_controller`	Constructs NVMe block device

Command	Description
`bdev_crypto_create`	Creates a virtual block device which encrypts write IO commands and decrypts read IO commands

mlx5_scan_accel_module

Accepts a list of devices to use for the crypto operation provided in the `--allowed-devs` parameter. If no devices are specified, then the first device which supports encryption is used.

For best performance, it is recommended to use the devices with the largest InfiniBand MTU (4096). The MTU size can be verified using the `ibv_devinfo` command (look for the max and active MTU fields). Normally, the `mlx5_2` device is expected to have an MTU of 4096 and should be used as an allowed crypto device.

Command parameters:

Parameter	Mandatory?	Type	Description
`qp-size`	No	Number	QP size
`num-requests`	No	Number	Size of the shared requests pool
`allowed-devs`	No	String	Comma-separated list of allowed device names (e.g., "mlx5_2") **Note** Make sure that the device used for RDMA traffic is selected to support zero copy.

Parameter	Mandatory?	Type	Description
`enable-driver`	No	Boolean	Enables accel_mlx5 platform driver. Allows AES_XTS RDMA zero copy.

accel_crypto_key_create

Creates crypto key. One key can be shared by multiple bdevs.

Command parameters:

Parameter	Mandatory?	Type	Description
`cipher`	Yes	Number	Crypto protocol (AES_XTS)
`key`	Yes	Number	Key
`key2`	Yes	Number	Key2
`name`	Yes	String	Key name

bdev_nvme_attach_controller

Creates NVMe block device.

Command parameters:

Parameter	Mandatory?	Type	Description
`name`	Yes	String	Name of the NVMe controller, prefix for each bdev name
`trtype`	Yes	String	NVMe-oF target trtype (e.g., rdma, pcie)
`traddr`	Yes	String	NVMe-oF target address (e.g., an IP address or BDF)

Parameter	Mandatory?	Type	Description
`trsvcid`	No	String	NVMe-oF target trsvcid (e.g., a port number)
`addrfam`	No	String	NVMe-oF target adrfam (e.g., ipv4, ipv6)
`nqn`	No	String	NVMe-oF target subnqn

bdev_crypto_create

This RPC creates a virtual crypto block device which adds encryption to the base block device.

Command parameters:

Parameter	Mandatory?	Type	Description
`base_bdev_name`	Yes	String	Name of the base bdev
`name`	Yes	String	Crypto bdev name
`key-name`	Yes	String	Name of the crypto key created with `accel_crypto_key_create`

SPDK Crypto Example

The following is an example of a configuration with a crypto virtual block device created on top of `bdev_nvme` with RDMA transport and zero copy support:

```
[dpu] # spdk_rpc.py mlx5_scan_accel_module --allowed-devs
"mlx5_2" --enable-driver
[dpu] # spdk_rpc.py framework_start_init
```

```
[dpu] # spdk_rpc.py accel_crypto_key_create -c AES_XTS -k
00112233445566778899001122334455 -e
11223344556677889900112233445500 -n test_dek
[dpu] # spdk_rpc.py bdev_nvme_attach_controller -b nvme0 -t rdma
-a 1.1.1.1 -f ipv4 -s 4420 -n nqn.2016-06.io.spdk:cnode0
[dpu] # spdk_rpc.py bdev_crypto_create nvme0n1 crypto_0 -n
test_dek
[dpu] # snap_rpc.py spdk_bdev_create crypto_0
[dpu] # snap_rpc.py nvme_subsystem_create --nqn nqn.2023-
05.io.nvda.nvme:0
[dpu] # snap_rpc.py nvme_controller_create --nqn nqn.2023-
05.io.nvda.nvme:0 --pf_id 0 --ctrl NVMeCtrl0 --suspended
[dpu] # snap_rpc.py nvme_namespace_create –nqn nqn.2023-
05.io.nvda.nvme:0 --bdev_name crypto_0 –-nsid 1 -–uuid 263826ad-
19a3-4feb-bc25-4bc81ee7749e
[dpu] # snap_rpc.py nvme_controller_attach_ns –-ctrl NVMeCtrl0 --
nsid 1
[dpu] # snap_rpc.py nvme_controller_resume –-ctrl NVMeCtrl0
```

Virtio-blk Live Migration

Live migration is a standard process supported by QEMU which allows system administrators to pass devices between virtual machines in a live running system. For more information, refer to QEMU VFIO Device Migration documentation.

Live migration is supported for SNAP virtio-blk devices. It can be activated using a driver with proper support (e.g., NVIDIA's proprietary vDPA-based Live Migration Solution).

```
snap_rpc.py virtio_blk_controller_create --dbg_admin_q …
```

SNAP Container Live Upgrade

Live upgrade enables updating the SNAP image used by a container without causing SNAP container downtime.

While newer SNAP releases may introduce additional content, potentially causing behavioral differences during the upgrade, the process is designed to ensure backward compatibility. Updates between releases within the same sub-version (e.g., 4.0.0-x to 4.0.0-y) should proceed without issues.

However, updates across different major or minor versions may require changes to system components (e.g., firmware, BFB), which may impact backward compatibility and necessitate a full reboot post update. In those cases, live updates are unnecessary.

Live Upgrade Prerequisites

To enable live upgrade, perform the following modifications:

1. Allocate double hugepages for the destination and source containers.

2. Make sure the requested amount of CPU cores is available.

   The default YAML configuration sets the container to request a CPU core range of 8-16. This means that the container is not deployed if there are fewer than 8 available cores, and if there are 16 free cores, the container utilizes all 16.

   For instance, if a container is currently using all 16 cores and, during a live upgrade, an additional SNAP container is deployed. In this case, each container uses 8 cores during the upgrade process. Once the source container is terminated, the destination container starts utilizing all 16 cores.

   > **Note**
   >
   > For 8-core DPUs, the `.yaml` must be edited to the range of 4-8 CPU cores.

3. Change the name of the `doca_snap.yaml` file that describes the destination container (e.g., `doca_snap_new.yaml`) so as to not overwrite the running container `.yaml`.

4. Change the name of the new `.yaml` pod and container on lines 16 and 20, respectively (e.g., `snap-new`).

5. Deploy the the destination container by copying the new yaml (e.g., `doca_snap_new.yaml`) to kubelet.

Note

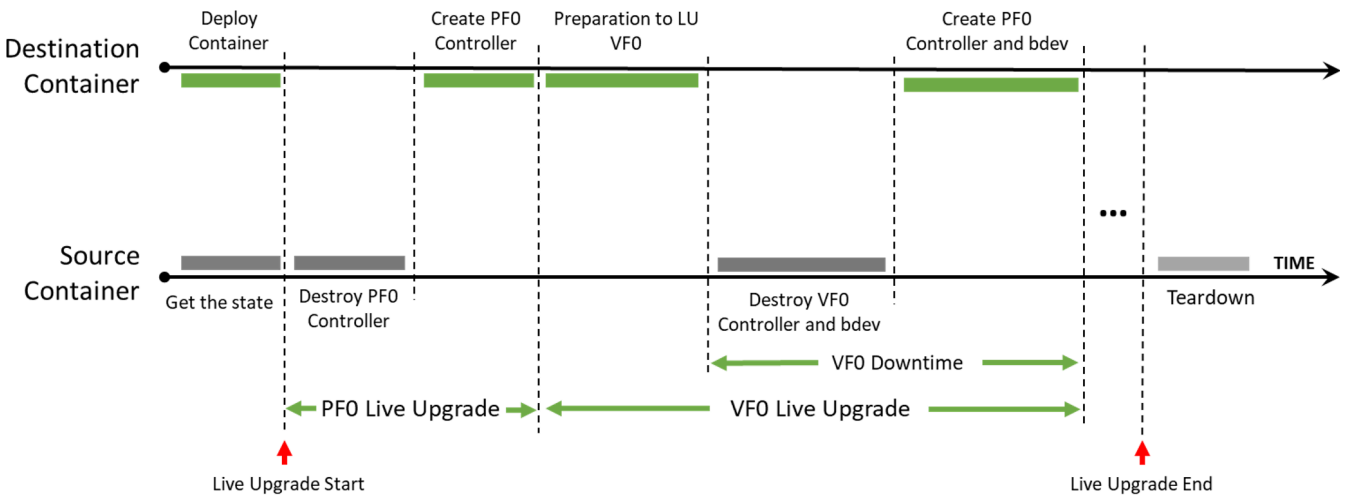
- After deploying the destination container, until the live update process is complete, avoid making any configuration changes via RPC. Specifically, do not create or destroy hotplug functions.
- When restoring a controller in the destination container during a live update, it is recommended to use the same arguments originally used for controller creation in the source container.
- User may need to update the RPC alias since the new container name has been edited.

