DOCA NVMe Emulation Application Guide

Table of contents

Running the Application

This document provides an NVMe emulation implementation on top of the NVIDIA® BlueField® DPU.

Introduction

The NVMe emulation application exhibits how to use the DOCA DevEmu PCI Generic API along with SPDK to emulate an NVMe PCIe function using hardware acceleration to fully emulate the storage device.

Application Architecture

Brief Introduction to NVMe

NVMe (Non-Volatile Memory Express) is a high-performance storage protocol designed for accessing non-volatile storage media. NVMe operates over the PCI Express bus, which provides a direct and high-speed connection between the CPU and storage. It enables significantly lower latency and higher input/output operations per second compared to older storage protocols. NVMe achieves this through its scalable, multi queue architecture, allowing thousands of I/O commands to be processed in parallel.

NVMe emulation is a mechanism that allows NVMe device behavior to be simulated in a virtualized or development environment without the need for physical NVMe hardware.

Controller Registers

NVMe controllers feature several memory-mapped registers located within memory regions defined by the Base Address Registers (BARs). These registers convey the controller's status, enable the host to configure operational settings, and facilitate error reporting.

Key NVMe controller registers include:

- CC (Controller Configuration) – Configures the controller's operational parameters, including enabling or disabling it and specifying I/O command sets.
- CSTS (Controller Status) – Reports the controller's status, including its readiness and any fatal error.
- CAP (Capabilities) – Details the capabilities of the controller.
- VS (Version) – Indicates the supported NVMe version.

- AQA (Admin Queue Attributes) – Specifies the sizes of the admin queues.

Initialization

Initializing an NVMe controller involves configuring the controller registers and preparing the system to communicate with the NVMe device. The process typically follows these steps:

1. The host clears the enable bit in the CC register (sets it to 0) to reset the controller.
2. The host configures the controller by setting initial parameters in the CC register and enables it by setting the enable bit.
3. The host sets up the admin submission and completion queues for handling administrative commands.
4. Administrative commands are issued to retrieve namespace information and perform other setup tasks.
5. The host creates I/O submission and completion queues, preparing the controller for I/O operations.

Reset and Shutdown

Reset and shutdown operations ensure the controller is properly handled and data integrity is maintained.

- The reset process sets the enable bit in CC register to 0, stopping all I/O operations and clearing the controller's state to ensure it returns to a known state.
- Shutdown is initiated by setting the Shutdown Notification (SHN) field in the CC register. This allows the controller to halt operations, flush caches, and ensure data in flight is safely handled before powering off or resetting.

Completion Queue (CQ)

The Completion Queue in NVMe stores entries that the controller writes after processing commands. Each entry in the CQ corresponds to a command submitted through a Submission Queue, and the host checks the CQ to track the status of these commands.

CQs are implemented as circular buffers in host memory. The host can either poll the CQ or use interrupts to be notified when new entries are available.

Completion Queue Element (CQE)

Each completion queue element in an NVMe Completion Queue (CQ) is an individual entry that contains status information about a completed command, including:

- CID – Identifier of the completed command.
- SQID – ID of the Submission Queue from which the command was issued.
- SQHD – Marks the point in the Submission Queue up to which commands have been completed.
- SF – Indicates the status of the completed command.
- P – Flags whether the completion entry is new.

Submission Queue

The submission queue (SQ) is where the host places admin and I/O commands for the controller to execute. It operates as a circular buffer, and each SQ is paired with a Completion Queue (CQ). NVMe supports multiple SQs, with each one assigned to a specific CQ.

The controller is notified of new commands via the doorbell mechanism.

Submission Queue Element (SQE)

A Submission Queue Element (SQE) is an individual entry in a Submission Queue that represents a command for the NVMe controller to process. Each SQE contains:

- CID – A unique identifier for the command
- OPC – The opcode that specifies the operation to be performed
- PSDT – Specifies whether PRPs (Physical Region Pages) or SGLs (Scatter-Gather Lists) are used for data transfer associated with the command
- NSID – The identifier of the target namespace for the command

Additionally, the SQE includes fields for command-specific details, such as logical block addresses and transfer sizes.

Admin Queue Pair (Admin QP)

The Admin Queue Pair consists of an Admin Submission Queue (SQ) and an Admin Completion Queue (CQ), both assigned a command identifier (CID) of 0. There is only one Admin Queue Pair (QP) for each NVMe controller, and it is created asynchronously during the initialization phase. Unlike I/O queues, this QP is dedicated solely to controller management tasks, facilitating the processing of administrative commands.

Admin Commands

- **Identify Command (** `SPDK_NVME_OPC_IDENTIFY 0x06` **)**

  This command allows the host to query information from the NVMe controller, retrieving a data buffer that describes attributes of the NVMe subsystem, the controller, or the namespace. This information is essential for host software to properly configure and utilize the storage device effectively.

- **Create I/O Submission Queue (** `SPDK_NVME_OPC_CREATE_IO_SQ 0x01` **)**

This command allows the host to instruct the NVMe controller to establish a new queue for submitting I/O commands.

Key Parameters:

-
  - SQID – Identifies the specific I/O Submission Queue being created.
  - Queue Depth – Specifies the number of entries the queue can hold.
  - CQID – The identifier of the associated Completion Queue.

Once the host sends the command with the necessary parameters, the controller allocates the required resources and returns a status indicating success or failure.

- **Delete I/O Submission Queue (** `SPDK_NVME_OPC_DELETE_IO_SQ 0x00` **)**

This command is used to remove an I/O Submission Queue when it is no longer required.

Key Parameters:

-
    - SQID – The identifier of the I/O Submission Queue to be deleted.

Once the host issues the command, the controller releases all resources associated with the queue and returns a status confirming the deletion. After the queue is deleted, no additional I/O commands can be submitted to it.

- **Create I/O Completion Queue (`SPDK_NVME_OPC_CREATE_IO_CQ 0x05`)**

This command is issued by the host to set up an I/O completion queue in the NVMe controller.