Note

When restoring a controller in the destination container during a live update, it is recommended to use the same arguments originally used for controller creation in the source container.

Live Upgrade Flow

The way to live upgrade the SNAP image is to move the SNAP controllers and SPDK block devices between different containers while minimizing the duration of the host VMs impact.



- Source container – the running container before live upgrade
- Destination container – the running container after live upgrade

SNAP Container Live Upgrade Procedure

1. Follow the steps in section "Live Upgrade Prerequisites" and deploy the destination SNAP container using the modified `yaml` file.
2. Query the source and destination containers:

```
crictl ps -r
```

3. Check for `SNAP started successfully` in the logs of the destination container, then copy the live update from the container to your environment.

```
[dpu] crictl logs -f <dest-container-id>
[dpu] crictl exec <dest-container-id> cp
/opt/nvidia/nvda_snap/bin/live_update.py /etc/nvda_snap/
```

4. Run the `live_update.py` script to move all active objects from the source container to the destination container:

```
[dpu] cd /etc/nvda_snap
[dpu] ./live_update.py -s <source-container-id> -d <dest-container-id>
```

5. Delete the source container.

> **Note**
>
> To post RPCs, use the crictl tool:
>
> ```
> crictl exec -it <container-id X> snap_rpc.py <RPC-method>
> crictl exec -it <container-id Y> spdk_rpc.py <RPC-method>
> ```

> **Note**
>
> To automate the SNAP configuration (e.g., following failure or reboot) as explained in section "Automate SNAP Configuration (Optional)", `spdk_rpc_init.conf` and `snap_rpc_init.conf` must not include any configs as part of the live upgrade. Then, once the transition to the new container is done, `spdk_rpc_init.conf` and `snap_rpc_init.conf` can be modified with the desired configuration.

SNAP Container Live Upgrade Commands

The live update tool is designed to support fast live updates. It iterates over the available emulation functions and performs the following actions for each one.

> **Info**
>
> Note that the physical function controller must remain available in the source container while its virtual functions are being live-updated. Only after the virtual function controllers in the source container is destroyed, the physical function controller can be removed from that container.

1. On the source container:

```
snap_rpc.py virtio_blk_controller_suspend --ctrl [ctrl_name]
--events_only
```

2. On the destination container:

```
spdk_rpc.py bdev_nvme_attach_controller ...
snap_rpc.py virtio_blk_controller_create ... --suspended --
live_update_listener
```

3. On the source container:

```
snap_rpc.py virtio_blk_controller_destroy --ctrl [ctrl_name]
spdk_rpc.py bdev_nvme_detach_controller [bdev_name]
```

SR-IOV Dynamic MSIX Management

Message Signaled Interrupts eXtended (MSIX) is an interrupt mechanism that allows devices to use multiple interrupt vectors, providing more efficient interrupt handling than traditional interrupt mechanisms such as shared interrupts. In Linux, MSIX is supported in the kernel and is commonly used for high-performance devices such as network adapters, storage controllers, and graphics cards. MSIX provides benefits such as reduced CPU utilization, improved device performance, and better scalability, making it a popular choice for modern hardware.

However, proper configuration and management of MSIX interrupts can be challenging and requires careful tuning to achieve optimal performance, especially in a multi-function environment as SR-IOV.

By default, BlueField distributes MSIX vectors evenly between all virtual PCIe functions (VFs). This approach is not optimal as users may choose to attach VFs to different VMs, each with a different number of resources. Dynamic MSIX management allows the user to manually control of the number of MSIX vectors provided per each VF independently.

> **Note**
>
> Configuration and behavior are similar for all emulation types, and specifically NVMe and virtio-blk.

Dynamic MSIX management is built from several configuration steps:

1. At this point, and in any other time in the future when no VF controllers are opened (`sriov_numvfs=0`), all PF-related MSIX vectors can be reclaimed from the VFs to the PF's free MSIX pool.

2. User must take some of the MSIX from the free pool and give them to a certain VF during VF controller creation.

3. When destroying a VF controller, the user may choose to release its MSIX back to the pool.

Once configured, the MSIX link to the VFs remains persistent and may change only in the following scenarios:

- User explicitly requests to return VF MSIXs back to the pool during controller destruction.
- PF explicitly reclaims all VF MSIXs back to the pool.
- Arm reboot (FE reset/cold boot) has occurred.

To emphasize, the following scenarios do not change MSIX configuration:

- Application restart/crash.
- Closing and reopening PF/VFs without dynamic MSIX support.

The following is an NVMe example of dynamic MSIX configuration steps (similar configuration also applies for virtio-blk):

1. Reclaim all MSIX from VFs to PF's free MSIX pool:

   ```
   snap_rpc.py nvme_controller_vfs_msix_reclaim <CtrlName>
   ```

2. Query the controller list to get information about the resources constraints for the PF:

   ```
   # snap_rpc.py nvme_controller_list -c <CtrlName>
   …     'free_msix': 100,
   …     'free_queues': 200,
   …     'vf_min_msix': 2,
   …     'vf_max_msix': 64,
   …     'vf_min_queues': 0,
   …     'vf_max_queues': 31,
   …
   ```

   Where:

   - `free_msix` stands for the number of total MSIX available in the PF's free pool, to be assigned for VFs, through the parameter `vf_num_msix` (of the

`<protocol>_controller_create` RPC).

- `free_queues` stands for the number of total queues (or "doorbells") available in the PF's free pool, to be assigned for VFs, through the parameter `num_queues` (of the `<protocol>_controller_create` RPC).
- `vf_min_msix` and `vf_max_msix` together define the available configurable range of `vf_num_msix` parameter value which can be passed in `<protocol>_controller_create` RPC for each VF.
- `vf_min_queues` and `vf_max_queues` together define the available configurable range of `num_queues` parameter value which can be passed in `<protocol>_controller_create` RPC for each VF.

3. Distribute MSIX between VFs during their creation process, considering the PF's limitations:

```
snap_rpc.py nvme_controller_create_ --vf_num_msix <n> --
num_queues <m> …
```

Note

It is strongly advised to provide both `vf_num_msix` and `num_queues` parameters upon VF controller creation. Providing only one of the values may result in a conflict between MSIX and queue configuration, which may in turn cause the controller/driver to malfunction.

Tip

In NVMe protocol, MSIX is used by NVMe CQ. Therefore, it is advised to assign 1 MSIX out of the PF's global pool (`free_msix`) for each assigned queue.

In virtio protocol, MSIX is used by virtqueue and one extra MSIX is required for BAR configuration changes notification. Therefore, it is advised to assign 1 MSIX out of the PF's global pool (`free_msix`) for every assigned queue, and one more as configuration MSIX.

In summary, the best practice for queues/MSIX ratio configuration is:

- For NVMe – `num_queues` = `vf_num_msix`
- For virtio – `num_queues` = `vf_num_msix`-1

4. Upon VF teardown, release MSIX back to the free pool:

```
snap_rpc.py nvme_controller_destroy_ --release_msix …
```

5. Set SR-IOV on the host driver:

```
echo <N> > /sys/bus/pci/devices/<BDF>/sriov_numvfs
```

Note

It is highly advised to open all VF controllers in SNAP in advance before binding VFs to the host/guest driver. That way, for example in case of a configuration mistake which does not leave enough MSIX for all VFs, the configuration remains reversible as MSIX is still modifiable. Otherwise, the driver may try to use the already-configured VFs before all VF configuration has finished but will not be able to use all of them (due to lack of MSIX). The latter scenario may result in host deadlock which, at worst, can be recovered only with cold boot.

Note

There are several ways to configure dynamic MSIX safely (without VF binding):

1. Disable kernel driver automatic VF binding to kernel driver:

   ```
   # echo 0 >
   /sys/bus/pci/devices/sriov_driver_autoprobe
   ```

   After finishing MSIX configuration for all VFs, they can then be bound to VMs, or even back to the hypervisor:

   ```
   echo "0000:01:00.0" >
   /sys/bus/pci/drivers/nvme/bind
   ```

2. Use VFIO driver (instead of kernel driver) for SR-IOV configuration.

For example:

```
# echo 0000:af:00.2 >
/sys/bus/pci/drivers/vfio-pci/bind
# Bind PF to VFIO driver
# echo 1 >
/sys/module/vfio_pci/parameters/enable_sriov
# echo <N> > /sys/bus/pci/drivers/vfio-
pci/0000:af:00.2/sriov_numvfs   # Create VFs
device for it
```

Recovery

NVMe Recovery

NVMe recovery allows the NVMe controller to be recovered after a SNAP application is closed whether gracefully or after a crash (e.g., `kill -9`).

To use NVMe recovery, the controller must be re-created in a suspended state with the same configuration as before the crash (i.e., the same bdevs, num queues, and namespaces with the same uuid, etc).

> **Note**
>
> The controller must be resumed only after all NSs are attached.

NVMe recovery uses files on the BlueField under `/dev/shm` to recover the internal state of the controller. Shared memory files are deleted when the BlueField is reset. For this reason, recovery is not supported after BF reset.

Virtio-blk Crash Recovery

The following options are available to enable virtio-blk crash recovery.

Virtio-blk Crash Recovery with --force_in_order

For virtio-blk crash recovery with `--force_in_order`, disable the `VBLK_RECOVERY_SHM` environment variable and create a controller with the `--force_in_order` argument.

In virtio-blk SNAP, the application is not guaranteed to recover correctly after a sudden crash (e.g., `kill -9`).

To enable the virtio-blk crash recovery, set the following:

```
snap_rpc.py virtio_blk_controller_create --force_in_order …
```

Note

Setting `force_in_order` to 1 may impact virtio-blk performance as it will serve the command in-order.

Note

If `--force_in_order` is not used, any failure or unexpected teardown in SNAP or the driver may result in anomalous behavior because of limited support in the Linux kernel virtio-blk driver.

Virtio-blk Crash Recovery without --force_in_order

For virtio-blk crash recovery without `--force_in_order`, enable the `VBLK_RECOVERY_SHM` environment variable and create a controller without the `--force_in_order` argument.

Virtio-blk recovery allows the virtio-blk controller to be recovered after a SNAP application is closed whether gracefully or after a crash (e.g., `kill -9`).

To use virtio-blk recovery without `--force_in_order` flag. `VBLK_RECOVERY_SHM` must be enabled, the controller must be recreated with the same configuration as before the crash (i.e., same bdevs, num queues, etc).

When `VBLK_RECOVERY_SHM` is enabled, virtio-blk recovery uses files on the BlueField under `/dev/shm` to recover the internal state of the controller. Shared memory files are

deleted when the BlueField is reset. For this reason, recovery is not supported after BlueField reset.

SNAP Configuration Recovery

SNAP can save its configuration as defined by user RPCs, and upon SNAP restart it can use the config json file to recover the configuration that was set before SNAP was closed.

This is used to recover SNAP state and cannot be used to initially configure SNAP.

> **Note**
>
> Currently supported for virtio blk.

To use SNAP configuration recovery, set a dir path for env var `SNAP_RPC_INIT_CONF_JSON`. snap_config.json file in this path should be deleted if a new configuration (different than the one before SNAP closing) is set after SNAP is restarted.

Running this way ensures that your configuration is saved only after all changes in the script have been successfully performed.

When SNAP is restarted, there is no need to re-run snap rpcs or set rpcs in init files after initial configuration. SNAP will load the configuration set in SNAP_RPC_INIT_CONF_JSON file. This method is better for fast recovery.

> **Warning**
>
> - When changing ctrl/function configuration, driver must remain unloaded until configuration change is done. Otherwise, configuration recovery may fail if the configuration change has not finished successfully.
> - Hotplugged emulation functions are persistent between SNAP runs (not BF resets) and should be set once during SNAP's initial

configuration. Only the ctrls created on these function are saved in the config state.

- If the use case supports SNAP crash recovery after a reboot, the recommended file path inside the container is /etc/nvda_snap. If recovery is for an unsupported use case, store the file in a path such as /tmp/ or /dev/shm.

Improving SNAP Recovery Time

The following table outlines features designed to accelerate SNAP initialization and recovery processes following termination.

Feature	Description	How to?
SPDK JSON-RPC configuration file	An initial configuration can be specified for the SPDK configuration in SNAP. The configuration file is a JSON file containing all the SPDK JSON-RPC method invocations necessary for the desired configuration. Moving from posting RPCs to JSON file improves bring-up time. **Info** For more information check SPDK JSON-RPC documentation.	To generate a JSON-RPC file based on the current configuration, run: `spdk_rpc.py save_config > config.json` The `config.json` file can then be passed to a new SNAP deployment using the environment variable in the YAML `SPDK_RPC_INIT_CONF_JSON` . **Note** If SPDK encou nters

Feature	Description	How to?
		an error while processing the JSON configuration file, the initialization phase fails, causing SNAP to exit with an error code.
Disable SPDK accel functionality	The SPDK accel functionality is necessary when using NVMe TCP features. If NVMe TCP is not used, accel should be manually disabled to reduce the SPDK startup time, which can otherwise take few seconds. To disable all accel functionality edit the flags `disable_signature`, `disable_crypto`, and `enable_module`.	Edit the config file as follows: { "method": "mlx5_scan_accel_module", "params": { "qp_size": 64, "cq_size": 1024, "num_requests": 2048,

Feature	Description	How to?

		<pre>"enable_driver": false,
		"split_mb_blocks": 0,
		"siglast": false,
		"qp_per_domain": false,
		"disable_signature": true,
		"disable_crypto": true,
		"enable_module": false
		}</pre>

Provide the emulation manager name	If the `SNAP_EMULATION_MANAGER` environment variable is not defined (default), SNAP searches through all available devices to find the emulation manager which may slow down initialization process. Explicitly defining the device reduces the chance of initialization delays.	Use `SNAP_EMULATION_MANAGER` to modify the the variable on the YAML. Refer to the "SNAP Environment Variables" page for more information.
DPU mode for virtio-blk	DPU mode is supported only with virtio-blk. DPU mode r educes SNAP downtime during crash recovery.	Set `VIRTIO_EMU_PROVIDER=dpu` to modify the the variable on the YAML. Refer to the "SNAP Environment Variables" page for more information.
SNAP configuration recover	SNAP configuration recovery enables restoring the SNAP state without the need to re-post SNAP RPCs. By moving from posting individual RPCs to	Set `SNAP_RPC_INIT_CONF_JSON`

Feature	Description	How to?
y for virtio-blk	using a pre-saved JSON configuration file, the bring-up time is significantly improved.	to a path where config file should be saved. Also use an SPDK JSON-RPC configuration file.
Hugepages allocation	SNAP allocates a mempool from hugepages. Reducing its size can impact the duration of SNAP's crash recovery.	SNAP_MEMPOOL_SIZE_MB is set to1024MB by default.

IO/Core Multiplexer (MP)

The IO/Core Multiplexer (MP) feature is a mechanism that allows users to control how I/O processing from a single source is distributed across different DPU cores. The feature offers two main policy options:

1. None (Default): In this policy, all I/Os from a single source are handled by a single DPU core. Different I/O sources are evenly distributed among the available DPU cores. This approach is optimized for I/O latency and is recommended for users prioritizing low-latency operations.

2. (Weighted) Round Robin: This policy distributes I/Os from a single source among multiple DPU cores in a round-robin fashion. For backends supporting per-core weight updates (e.g., SPDK nvmf bdev), I/Os are distributed according to given weights. Otherwise, they are evenly distributed among all DPU cores. This approach is designed to optimize I/O bandwidth and is recommended for users seeking to maximize throughput or those dealing with low per-core backend performance (e.g., TCP-based backends).