Key Parameters:

-
    - CQID – The identifier of the I/O Completion Queue to be created.
        - Queue Depth – The number of entries the completion queue can hold.
    - PRP1/PRP2 – Pointers to the memory location where the CQ entries are stored.
    - MSIX – The Interrupt vector associated with this CQ.

Once the host issues the command, the controller allocates the necessary resources, links the CQ to the specified interrupt vector, and returns a status confirming the creation.

- **Delete I/O Completion Queue (`SPDK_NVME_OPC_DELETE_IO_CQ 0x04`)**

This command is issued by the host to remove an existing I/O Completion Queue from the NVMe controller.

Key Parameters:

-
    - CQID – The identifier of the I/O Completion Queue to be deleted.

Upcoming receiving this command, The NVMe controller removes the specified CQ and frees all associated resources. Before a CQ can be deleted, all SQs linked to it must be either deleted or reassigned to another CQ. The Controller returns a status code

indicating whether the deletion was successful or if an error occurred. Once deleted, the CQ no longer processes completions entries from any linked SQ.

- **Get Feature (`SPDK_NVME_OPC_GET_FEATURES 0x0A`)**

  This command is issued by the host to query specific features supported by the NVMe controller.

  Key Parameters:

  - FID (Feature ID) – Specifies which feature the host wants to retrieve. The controller returns information based on the requested feature. Common features include:

    - Arbitration (FID 0x01)
    - Power Management (FID 0x02)
    - Temperature Threshold (FID 0x04)
    - Error Recovery (FID 0x05)
    - Volatile Write Cache (FID 0x06)
    - Number of Queues (FID 0x07)
    - Interrupt Coalescing (FID 0x08)

    Depending on the FID, the feature information may be returned in the Completion Queue Entry (CQE) or written to an output buffer in host memory. If an output buffer is used, the host provides a memory region that the controller accesses via PRP entries or SGL.

-

  - Select – Determines which version of the feature value to return. There are four options:

    - Current (0x0) – Returns the active value of the feature
    - Default (0x1) – Returns the default value of the feature
    - Saved (0x2) – Returns the saved value from non-volatile memory

        - Supported Capabilities (0x3) – Returns the capabilities supported by the controller for the feature

    After executing the command, the controller returns a status code in the CQE indicating whether the query was successful.

- **Set Feature (** `SPDK_NVME_OPC_SET_FEATURES 0x09` **)**

    This command is issued by the host to modify specific features on the NVMe controller.

    Key Parameters:

    - FID (Feature ID): Specifies which feature the host intends to modify. Common features include:
        - Arbitration (FID 0x01)
        - Power Management (FID 0x02)
        - Temperature Threshold (FID 0x04)
        - Error Recovery (FID 0x05)
        - Volatile Write Cache (FID 0x06)
        - Number of Queues (FID 0x07)
        - Interrupt Coalescing (FID 0x08)
    - Data location: depending on the FID, the new value can be provided directly in the SQE or stored in an input buffer located in host memory and is accessible by the controller via PRP or SGL.
    - Save: this field allows the host to specify whether the modification should persist after a controller reset, if set the modified value is saved in the controller non-volatile memory.

    After the command is issued and the controller modifies the feature as requested, it returns a status code in the CQE, indicating whether the modification was successful or if an error occurred.

- **Log Page (** `SPDK_NVME_OPC_GET_LOG_PAGE 0x02` **)**

This command is issued by the host to retrieve various types of log pages from the NVMe controller for monitoring and diagnosing the state of an NVMe device.

Key Parameters:

- LID: log page identifier that specifies the type of log page to retrieve. some common log pages include:
  - SMART / Health Information (LID 0x02): provides device health metrics, temperature, available spare, and more.
  - Error Information (LID 0x01): contains details about errors encountered by the controller.
  - Firmware Slot Information (LID 0x03): information on the firmware slots and active firmware.
  - Telemetry Host-Initiated (LID 0x07): contains telemetry data about device performance.
- NUMD: the number of DWORDs of log data to return. this allows partial or full-page retrieval.
- Log page data location: the retrieved log data is written into the output buffer provided by the host. The buffer is accessible by the controller via PRP or an SGL.

When the host issues the get log command, the controller retrieves the requested log information and writes it to the host-provided memory. The controller then returns a status code in the CQE, indicating the success or failure of the operation.

Input/Output Queue Pair (I/O QP)

An I/O Queue Pair consists of one Submission Queue and its corresponding Completion Queue, which are used to perform data transfers (I/O operations). Multiple I/O Queue Pairs can be created to enable parallel I/O operations. Each queue pair functions independently, maximizing the use of multi-core processors.

- I/O Submission Queue: where the host places read/write/flush commands
- I/O Completion Queue: where the controller posts completion entries after processing the commands

NVM Commands

- **Flush (** `SPDK_NVME_OPC_FLUSH 0x00` **)**

  The NVMe Flush Command is issued by the host to ensure that any data residing in volatile memory is securely written to permanent storage. If no volatile write cache is present or enabled, the Flush command completes successfully without any effect. Once the Flush operation is finished, the controller updates the associated I/O Completion Queue with a completion entry.

- **Read** (`SPDK_NVME_OPC_READ 0x02` **)**

  The NVMe read command is one of the core I/O operations in NVMe. This command is issued by the host to retrieve data from a specified namespace and transfer it to the host memory.

Key Parameters:

-
  - NSID – the identifier of the namespace from which the data is being read.
  - LBA – the starting address of the data to be read within the namespace.
  - NLB (Number of LBAs) – specifies the size of the read operation in terms of the number of logical blocks to be read.
  - Destination buffer – The controller gets the read data from the namespace and sends it to the host memory, where the destination buffer is specified using PRP or SGL.

  Upon completion, the controller posts a completion entry to the I/O Completion Queue which includes a status code indicating success or failure.

- **Write (** `SPDK_NVME_OPC_WRITE 0x01` **)**

  The write command is also one of the core I/O operations in NVMe. this command is issued by the host to write data to a specific namespace at a given logical block address (LBA).

Key Parameters:

-

- NSID – The identifier of the namespace where the data is being written
- LBA – The destination address within the namespace where data is written
- NLB – Specifies the size of the write operation in terms of the number of logical blocks to be written
- Source buffer – The data to be written is located in host memory, and the controller reads this data from the source buffer, which is provided using PRP or SGL.

Upon completion, the controller posts a completion entry to the I/O Completion Queue, which includes a status code indicating the success or failure of the write operation.

Brief Introduction To SPDK

The Storage Performance Development Kit (SPDK) is an open-source framework that offers tools and libraries for building high-performance, scalable storage solutions, particularly for NVMe devices. SPDK allows applications to implement transport protocols for NVMe over Fabrics (NVMe-oF) by bypassing the kernel and using user-space drivers, enabling direct interaction with storage hardware. This approach significantly reduces latency and overhead, making it ideal for demanding storage environments.

A key component of SPDK is its highly optimized NVMe driver, which operates entirely in user space. By allowing direct communication with NVMe devices without involving the kernel, this driver minimizes I/O latency and boosts performance, supporting both local NVMe storage and remote NVMe devices over NVMe-oF.

SPDK Threading Model

SPDK's threading model is designed for high concurrency, scalability, and low-latency I/O processing. It operates on a cooperative multitasking model where SPDK threads are assigned to pollers and tasks are executed entirely in user space without kernel involvement. Each SPDK thread runs on a dedicated CPU core, ensuring minimal context switching and enabling tight control over workloads in a non-preemptive environment.

Reactor Threads

At the core of this model are reactor threads, which are SPDK's main execution threads responsible for handling I/O processing and application logic. Each reactor thread is bound to a specific core and runs in polling mode, meaning it continuously polls for tasks and I/O requests rather than relying on interrupts.

Example:

```
struct spdk_thread *thread = spdk_get_thread();
```

This function retrieves the current SPDK thread, which is mapped to a specific core.

SPDK provides the ability to register pollers to reactor threads, which are periodic functions designed to complete asynchronous I/O operations, manage RPC servers, or perform custom operations. Once a poller completes an operation, it can trigger a user-defined callback to finalize the task.

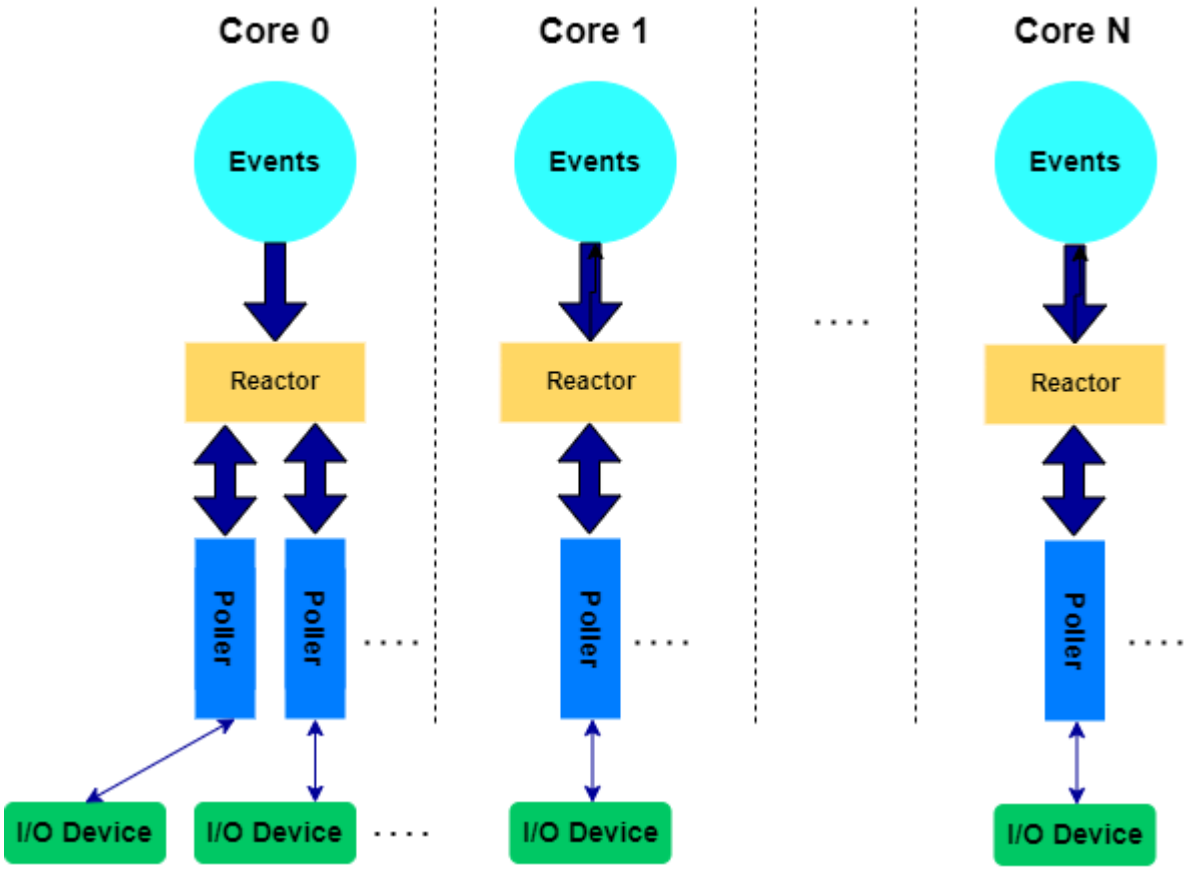
Example of registering a poller:

```
struct spdk_poller *my_poller =
spdk_poller_register(my_poll_function, arg, poll_interval_us);
```

Here, `my_poll_function` is invoked repeatedly by the reactor thread.

Poll Groups

Poll Groups consist of multiple SPDK threads working together to manage I/O across multiple devices or connections. A poll group allows a set of threads to coordinate and handle shared workloads, ensuring efficient distribution of tasks across available cores with minimal latency.