To configure the IO/Core Multiplexer policy, users need to set the IO_CORE_DISTRIBUTION_POLICY environment variable. The available options are:

- "none": Refers to the default policy where all I/Os from a single source are handled by a single DPU core.

- "weighted_rr": Refers to the (Weighted) Round Robin policy, distributing I/Os across multiple cores.

Note

NOTE: weighted_rr policy is not supported for virtio-blk.

SNAP ML Optimizer

The SNAP ML optimizer is a tool designed to fine-tune SNAP's poller parameters, enhancing SNAP I/O handling performance and increasing controller throughput based on specific environments and workloads.

During workload execution, the optimizer iteratively adjusts configurations (actions) and evaluates their impact on performance (reward). By predicting the best configuration to test next, it efficiently narrows down to the optimal setup without needing to explore every possible combination.

Once the optimal configuration is identified, it can be applied to the target system, improving performance under similar conditions. Currently, the tool supports "IOPS" as the reward metric, which it aims to maximize.

SNAP ML Optimizer Preparation Steps

Machine Requirements

The device should be able to SSH to the BlueField:

- Python 3.10 or above
- At least 6 GB of free storage

Setting Up SNAP ML Optimizer

To set up the SNAP ML optimizer:

1. Copy the `snap_ml` folder from the container to the shared `nvda_snap` folder and then to the requested machine:

```
crictl exec -it $(crictl ps -s running -q --name snap) cp -r
/opt/nvidia/nvda_snap/bin/snap_ml /etc/nvda_snap/
```

2. Change directory to the `snap_ml` folder:

```
cd tools/snap_ml
```

3. Create a virtual environment for the SNAP ML optimizer.

```
python3 -m venv snap_ml
```

This ensures that the required dependencies are installed in an isolated environment.

4. Activate the virtual environment to start working within this isolated environment:

```
source snap_ml/bin/activate
```

5. Install the Python package requirements:

```
pip3 install --no-cache-dir -r requirements.txt
```

This may take some time depending on your system's performance.

6. Run the SNAP ML Optimizer.

```
python3 snap_ml.py --help
```

Use the `--help` flag to see the available options and usage information:

```
--version Show the version and exit.
-f, --framework <TEXT> Name of framework (Recommended: ax ,
supported: ax, pybo).
-t, --total-trials <INTEGER> Number of optimization
iterations. The recommended range is 25-60.
--filename <TEXT> where to save the results (default:
last_opt.json).
--remote <TEXT> connect remotely to the BlueField card,
format: <bf_name>:<username>:<password>
--snap-rpc-path <TEXT> Snap RPC prefix (default: container
path).
--log-level <TEXT> CRITICAL | ERROR | WARN | WARNING | INFO |
DEBUG
--log-dir <TEXT> where to save the logs.
```

SNAP ML Optimizer Related RPCs

snap_actions_set

The `snap_actions_set` command is used to dynamically adjust SNAP parameters (known as "actions") that control polling behavior. This command is a core feature of SNAP-AI tools, enabling both automated optimization for specific environments and workloads, as well as manual adjustment of polling parameters.

Command parameters:

Parameter	Mandatory?	Type	Description
`poll_size`	No	Number	Maximum number of IOs SNAP passes in a single polling cycle (integer; 1-256)

Parameter	Mandatory?	Type	Description
`poll_ratio`	No	Number	The rate in which SNAP poll cycles occur (float; 0< `poll_ratio` ⩽1)
`max_inflights`	No	Number	Maximum number of in-flight IOs per core (integer; 1-65535)
`max_iog_batch`	No	Number	Maximum fairness batch size (integer; 1-4096)
`max_new_ios`	No	Number	Maximum number of new IOs to handle in a single poll cycle (integer; 1-4096)

snap_reward_get

The `snap_reward_get` command retrieves performance counters, specifically completion counters (or "reward"), which are used by the optimizer to monitor and enhance SNAP performance.

No parameters are required for this command.

Run the ML Optimizer

To optimize SNAP's parameters for your environment, use the following command:

```
python3 snap_ml.py --framework ax --total-trials 40 --filename example.json --remote <bf_hostname>:<username>:<password> --log-dir <log_directory>
```

Results and Post-optimization Actions

Once the optimization process is complete, the tool automatically applies the optimized parameters. These parameters are also saved in a `example.json` file in the following format:

```
{
    "poll_size": 30,
    "poll_ratio": 0.6847347955107689,
    "max_inflights": 32768,
    "max_iog_batch": 512,
    "max_new_ios": 32
}
```

Additionally, the tool documents all iterations, including the actions taken and the rewards received, in a timestamped file named `example_<timestamp>.json`.

Applying Optimized Parameters Manually

Users can apply the optimized parameters on fresh instances of SNAP service by explicitly calling the `snap_actions_set` RPC with the optimized parameters as follows:

```
snap_rpc.py snap_actions_set –poll_size 30 –poll_ratio 0.6847 --
max_inflights 32768 –max_iog_batch 512 –max_new_ios 32
```

> **Note**
>
> It is only recommended to use the optimized parameters if the system is expected to behave similarly to the system on which the SNAP ML optimizer is used.

Deactivating Python Environment

Once users are done using the SNAP ML Optimizer, they can deactivate the Python virtual environment by running:

```
deactivate
```

Plugins

Plugins are modular components or add-ons that enhance the functionality of the SNAP application. They integrate seamlessly with the main software, allowing additional features without requiring changes to the core codebase. Plugins are designed for use only with the source package, as it allows customization during the build process, such as enabling or disabling plugins as needed.

In containerized environments, the SNAP application is shipped as a pre-built binary with a fixed configuration. Since the binary in the container is precompiled, adding or removing plugins is not possible. The containerized software only supports the plugins included during its build. For environments requiring plugin flexibility, such as adding custom plugins, the source package must be used.

To build a SNAP source package with a plugin, perform the following instead of following the basic build steps :

1. Move to the sources folder. Run:

   ```
   cd /opt/nvidia/nvda_snap/src/
   ```

2. Build the sources with plugin to be enabled. Run:

   ```
   meson setup /tmp/build -Denable-bdev-null=true -Denable-bdev-malloc=true
   ```

3. Compile the sources. Run:

```
meson compile -C /tmp/build
```

4. Install the sources. Run:

```
meson install -C /tmp/build
```

5. Configure the SNAP environment variables and run SNAP service as explained in sections "Configure SNAP Environment Variables" and "Run SNAP Service".

Bdev

SNAP supports various types of block devices (bdev), offering flexibility and extensibility in interacting with storage backends. These bdev plugins provide different storage emulation options, allowing customization without requiring modifications to the core software.

SPDK

- SPDK is the default plugin used by SNAP. If no specific plugin is explicitly specified, SNAP will default to using SPDK for block device operations.
- For more information, refer to spdk_bdev.

Malloc

- The Malloc plugin is intended for performance analysis and debugging purposes only; it is not suitable for production use
- It creates a memory-backed block device by allocating a buffer in memory and exposing it as a block device
- Since data is stored in memory, it is lost when the system shuts down
- This plugin can be enabled using the `enable-bdev-malloc` build option

Malloc configuration example:

1. Create Malloc bdev and use it with an NVMe controller:

```
# snap_rpc.py snap_bdev_malloc_create --bdev test 64 512
# snap_rpc.py nvme_subsystem_create -s nqn.2020-12.mlnx.snap
# snap_rpc.py nvme_namespace_create -s nqn.2020-12.mlnx.snap
-t malloc -b test -n 1
# snap_rpc.py nvme_controller_create --pf_id=0 -s nqn.2020-
12.mlnx.snap --mdts=7
# snap_rpc.py nvme_controller_attach_ns -c NVMeCtrl1 -n 1
```

2. Delete Malloc bdev:

```
# snap_rpc.py snap_bdev_malloc_destroy test
```

3. Resize Malloc bdev:

```
# snap_rpc.py snap_bdev_malloc_resize test 32
```

   This removes the existing bdev and creates a new one with the specified size. Data on the existing bdev will be lost.

NULL

- The NULL plugin is designed for performance analysis and debugging purposes and is not intended for production use.
- It acts as a dummy block device, accepting I/O requests and emulating a block device without performing actual I/O operations.

- It is useful for testing or benchmarking scenarios that do not involve real storage devices.
- The plugin consumes minimal system resources.
- It can be enabled using the `enable-bdev-null` build option.

NULL configuration example:

1. Create a NULL bdev and use it with an NVMe controller:

```
# snap_rpc.py snap_bdev_null_create_dbg test 1 512
# snap_rpc.py nvme_subsystem_create -s nqn.2020-12.mlnx.snap
# snap_rpc.py nvme_namespace_create -s nqn.2020-12.mlnx.snap
-t malloc -b test -n 1
# snap_rpc.py nvme_controller_create --pf_id=0 -s nqn.2020-
12.mlnx.snap --mdts=7
# snap_rpc.py nvme_controller_attach_ns -c NVMeCtrl1 -n 1
```

2. Delete the NULL bdev:

```
# snap_rpc.py snap_bdev_null_destroy_dbg test
```

Appendix – DPU Firmware Configuration

Before configuring SNAP, the user must ensure that all firmware configuration requirements are met. By default, SNAP is disabled and must be enabled by running both common SNAP configurations and additional protocol-specific configurations depending on the expected usage of the application (e.g., hot-plug, SR-IOV, UEFI boot, etc).

After configuration is finished, the host must be power cycled for the changes to take effect.

Note

To verify that all configuration requirements are satisfied, users may query the current/next configuration by running the following:

```
mlxconfig -d /dev/mst/mt41692_pciconf0 -e query
```

System Configuration Parameters

Parameter	Description	Possible Values
`INTERNAL_CPU_MODEL`	Enable BlueField to work in internal CPU model **Note** Must be set to `1` for storage emulations.	0/1
`SRIOV_EN`	Enable SR-IOV	0/1
`PCI_SWITCH_EMULATION_ENABLE`	Enable PCI switch for emulated PFs	0/1
`PCI_SWITCH_EMULATION_NUM_PORT`	The maximum number of hotplug emulated PFs which equals `PCI_SWITCH_EMULATION_NUM_PORT`–1. For example, if `PCI_SWITCH_EMULATION_NUM_PORT=32`, then the maximum number of hotplug emulated PFs would be 31.	[0,2-32]

Parameter	Description	Possible Values
	Note One switch port is reserved for all static PFs.	

Note

`SRIOV_EN` is valid only for static PFs.

RDMA/RoCE Configuration

BlueField's RDMA/RoCE communication is blocked for BlueField's default OS interfaces (nameds ECPFs, typically mlx5_0 and mlx5_1). If RoCE traffic is required, additional network functions (scalable functions) must be added which support RDMA/RoCE traffic.

Note

The following is not required when working over TCP or even RDMA/IB.

To enable RoCE interfaces, run the following from within the DPU:

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s PER_PF_NUM_SF=1
```

```
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0 s PF_SF_BAR_SIZE=8
PF_TOTAL_SF=2
[dpu] mlxconfig -d /dev/mst/mt41692_pciconf0.1 s PF_SF_BAR_SIZE=8
PF_TOTAL_SF=2
```

NVMe Configuration

Parameter	Description	Possible Values
`NVME_EMULATION_ENABLE`	Enable NVMe device emulation	0/1
`NVME_EMULATION_NUM_PF`	Number of static emulated NVMe PFs	[0-4]
`NVME_EMULATION_NUM_MSIX`	Number of MSIX assigned to emulated NVMe PFs **Note** The firmware treats this value as a best effort value. The *effective* number of MSI-X given to the function should be queried as part of the nvme_controller_list RPC command.	[0-63]
`NVME_EMULATION_NUM_VF_MSIX`	Number of MSIX per emulated NVMe VF **Note** The firmware treats this value as a best effort value. The *effective* number of MSI-X given to the	[0-4095]

Parameter	Description	Possible Values
	function should be queried as part of the nvme_controller_list RPC command. **Note** This value should match the maximum number of queues assigned to a VF's NVMe SNAP controller through the nvme_controller_create `num_queues` parameter, as each queue requires one MSIX interrupt.	
`NVME_EMULATION_NUM_VF`	Number of VFs per emulated NVMe PF **Note** If not 0, overrides `NUM_OF_VFS`; valid only when `SRIOV_EN=1`.	[0-256]
`EXP_ROM_NVME_UEFI_x86_ENABLE`	Enable NVMe UEFI exprom driver **Note** Used for UEFI boot process.	0/1

Virtio-blk Configuration

⚠ Warning

Due to virtio-blk protocol limitations, using bad configuration while working with static virtio-blk PFs may cause the host server OS to fail on boot.

Before continuing, make sure you have configured:

- A working channel to access Arm even when the host is shut down. Setting such channel is out of the scope of this document. Please refer to *NVIDIA BlueField DPU BSP* documentation for more details.
- Add the following line to `/etc/nvda_snap/snap_rpc_init.conf`:

```
virtio_blk_controller_create –pf_id 0
```

For more information, please refer to section "Virtio-blk Emulation Management".

Parameter	Description	Possible Values
`VIRTIO_BLK_EMULATION_ENABLE`	Enable virtio-blk device emulation	0/1
`VIRTIO_BLK_EMULATION_NUM_PF`	Number of static emulated virtio-blk PFs **ⓘ Note**	[0-4]

Parameter	Description	Possible Values
	See WARNING above.	
`VIRTIO_BLK_EMULATION_NUM_MSIX`	Number of MSIX assigned to emulated virtio-blk PFs **Note** The firmware treats this value as a best effort value. The *effective* number of MSI-X given to the function should be queried as part of the virtio_blk_controller_list RPC command.	[0-63]
`VIRTIO_BLK_EMULATION_NUM_VF_MSIX`	Number of MSIX per emulated virtio-blk VF **Note** The firmware treats this value as a best effort value. The *effective* number of MSI-X given to the function should be queried as part of the virtio_blk_controller_list RPC command. **Note** This value should match the maximum number of queues assigned to a VF's NVMe SNAP controller through the nvme_controller_create `num_queues` parameter, as each queue requires one MSIX interrupt.	[0-4095]

Parameter	Description	Possible Values
`VIRTIO_BLK_EMULATION_NUM_VF`	Number of VFs per emulated virtio-blk PF **Note** If not 0, overrides `NUM_OF_VFS`; valid only when `SRIOV_EN=1`	[0-2000]
`EXP_ROM_VIRTIO_BLK_UEFI_x86_ENABLE`	Enable virtio-blk UEFI exprom driver **Note** Used for UEFI boot process.	0/1

Appendix – Configure Persistent Network Interfaces

To configure persistent network interfaces so they are not lost after reboot. Under `/etc/sysconfig/network-scripts` modify the following four files, or create them if do not exist, then perform a reboot:

```
# cd /etc/sysconfig/network-scripts/
# cat ifcfg-p0
  NAME="p0"
  DEVICE="p0"
  NM_CONTROLLED="no"
  DEVTIMEOUT=30
  PEERDNS="no"
```

```
  ONBOOT="yes"
  BOOTPROTO="none"
  TYPE=Ethernet
  MTU=9000

# cat ifcfg-p1
  NAME="p1"
  DEVICE="p1"
  NM_CONTROLLED="no"
  DEVTIMEOUT=30
  PEERDNS="no"
  ONBOOT="yes"
  BOOTPROTO="none"
  TYPE=Ethernet
  MTU=9000

# cat ifcfg-enp3s0f0s0
  NAME="enp3s0f0s0"
  DEVICE="enp3s0f0s0"
  NM_CONTROLLED="no"
  DEVTIMEOUT=30
  PEERDNS="no"
  ONBOOT="yes"
  BOOTPROTO="static"
  TYPE=Ethernet
  IPADDR=1.1.1.1
  PREFIX=24
  MTU=9000

# cat ifcfg-enp3s0f1s0
  NAME="enp3s0f1s0"
  DEVICE="enp3s0f1s0"
  NM_CONTROLLED="no"
  DEVTIMEOUT=30
  PEERDNS="no"
  ONBOOT="yes"
```