Thread Synchronization

In SPDK's cooperative threading model, thread synchronization is designed to be efficient and minimal, as threads do not experience preemptive context switching like traditional kernel threads. This allows for fine-grained control over when tasks are executed. However, there are situations where coordination between threads becomes necessary, such as when handling shared resources or passing tasks between cores.

Instead of relying on traditional locking mechanisms, which can introduce performance bottlenecks due to contention, SPDK uses message passing as the primary method for thread communication. This involves sending events or tasks between threads through a lockless event ring, which allows for coordination without the overhead associated with locks.

Example of sending a message between threads:

```
void send_message(struct spdk_thread *target_thread, spdk_msg_fn
fn, void *arg) {
    spdk_thread_send_msg(target_thread, fn, arg);
}
```

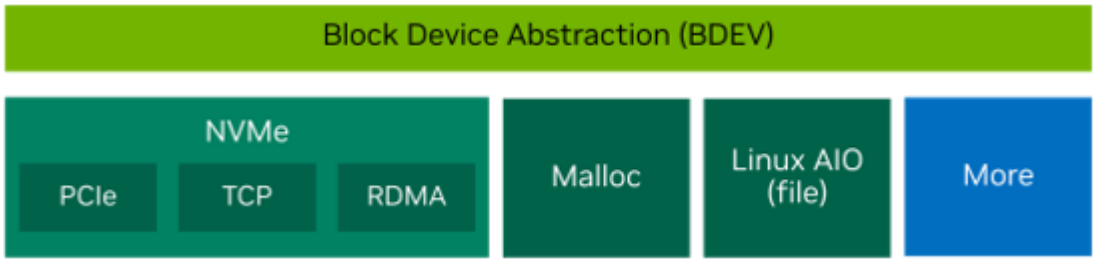
`target thread` is the SPDK we want to send a message to, `fn` is the function to be executed in the context of the target thread, and `arg` is the argument passed to the function.

Block Device

SPDK provides a flexible system for working with different types of storage devices, like NVMe SSDs, virtual block devices, AIO devices, and RAM disks. It uses high-performance, user-space APIs to let applications bypass the operating system's kernel, reducing delays and improving performance.

SPDK's block device (bdev) layer gives applications a unified way to perform read and write operations on these devices. It supports popular devices out of the box and allows users to create custom block devices. Advanced features like RAID and acceleration with technologies like DPDK are also supported.

Block devices can be easily created or destroyed using SPDK's RPC server. In NVMe-oF environments, a block device represents a namespace, which helps manage storage across different systems. This makes SPDK ideal for building fast, scalable storage solutions.



NVMe-oF Target

The NVMe-oF target is a user-space application within the SPDK framework that exposes block devices over network fabrics such as Ethernet, InfiniBand, or Fibre Channel. It typically uses transport protocols like TCP, RDMA, or vfio-user to enable clients to access remote storage.

To use the NVMe-oF target, it is required to configure it to work with one of these transport protocols:

- TCP
- RDMA (over InfiniBand or RoCE)
- vfio-user (mainly for virtual machines)
- FC-NVMe (Fibre Channel, less common in SPDK environments)

NVMe-oF Transport

Each NVMe-oF transport plays a crucial role in facilitating communication between the NVMe-oF initiator (the client) and the target. The transport handles how NVMe commands are transmitted over the network fabric. For example:

- TCP uses IP-based addressing over Ethernet.
- RDMA leverages the low-latency, high-throughput characteristics of InfiniBand or RoCE.
- vfio-user provides virtualization support, allowing virtual machines to access NVMe devices.
- FC-NVMe uses Fibre Channel, often found in enterprise SAN environments.

Additionally, SPDK is designed to be flexible, allowing developers to create custom transports and extend the functionality of the NVMFS target beyond the standard transports provided by SPDK.

The NVMe-oF transport is responsible for:

- Establishing the connection between the initiator and the target.
- Translating network-layer commands into SPDK NVMe commands.
- Managing data transfer across the network fabric.

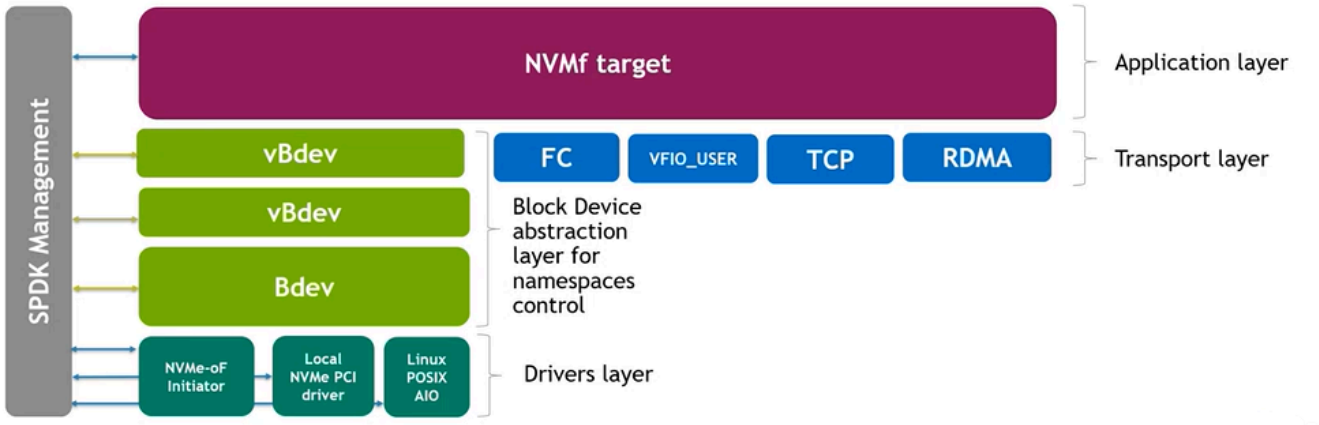
Once a connection is established through the transport, the NVMe-oF target processes the NVMe commands sent by the initiator.