```
BOOTPROTO="static"
TYPE=Ethernet
IPADDR=1.1.1.2
PREFIX=24
MTU=9000
```

Appendix – Building SNAP Container with Custom SPDK

The SNAP source package contains the files necessary for building a container with a custom SPDK.

To build the container:

1. Download and install the SNAP sources package:

   ```
   [dpu] # dpkg -i /path/snap-sources_<version>_arm64.deb
   ```

2. Navigate to the `src` folder and use it as the development environment:

   ```
   [dpu] # cd /opt/nvidia/nvda_snap/src
   ```

3. Copy the following to the container folder:

   - SNAP source package – required for installing SNAP inside the container
   - Custom SPDK – to `container/spdk`. For example:

     ```
     [dpu] # cp /path/snap-sources_<version>_arm64.deb
     container/
     ```

```
[dpu] # git clone -b v23.01.1 --single-branch --depth 1
--recursive --shallow-submodules
https://github.com/spdk/spdk.git container/spdk
```

4. Modify the `spdk.sh` file if necessary as it is used to compile SDPK.

5. To build the container:

   - For Ubuntu, run:

     ```
     [dpu] # ./container/build_public.sh --snap-pkg-
     file=snap-sources_<version>_arm64.deb
     ```

   - For CentOS, run:

     ```
     [dpu] # rpm -i snap-sources-<version>.el8.aarch64.rpm
     [dpu] # cd /opt/nvidia/nvda_snap/src/
     [dpu] # cp /path/snap-sources_<version>_arm64.deb
     container/
     [dpu] # git clone -b v23.01.1 --single-branch --depth 1
     --recursive --shallow-submodules
     https://github.com/spdk/spdk.git container/spdk
     [dpu] # yum install docker-ce docker-ce-cli
     [dpu] # ./container/build_public.sh --snap-pkg-
     file=snap-sources_<version>_arm64.deb
     ```

6. Transfer the created image from the Docker tool to the crictl tool. Run:

   ```
   [dpu] # docker save doca_snap:<version> doca_snap.tar
   [dpu] # ctr -n=k8s.io images import doca_snap.tar
   ```

> **Note**
>
> To transfer the container image to other setups, refer to appendix "Appendix - Deploying Container on Setups Without Internet Connectivity".

7. To verify the image, run:

```
[DPU] #  crictl images
IMAGE                                     TAG
IMAGE ID            SIZE
docker.io/library/doca_snap               <version>
79c503f0a2bd7       284MB
```

8. Edit the image filed in the `container/doca_snap.yaml` file. Run:

```
image: doca_snap:<version>
```

9. Use the YAML file to deploy the container. Run:

```
[dpu] # cp doca_snap.yaml /etc/kubelet.d/
```

> **Note**
>
> The container deployment preparation steps are required.

Appendix – Deploying Container on Setups Without Internet Connectivity

When Internet connectivity is not available on a DPU, Kubelet scans for the container image locally upon detecting the SNAP YAML. Users can load the container image manually before the deployment.

To accomplish this, users must download the necessary resources using a DPU with Internet connectivity and subsequently transfer and load them onto DPUs that lack Internet connectivity.

1. To download the `.yaml` file:

```
[bf] # wget --content-disposition
https://api.ngc.nvidia.com/v2/resources/nvidia/doca/doca_conta
to-yaml>/doca_snap.yaml
```

> **Note**
>
> Access the latest download command on NGC by visiting https://catalog.ngc.nvidia.com/orgs/nvidia/teams/doca/containers/doca_snap. The SNAP tag `doca_snap:4.1.0-doca2.0.2` is used in this section as an example. Latest tag is also available on NGC.

2. To download SNAP container image:

```
[bf] # crictl pull nvcr.io/nvidia/doca/doca_snap:4.1.0-
doca2.0.2
```

3. To verify that the SNAP container image exists:

```
[bf] # crictl images

IMAGE                                     TAG
IMAGE ID            SIZE
nvcr.io/nvidia/doca/doca_snap             4.1.0-doca2.0.2
9d941b5994057       267MB
k8s.gcr.io/pause                          3.2
2a060e2e7101d       251kB
```

> **Note**
>
> `k8s.gcr.io/pause` image is required for the SNAP container.

4. To save the images as a `.tar` file:

```
[bf] # mkdir images
[bf] # ctr -n=k8s.io image export
images/snap_container_image.tar
nvcr.io/nvidia/doca/doca_snap:4.1.0-doca2.0.2
[bf] # ctr -n=k8s.io image export images/pause_image.tar
k8s.gcr.io/pause:3.2
```

5. Transfer the `.tar` files and run the following to load them into Kubelet:

```
[bf] # sudo ctr --namespace k8s.io image import
images/snap_container_image.tar
```

```
[bf] # sudo ctr --namespace k8s.io image import
images/pause_image.tar
```

6. Now, the image exists in the tool and is ready for deployment.

```
[bf] # crictl images
IMAGE                                     TAG
IMAGE ID            SIZE
nvcr.io/nvidia/doca/doca_snap             4.1.0-doca2.0.2
9d941b5994057       267MB
k8s.gcr.io/pause                          3.2
2a060e2e7101d       251kB
```

Appendix – Install Legacy SPDK

To build SPDK-19.04 for SNAP integration:

1. Cherry-pick a critical fix for SPDK shared libraries installation (originally applied on upstream only since v19.07).

```
[spdk.git] git cherry-pick cb0c0509
```

2. Configure SPDK:

```
[spdk.git] git submodule update --init
[spdk.git] ./configure --prefix=/opt/mellanox/spdk --disable-
tests --without-crypto --without-fio --with-vhost --without-
pmdk --without-rbd --with-rdma --with-shared --with-iscsi-
initiator --without-vtune
```

```
[spdk.git] sed -i -e
's/CONFIG_RTE_BUILD_SHARED_LIB=n/CONFIG_RTE_BUILD_SHARED_LIB=y
dpdk/build/.config
```

> **Note**
>
> The flags `--prefix`, `--with-rdma`, and `--with-shared` are mandatory.

3. Make SPDK (and DPDK libraries):

```
[spdk.git] make && make install
[spdk.git] cp dpdk/build/lib/* /opt/mellanox/spdk/lib/
[spdk.git] cp dpdk/build/include/*
/opt/mellanox/spdk/include/
```

Appendix – PCIe BDF to VUID Translation

PCIe BDF (Bus, Device, Function) is a unique identifier assigned to every PCIe device connected to a computer. By identifying each device with a unique BDF number, the computer's OS can manage the system's resources efficiently and effectively.

PCIe BDF values are determined by host OS and are hence subject to change between different runs, or even in a single run. Therefore, the BDF identifier is not the best fit for permanent configuration.

To overcome this problem, NVIDIA devices add an extension to PCIe attributes, called VUIDs. As opposed to BDF, VUID is persistent across runs which makes it useful as a PCIe function identifier.

PCI BDF and VUID can be extracted one out of the other, using `lspci` command:

1. To extract VUID out of BDF:

```
[host] lspci -s <BDF> -vvv | grep -i VU | awk '{print $4}'
```

2. To extract BDF out of VUID:

```
[host] ./get_bdf.py <VUID>
[host] cat ./get_bdf.py
#!/usr/bin/python3

import subprocess
import sys

vuid = sys.argv[1]

# Split the output into individual PCI function entries
lspci_output =
subprocess.check_output(['lspci']).decode().strip().split('\n'

# Create an empty dictionary to store the results
pci_functions = {}

# Loop through each PCI function and extract the BDF and full
info
for line in lspci_output:
    bdf = line.split()[0]
    if vuid in subprocess.check_output(['lspci', '-s', bdf,
'-vvv']).decode():
        print(bdf)
        exit(0)

print("Not Found")
```

Appendix – SNAP Memory Consumption

This appendix explains how SNAP consumes memory and how to manage memory allocation.

The user must allocate the DPA hugepages memory according to the section "Step 1: Allocate Hugepages". It is possible to use use a portion of the DPU memory allocation in the SNAP container as described in section "Adjusting YAML Configuration". This configuration includes the following minimum and maximum values:

- The minimum allocation which the SNAP container consumes:

```
resources:
  requests:
    memory: "4Gi"
```

- The maximum allocation that the SNAP container is allowed to consume:

```
resources:
  limits:
    hugepages-2Mi: "4Gi"
```

Hugepage memory is used by the following:

- SPDK `mem-size` global variable which controls the SPDK hugepages consumption (configurable in SPDK, 1GB by default)
- SNAP `SNAP_MEMPOOL_SIZE_MB` – used with non-ZC mode as IO buffers staging buffers on the Arm. By default, the SNAP mempool consumes 1G from the SPDK `mem-size` hugepages allocation. SNAP mempool may be configured using the `SNAP_MEMPOOL_SIZE_MB` global variable (minimum is 64 MB).

> **Note**

If the value assigned is too low, with non-ZC, a performance degradation could be seen.

- SNAP and SPDK internal usage – 1G should be used by default. This may be reduced depending on the overall scale (i.e., VFs/num queues/QD).
- XLIO buffers – allocated only when NVMeTCP XLIO is enabled.

The following is the limit of the container memory allowed to be used by the SNAP container:

```
resources:
  limits:
    memory: "6Gi"
```

Info

This includes the hugepages limit (in this example, additional 2G of non-hugepages memory).

The SNAP container also consumes DPU SHMEM memory when NVMe recovery is used (described in section "NVMe Recovery"). In addition, the following resources are used:

```
limits:
  memory:
```

Appendix – Host OS Configuration

With Linux environment on host OS, additional kernel boot parameters may be required to support SNAP related features:

- To use SR-IOV:
  - For Intel, `intel_iommu=on iommu=pt` must be added
  - For AMD, `amd_iommu=on iommu=pt` must be added
- To use PCIe hotplug, `pci=realloc` must be added
- `modprobe.blacklist=virtio_blk,virtio_pci` for non-built-in `virtio-blk` driver or `virtio-pci` driver

To view boot parameter values, run:

```
cat /proc/cmdline
```

It is recommended to use the following with virtio-blk:

```
[dpu] cat /proc/cmdline BOOT_IMAGE … pci=realloc
modprobe.blacklist=virtio_blk,virtio_pci
```

To enable VFs (virtio_blk/NVMe):

```
echo 125 > /sys/bus/pci/devices/0000\:27\:00.4/sriov_numvfs
```

Intel Server Performance Optimizations

```
cat /proc/cmdline
BOOT_IMAGE=(hd0,msdos1)/vmlinuz-5.15.0_mlnx root=UUID=91528e6a-
b7d3-4e78-9d2e-9d5ad60e8273 ro crashkernel=auto
resume=UUID=06ff0f35-0282-4812-894e-111ae8d76768 rhgb quiet
```

```
iommu=pt intel_iommu=on pci=realloc
modprobe.blacklist=virtio_blk,virtio_pci
```

AMD Server Performance Optimizations