Application Layer

Above the transport layer is the SPDK application layer, which is transport-agnostic. This means that regardless of the transport being used (TCP, RDMA, etc.), the application layer handles NVMe commands uniformly. It is responsible for:

- Managing subsystems, namespaces, and controllers.
- Processing NVMe commands received over the network.
- Mapping these commands to the appropriate storage devices (such as NVMe SSDs or virtual devices like SPDK's `malloc` or `null` devices).

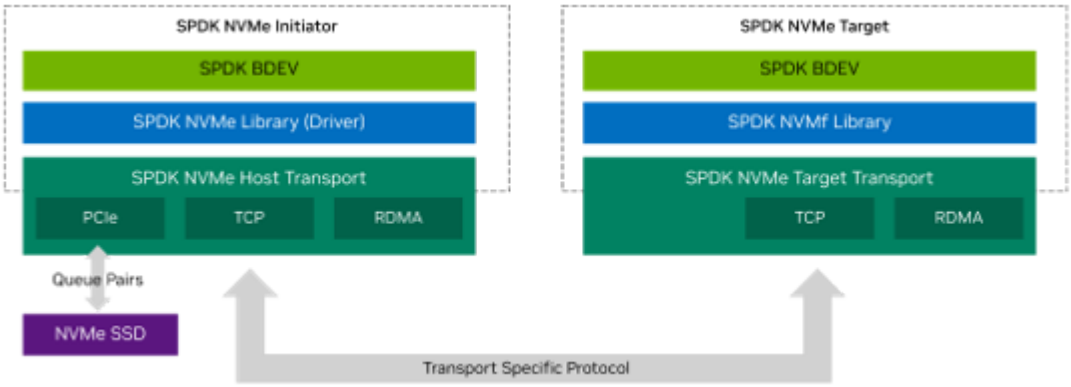
This uniform application layer ensures that the transport layer interacts with the same logic for processing and responding to NVMe commands, regardless of the underlying network fabric.



NVMe Driver in SPDK

In the context of NVMe storage, the initiator is the host system that needs to access an NVMe storage device to send data or retrieve data from an NVMe storage device, and the driver is responsible for generating and sending the appropriate commands such as read and write. the driver uses the transport to communicate with the NVMe target. the transport sends those commands either over a network in case of remote target or

directly to a local NVMe device through the PCIe bus. The target receives these requests from the initiator, processes them and responds back with the data or completion status.



RPC Server

SPDK's Remote Procedure Call (RPC) server provides a flexible interface for clients to interact with various SPDK services, such as NVMe-oF, block devices (bdevs), and other storage subsystems. The server is based on JSON-RPC and runs within the SPDK application, allowing external clients to dynamically configure, control, and manage SPDK components without the need for application restarts. Through RPC requests, users can create, delete, or query subsystems, configure network storage layers, and manage NVMe-oF targets. The main functionality of RPC server is to process incoming RPC commands that are usually in JSON format, to execute the functions based on the RPC requests and to return the result back to the client.

The RPC server runs within the SPDK application.

RPC Client

The RPC client in SPDK interacts with the RPC server to issue commands for configuring and managing various SPDK subsystems. It sends a command, typically in JSON format, asking the server to perform a certain task or retrieve data. The main functionality of RPC client is to construct the request message that it wants the server to process sends the request to the RPC server and receives the response from the server,

The RPC client runs on the user or application side.

Transport RPCs

SPDK provides several transport RPCs are used to configure and manage the transport layer for NVMe-oF. Here are the primary transport RPCs:

- `nvmf_create_transport`
- `nvmf_get_transports`
- `nvmf_subsystem_add_listener`
- `nvmf_get_subsystems`
- `nvmf_delete_listener`
- `nvmf_get_stats`
- `nvmf_delete_transport`

Block Device RPCs

SPDK provides several transport RPCs are used to configure and manage the transport layer for NVMe-oF. Here are the primary transport RPCs:

- `nvmf_create_transport`
- `nvmf_get_transports`
- `nvmf_subsystem_add_listener`
- `nvmf_get_subsystems`
- `nvmf_delete_listener`
- `nvmf_get_stats`
- `nvmf_delete_transport`

Namespace RPCs

Namespace RPCs are used to manage NVMe namespaces within NVMe-oF subsystems or NVMe controllers, here is a list of Namespaces RPCs available:

- `nvmf_subsystem_add_ns`
- `nvmf_subsystem_remove_ns`
- `nvmf_subsystem_get_ns`
- `nvmf_subsystem_get_ns_stats`
- `nvmf_subsystem_get_namespaces`

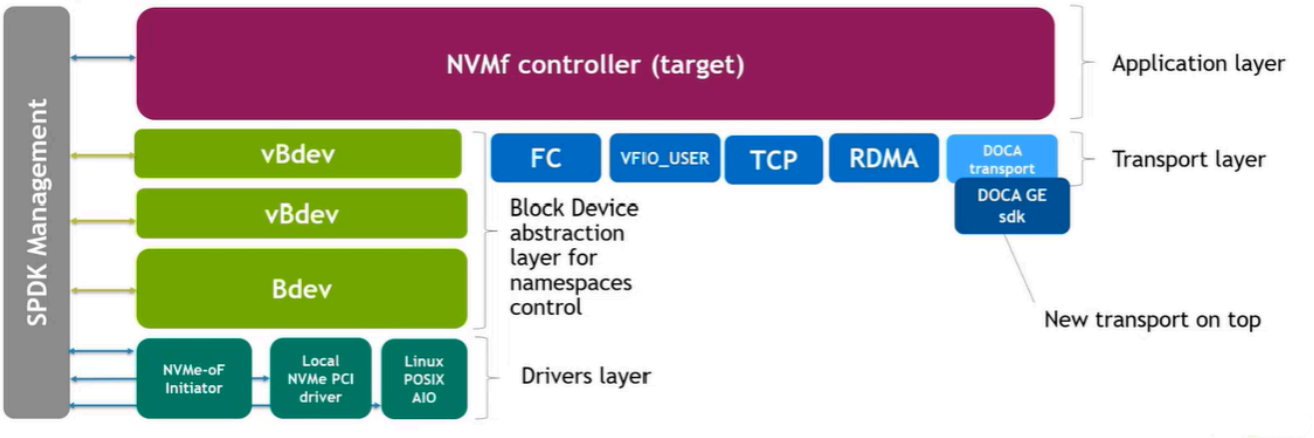
Solution Overview

Integration with SPDK

Using the DOCA Generic PCI Emulation SDK, the BlueField DPU can emulate NVMe devices through PCIe endpoints. This allows the DPU to appear as a physical NVMe device to the host system, enabling the host to send NVMe commands. While the DPU hardware can handle data movement and basic I/O tasks, managing the complete NVMe protocol, including admin commands, queue management, and register operations, requires additional software support. This is where SPDK comes in.