```
cat /proc/cmdline
cat /proc/cmdline BOOT_IMAGE=(hd0,msdos1)/vmlinuz-5.15.0_mlnx
root=UUID=91528e6a-b7d3-4e78-9d2e-9d5ad60e8273 ro
crashkernel=auto resume=UUID=06ff0f35-0282-4812-894e-111ae8d76768
rhgb quiet iommu=pt amd_iommu=on pci=realloc
modprobe.blacklist=virtio_blk,virtio_pci
```

SNAP-4 Service Release Notes

The release note pages provide information for NVIDIA® BlueField®-3 SNAP software such as changes and new features, software known issues, and bug fixes.

Changes and New Features

Key features in NVIDIA SNAP 4.6.0:

- Improved application crash recovery downtime when using the virtio-blk protocol
- SNAP recovery JSON configuration files for virtio-blk protocol
- Introduction of a new plugins concept (plugin advanced feature - link)
- Bug fixes

Release Notes:

Release notes in NVIDIA SNAP 4.6.0:

- DPU Virtio-blk data path provider as default.

Bug Fixes

Ref #	Issue
4244085	Description: Deleting the controller of the VF on DPU side causes SNAP to crash when device on host unbinds the driver
	Keywords:
	Discovered in version: 4.6.0
4212912	Description: serial_number is not correct with virtio_blk_controller_bdev_attach
	Keywords:

Ref #	Issue
	Discovered in version: 4.6.0
4196047	Description: Failed to get MSIX_EQ when do hot plug / unplug
	Keywords:
	Discovered in version: 4.6.0
4176109	Description: msix number is less than queue number and no way to modify it
	Keywords:
	Discovered in version: 4.6.0

Known Issues

SNAP Issues

The following are known limitations of this NVMe/virtio-blk SNAP software version.

Ref #	Issue
3631346	Description: When using dynamic MSIX with NVMe protocol, the `free_queues` PF controller property (as described in s ection "SR-IOV Dynamic MSIX Management") is not valid and always shows `0`.
	Workaround: Ignore the value and assume the `free_queues` pool is large enough.
	Keywords: NVMe; MSIX
	Discovered in version: 4.4.1
-	Description: The SPDK `bdev_uring` is not supported. It will be supported next release.
	Workaround: N/A
	Keywords: NVMe
	Discovered in version: 4.4.0
3817040	Description: When running `nvme_controller_suspend` RPC with the `--timeout` parameter, if timeout expires, the device is no longer operational and cannot be resumed.
	Workaround: Destroy and re-create the controller.

Ref #	Issue
	Keywords: NVMe
	Discovered in version: 4.4.0
3809646	Description: When working with a new DPA provider, when sending DMA followed by an interrupt to DPA, it wakes up before DMA is written to the buffer causing DPA to miss events.
	Workaround: Add a software-based periodic wake-up mechanism.
	Keywords: NVMe
	Discovered in version: 4.4.0
3773346	Description: In virtio-blk controller configuration, when running with SPDK NVMe-oF initiator as a backend, an unaligned `size_max` value may cause memory corruption.
	Workaround: `size_max` and `seg_max` values must be a power of 2.
	Keywords: Virtio-blk; NVMe-oF; spdk
	Discovered in version: 4.3.1
3745842	Description: When running with NVMe/TCP SPDK block device as a backend, SNAP cannot work over more than 8 cores.
	Workaround: Work with Arm core mask which uses only 8 cores.
	Keywords: NVMe; TCP; SPDK
	Discovered in version: 4.3.1
-	Description: The container image may becomes corrupted, resulting in the container status showing as `exited` with the error message `/usr/bin/supervisord: exec format error`.
	Workaround: Remove the YAML from kubelet, use `crictl images` to list the images and `crictl rmi <image-id>` to remove the image. Run `systemctl restart containerd` and `systemctl restart kubelet`, then copy the YAML file again to kubelet.
	Keywords: NGC; container image
	Discovered in version: 4.3.1
3757	Description: When running virtio-blk emulation with large IOs (>128K) and SPDK's nvmf initiator as a backend, IOs may fail in SPDK layer due to bad alignment.

Ref #	Issue
171	Workaround: `size_max` value of `virtio_blk_controller_create` RPC must be set and be a power of 2.
	Keywords: SPDK, virtio-blk, size_max
	Discovered in version: 4.3.1
3689918375363637	Description: SNAP container bring-up takes a long time when configured with a large number of emulations, possibly taking longer than the default NVMe driver timeout.
	Workaround: Increase NVMe driver IO timeout to 300 seconds (instead of 30).
	Keywords: NVMe; recovery; kernel driver
	Discovered in version: 4.3.0
-	Description: NVMeTCP XLIO is currently not supported when running 64K page size kernels on the DPU Arm cores (as is the case for CentOS 8.x, Rocky 8.x, or openEuler 20.x).
	Workaround: N/A
	Keywords: 64K page size; NVMeTCP XLIO
	Discovered in version: 4.1.0
3264154	Description: NVMeTCP XLIO is not supported when running 64K page size kernels on the DPU Arm cores (such is the case with CentOS 8.x, Rocky 8.x, or openEuler 20.x).
	Workaround: N/A
	Keywords: Page size; NVMeTCP XLIO
	Discovered in version: 4.1.0
-	Description: NVMe over RDMA full offload is not supported.
	Workaround: N/A
	Keywords: NVMe over RDMA; support
	Discovered in version: 4.0.0
41109433	Description: Hot-unplugging a hotplugged Virtio BLK device is not allowed unless a Virtio BLK controller has previously been created for the device.
	Workaround: Create a Virtio BLK controller on the device which need to do hotunplug

Ref #	Issue
	Keywords: hotplug, hotunplug
	Discovered in version: 4.5.0
4104709	Description: Some legacy operating systems (e.g., RockyLinux with kernel 4.18) issues virtio-blk zero-length I/Os during boot (e.g., during EDD probing).
	Workaround: Set VIRTIO_BLK_ZERO_LEN_IO_FAIL=1 environment variable to configure SNAP to fail zero-length I/Os in virtio-blk.
	Keywords: virtio-blk, zero-length I/Os
	Reported in version: 4.6.0
-	Description: When using both NVMe and Virtio-blk protocols in SNAP, their data providers may share the same DPA harts, potentially causing NVMe configuration or performance issues. This is especially relevant when Virtio-blk is in DPU mode and NVMe is in DPA mode.
	Workaround: Use dpa_helper_core_mask and dpa_nvme_core_mask environment variable as DPA core masks, for example dpa_helper_core_mask=0x0000FFFF and dpa_nvme_core_mask=0xFFFF0000,
	Keywords: Virtio-blk and NVMe, DPA core mask.
	Reported in version: 4.6.0
4094152	Description: When one (or more) of the NVMe OCI functiuons is undergoing FLR, nvme_controller_list RPC may fail.
	Workaround: Retry nvme_controller_list RPC until successful,
	Keywords: FLR .
	Reported in version: 4.6.0

OS/vendor Issues

Info

The following are not BlueField SNAP limitations.

Ref #	Issue
-	Description: Some old Windows OS NVMe drivers have buggy usage of SGL support.
	Workaround: Disable SGL support when using Windows OS by setting the `--quirks` bit 4 to 1 in `snap_rpc.py nvme_controller_create` RPC.
	Keywords: Windows; NVMe
	Reported in version: 4.4.0
2879262	Description: When the virtio-blk kernel driver cannot find enough MSI-X vectors to satisfy all its opened virtqueues, it failovers to assign a single MSI-X vector to all virtqueues which negatively impacts performance. In addition, when a large number (e.g., 64) of virtqueues are associated with a single MSI-X, the kernel may enter a soft-lockup (kernel bug) and the IO will hang.
	Workaround: Always keep `num_queues` < `num_msix`. Best practice is to not set `--num_queues` at all when creating virtio-blk controllers, and the best-suited value is automatically chosen based on available MSI-X.
	Keywords: Virtio-blk; kernel driver; MSI-X
	Reported in version: 4.3.0
-	Description: If PCIe devices are inserted prior to the hot-plug driver being loaded on host, the hot-plug driver in kernel version less than 4.19 does not enable the slot even if the slot is occupied (i.e., presence detected in slot status register). That is, only the presence state of the slot is changed by firmware but the PCIe slot is not enabled by the kernel after host bootup (i.e., So that we can't get the PCIe device by lspci on host side, and the bdf is 0 on controller.
	Workaround: Add `pciehp.pciehp_force=1` to the boot command line on host.
	Keywords: Virtio-blk; kernel driver; hot-plug
	Reported in version: 4.2.1
-	Description: RedHat/Centos 7.x does not handle "online" (post driver probe) namespace additions/removals correctly.
	Workaround: Use `--quirks=0x2` option in `snap_rpc.py nvme_controller_create`.
	Keywords: NVMe; CentOS; RedHat; kernel
	Reported in version: 4.1.0

Ref #	Issue
-	Description: Some Windows drivers have experimental support for "online" (post driver probe) namespace additions/removal, although such support is not communicated with the device.
	Workaround: Use `--quirks=0x1` option in `snap_rpc.py nvme_controller_create`.
	Keywords: NVMe; Windows
	Reported in version: 4.1.0
-	Description: VMWare ESXi supports "online" (post driver probe) namespace additions/removal, only if “Namespace Management” is supported by controller.
	Workaround: Use `--quirks=0x8` option in `snap_rpc.py nvme_controller_create`.
	Keywords: NVMe, ESXi
	Reported in version: 4.1.0
-	Description: Ubuntu 22.04 does not support 500 VFs.
	Workaround: N/A
	Keywords: Virtio-blk; kernel driver; Ubuntu 22.04
	Reported in version: 4.1.0
-	Description: Virtio-blk Linux kernel driver does not handle PCIe FLR events.
	Workaround: N/A
	Keywords: Virtio-blk; kernel driver
	Reported in version: 4.0.0
3679373	Description: Virtio-blk spdk driver (vfio-pci based) does not handle PCIe FLR events.
	Workaround: N/A
	Keywords: Virtio-blk; SPDK driver
	Reported in version: 4.3.0
-	Description: A n ew virtio-blk Linux kernel driver (starting kernel 4.18) does not support hot-unplug during traffic. Since the kernel may self-generate spontaneous IOs, on rare occasions, an issue may happen even when no traffic is explicitly being run.

Ref #	Issue
	Workaround: N/A
	Keywords: Virtio-blk; kernel driver
	Reported in version: 4.0.0
	Description: SPDK NVMf/RDMA initiator fails to connect to kernel NVMf/RDMA remote target.
	Workaround: Use setting `spdk_rpc.py bdev_nvme_set_options --io-queue-requests=128` on SPDK configuration
	Keywords: SPDK, NVMf, RDMA, kernel
	Reported in version: 4.3.1
-	Description: Windows OS virtio-blk driver expects at least 64K data to be available for a single IO request
-	Workaround: Use `seg_max` and `size_max` parameters configuration to match requirements (`seg_max * size_max > 64K`).
-	Keywords: Windows, virtio-blk
-	Reported in version: 4.3.1
-	Description: Some old Windows OS versions have malfunctioning inbox virtio-blk driver, expects a 3-party virtio-blk driver to be pre-installed to operate properly.
-	Workaround: Use verified 3-party driver published by fedora (link).
-	Keywords: Windows, virtio-blk
-	Reported in version: 4.3.1
-	Description: When using hotplugged PCIe devices, after all devices are plugged, the host must be rebooted for Windows to detect all devices (some Windows versions will perform reboot automatically).
-	Workaround: N/A
-	Keywords: Hotplug, Windows.
-	Reported in version: 4.5.0
3748674	Description: On most modern Linux distributions, unplugging a PCIe function from the host while there are inflight I/Os can cause the `virtio-blk` driver to hang.
3748674	Workaround: N/A

Ref #	Issue
	Keywords: Hotplug, Linux.
	Reported in version: 4.5.0
4158322	Description: Windows OS does not support online PCIe rescan, requiring a system reboot after each PCIe hotplug operation. Without a reboot, the OS cannot detect subsequent hotplug operations, causing additional hotplug attempts to fail.
	Workaround: Wait for the Windows OS to complete the reboot process before submitting another `virtio_blk_controller_hotplug` RPC to expose an additional hotplug device.
	Keywords: Windows OS, hotplug, response timeout
	Reported in version: 4.5.0
4206444	Description: The Linux kernel driver does not restrict `seg_max` based on the maximum `queue_size` reported by the device. This can lead to a `WARN_ON_ONCE()` trigger in the kernel, resulting in driver misbehavior and potential system hangs.
	Workaround: Ensure that all `virtio-blk` controllers are configured with a `seg_max` value smaller than `(queue_size-2)`.
	Keywords: Virtio-blk; kernel driver; `seg_max`; `queue_size`
	Reported in version: 4.6.0
-	Description: When using SRIOV with VFs sharing the same driver as their PF (sriov_driver_autoprobe=1) - unprobing the driver may take a long time, and some admin commands might get timeout
	Workaround: N/A
	Keywords: SRIOV; driver.
	Discovered in version: 3.8.0-8

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation (“NVIDIA”) makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer (“Terms of Sale”). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Trademarks

NVIDIA and the NVIDIA logo are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

© Copyright 2025, NVIDIA. PDF Generated on 05/05/2025