With SPDK, the DPU can offload the complex handling of NVMe commands, making it a fully functional and high-performance NVMe device without the need to develop custom firmware for each command.

DOCA Generic Device Emulation as NVMe-oF Transport



While NVMe-oF is designed for remote transports like TCP or RDMA, SPDK enables us to treat PCIe as another transport option by adding a memory-based transport. This allows the DPU to function as if it's communicating with a remote NVMe-oF target, even though it's local to the host system.

To implement this, we use a DOCA transport, a custom transport layer that acts as a connection tunnel and provides NVMe-oF with generic emulation capabilities. By leveraging SPDK's RPCs for NVMe-oF, we can create a DPU that effectively emulates an NVMe device. The DOCA transport ensures efficient routing and processing of NVMe commands, while SPDK takes care of the software-based emulation.

(The application utilizes the NVMe-oF application transport layer to implement an NVMe emulation solution, inspired by SPDK blogpost.)

Emulated Function as NVMe Controller

In the DOCA transport, the NVMe controller is mapped to a PCIe DOCA device known as the emulation manager. In this context, the emulation manager serves as the hardware interface that provides access to the NVMe controller, which exposes its capabilities through specific PCIe registers and memory-mapped regions.

When connecting a device to a controller, the transport is responsible for providing the controller's unique ID through the Connect command, as specified by the NVMe-oF protocol.

To make the core NVMe-oF target logic work with our DOCA transport, we need to implement specific operations in the `spdk_nvmf_transport_ops` structure. These operations handle tasks like managing connections, transferring data, and processing NVMe commands for DOCA. This structure provides a standard way to connect different

transports to SPDK, so the core NVMe-oF logic can work with any transport without needing to know its specific details.

```
const struct spdk_nvmf_transport_ops spdk_nvmf_transport_doca = {
    .name = "DOCA",
    .type = SPDK_NVME_TRANSPORT_CUSTOM,
    .opts_init = nvmf_doca_opts_init,
    .create = nvmf_doca_create,
    .dump_opts = nvmf_doca_dump_opts,
    .destroy = nvmf_doca_destroy,

    .listen = nvmf_doca_listen,
    .stop_listen = nvmf_doca_stop_listen,
    .listen_associate = nvmf_doca_listen_associate,

    .poll_group_create = nvmf_doca_poll_group_create,
    .get_optimal_poll_group = nvmf_doca_get_optimal_poll_group,
    .poll_group_destroy = nvmf_doca_poll_group_destroy,
    .poll_group_add = nvmf_doca_poll_group_add,
    .poll_group_remove = nvmf_doca_poll_group_remove,
    .poll_group_poll = nvmf_doca_poll_group_poll,

    .req_free = nvmf_doca_req_free,
    .req_complete = nvmf_doca_req_complete,

    .qpair_fini = nvmf_doca_close_qpair,
    .qpair_get_listen_trid = nvmf_doca_qpair_get_listen_trid,
};
```

New SPDK RPCs

Since the DOCA transport requires specific configurations that are not covered by the existing SPDK RPCs and differ from other transports, such as managing emulation managers, we need to implement custom RPCs to expose these options to users:

RPC	Description	Details	Arguments	Output	Example

`nvmf_doca_get_managers`	Provides the ability to list all emulation managers, which are equivalent to DOCA devices	Returns the names of all available local DOCA devices with management capabilities.	None.	If successful, the RPC returns a list of device names for the emulation managers. If it fails, it returns an error code.	`/usr/bin/spdk_rpc.py --plugin rpc_nvmf_doca nvmf_doca_get_managers`
					`[`
					`  {`
					`  "name": "mlx5_0"`
					`  }`
					`]`

`nvmf_doca_create_function`	Provides the ability to create an emulated function under a specified device name.	Creates a new representor device, retrieves its VUID, and then closes the device.	• `-d` – Device name (string)	If successful, the RPC returns the VUID of the newly created function. In case of failure, it returns an error code.	`/usr/bin/spdk_rpc.py --plugin rpc_nvmf_doca nvmf_doca_get_managers`
					`[`
					`  {`
					`  "name": "mlx5_0"`
					`  }`
					`]`

RPC	Description	Details	Arguments	Output	Example
`nvmf_doca_destroy_function`	Provides the ability to destroy an emulated function.	Destroys a DOCA device representor.	• `-d` – The device name associated with the created function (string) • `-v` – The VUID of the function to be destroyed (string)	On success, the RPC returns nothing. On failure, it returns an error code.	`/usr/bin/spdk_rpc.py --plugin rpc_nvmf_doca nvmf_doca_destroy_function -d mlx5_0 -v MT2306XZ00AYGES2D0F0`

`nvmf_doca_list_functions`	Lists all the emulated functions under the specified device name.	Lists all the available representor devices.	• `-d` – Device name (string)	If successful, the RPC returns a list containing the VUID and PCIe address of all the emulated functions under the specified device name. In case of failure, it returns an error code.	`/usr/bin/spdk_rpc.py --plugin rpc_nvmf_doca nvmf_doca_list_functions -d mlx5_0`
					`[`
					`  {`
					`  "Function VUID": "MT2306XZ00AYGES1D0F0",`
					`  "PCI Address":`

RPC	Description	Details	Arguments	Output	Example

					"0000:00:00.0"
					},
					{
					"Function VUID": "MT2306XZ00AYGES2D0F0",
					"PCI Address": "0000:00:00.0"
					}
					]

Note

`/usr/bin/spdk_rpc.py` is the Python script that sends RPC commands to SPDK. The `spdk_rpc.py` script is responsible for handling SPDK commands via the JSON-RPC interface.

Extended RPCs

Some existing SPDK RPCs need to be modified because certain configurations or capabilities of the DOCA transport are not supported by the default RPCs:

RPC	Description	Details	Arguments	Output	Example
`nvmf_create_transport`	Creates a new NVMe-oF transport by defining the transport type and configuration parameters, allowing the SPDK target to communicate with hosts using the specified transport.	Creates DOCA transport and its resources.	• `-t` – The transport type (string) • `-u` – The number of I/O queue pairs per system. (Optional) • `-i` – The maximum I/O size in bytes. (Optional) • `-c` – The maximum of inline data size in bytes. (Optional)	None	`/usr/bin/spdk_rpc.py nvmf_create_transport -t doca -u 8192 -i 131072 -c 819`
`nvmf_subsystem_add_listener`	Adds a listener to an NVMe-oF subsystem, enabling it to accept connections over a specified transport.	Hot-plugs the device, allowing the host to interact with it as an NVMe device.	• The subsystem NQN (NVMe qualified name) to which to add the listener. • `-t` – The transport type for this listener. • `-a` – The VUID of	None	`/usr/bin/spdk_rpc.py nvmf_subsystem_add_listener nqn.2016-06.io.spdk:cnode1 -t doca -a MT2306XZ00AYGES1D0F0`

RPC	Description	Details	Arguments	Output	Example
			the function we want to use to create the emulated device.		

Note

`/usr/bin/spdk_rpc.py` is the Python script that sends RPC commands to SPDK. The `spdk_rpc.py` script is responsible for handling SPDK commands via the JSON-RPC interface.

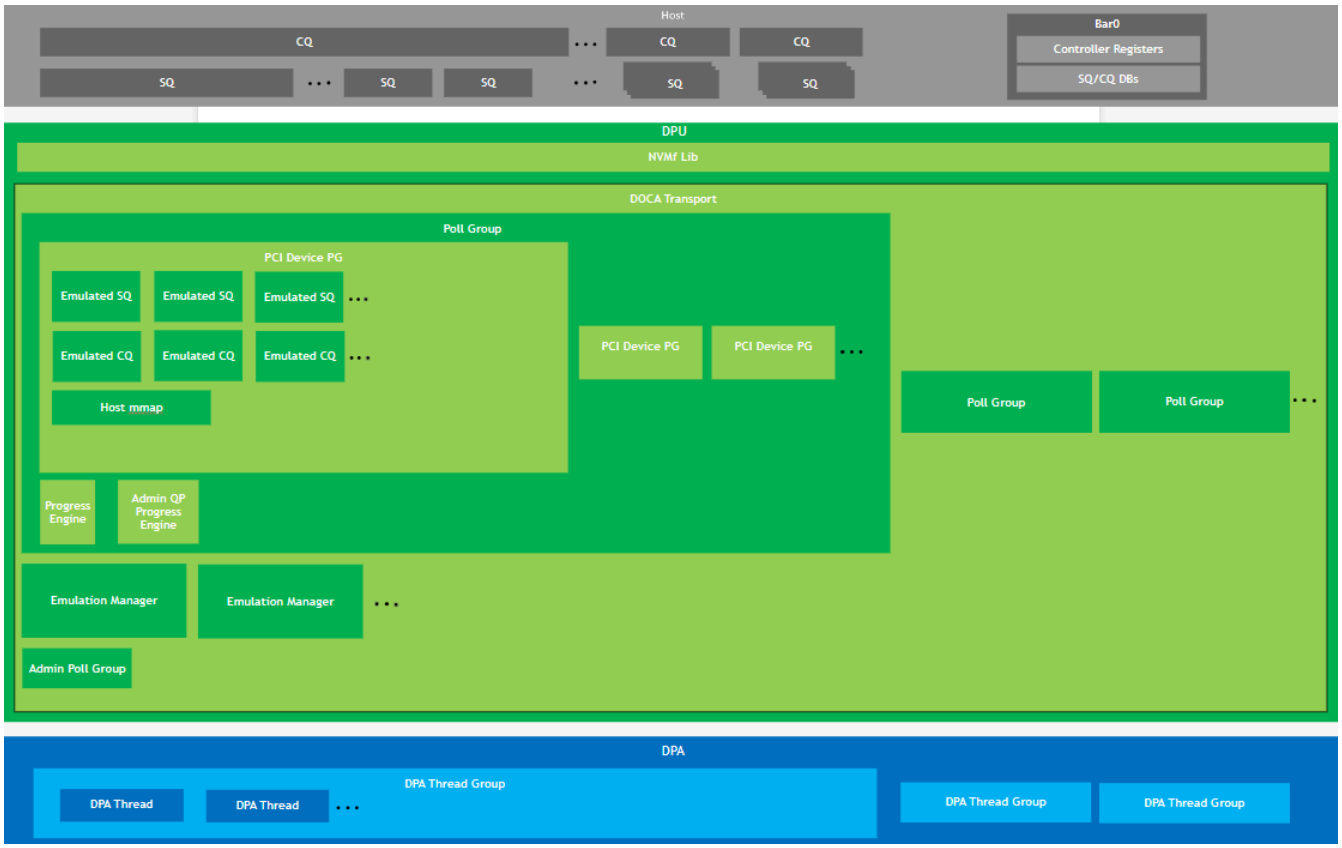
Data Structures

To implement the APIs for `spdk_transport_ops` mentioned above, we created transport-specific data structures that efficiently interact with these APIs. These structures are designed to manage the transport's state, connections, and operations.

The upper layer represents the host, followed by the DPU, and further down is the DPA, which is part of the DPU. Within the DPU, the NVMe-oF application is running, divided into two parts: the NVMe-oF library, which we use as a black box, and the DOCA transport, which we implement.

In the DOCA transport, there are several poll groups, each representing a thread. In addition to the poll group list, the transport maintains an emulation managers list, consisting of all devices managed by this transport. There is also a special poll group instance dedicated to polling PCIe events.

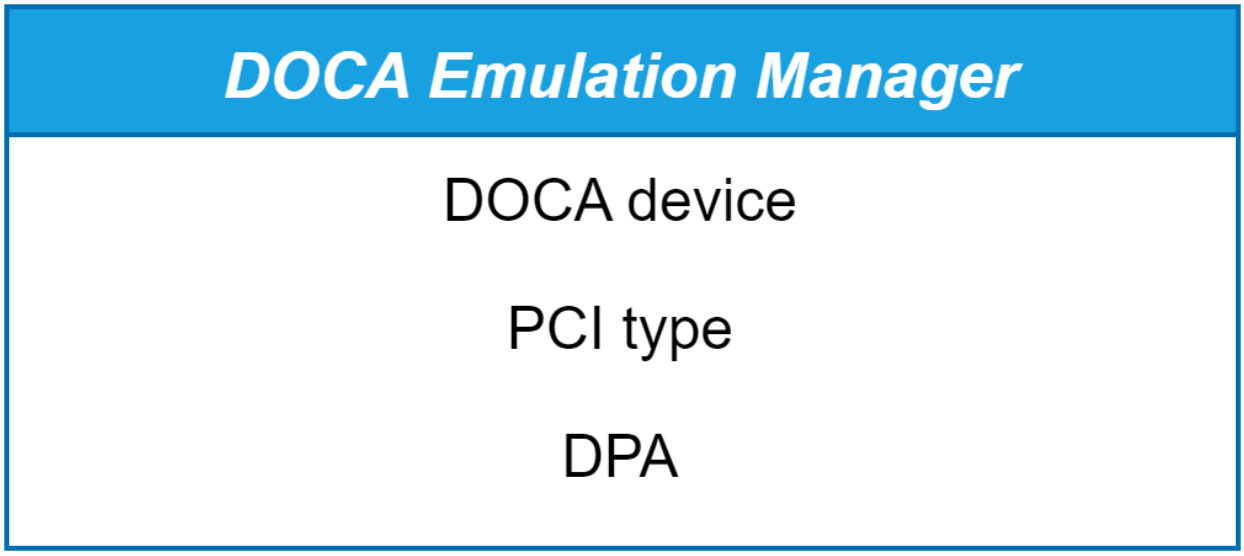
If we dive into the poll group, there are two progress engines: one that polls the I/O queues and another that handles the admin queues. Additionally, there is a list of PCIe device poll groups, each of which manages the completion queues, submission queues, and host memory mappings for a specific device.



The following sections provide more information on each of the main structures.

Emulation Manager

The first structure to address is the emulation manager context. During initialization, the DOCA transport scans for all available DOCA devices that can serve as emulation managers. For each of these devices, it creates a PCIe type, initializes a DPA instance, assigns a DPA application, and starts the DPA process. It also opens the DOCA device (emulation manager). All of these devices are stored within the emulation manager context, where they are tracked for as long as the transport is active.



PCIe Device Admin

This is the representor of the emulated device. and it contains the following:

DOCA PCI Device Admin
Doca transport
Emulation managers
PCI device
SPDK subsystem
Device representor
Transport ID
Doca Listener state
Controller ID
Stateful region values
SDPK NVMF controller
Doca admin QP
Admin QP poll group
FLR flag
Destroy flow falg

- A pointer to the DOCA transport this device belongs to.
- The emulation manager described previously
- The emulated PCIe device
- A pointer to the SPDK subsystem this device belongs to
- The transport ID this device belongs to
- A pointer to SPDK NVMe-oF controller
- The stateful region values and are updated after each query
- Admin Queue Pair context
- Admin Queue Pair poll group that manages the admin QP, and it is selected from the system using the round-robin method
- The FLR flag Indicates if an FLR event has occurred

- Destroy flag that indicates if PCIe device should be destroyed

Once the user issues the add listener RPC, this context is established to facilitate the hot-plug of the emulated device to the host, enabling it to start listening for interactions from the host.

DOCA Poll Group

Each poll group in SPDK is associated with a thread running on a specific CPU core. For each core, a reactor thread is responsible for executing pollers, and the poll group is one of those pollers. When an NVMe-oF is created, it is assigned to each poll group so that the transport can handle I/O and connection management for the devices across multiple CPU cores, so each transport has a representative within each poll group.

DOCA poll group Fields:

DOCA Poll Group
SPDK NVMF transport poll group
Progress engine
Admin QP progress engine
Admin QP poll rate
Admin QP rate limiter
List of PCI device poll group

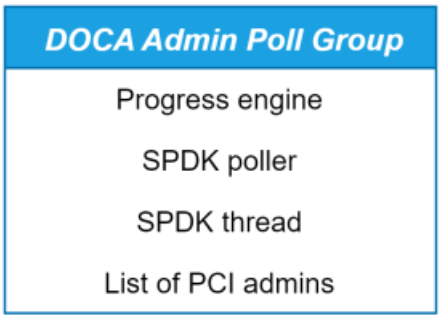
- SPDK NVMe-oF transport poll group (`struct spdk_nvmf_transport_poll_group`): refers to a structure that is part of the NVMe-oF subsystem and is responsible for handling transport specific I/O operations at the poll group level.

- Progress engine (`struct doca_pe`): When each poll group runs its set of pollers, it also invokes the DOCA progress engine to manage transport operations. On the DOCA side, the doca_pe_progress function is called to drive the progress engine within each poll group. This is how DOCA's PE integrates into SPDK's poll group mechanism.

- Admin QP progress engine (`struct doca_pe`): Another progress engine (`struct doca_pe`) dedicated to handling admin queues while the previous one is dedicated to handling the I/O queues. this separation allows for more control over the polling rate of each queue type, which helps optimize performance.
- Admin QP poll rate and Admin QP rate limiter: They determine how often the system checks the admin queue for new commands.
- List of PCIe device poll group: List of devices that this poll group typically polls for their queues.

Admin Poll Group

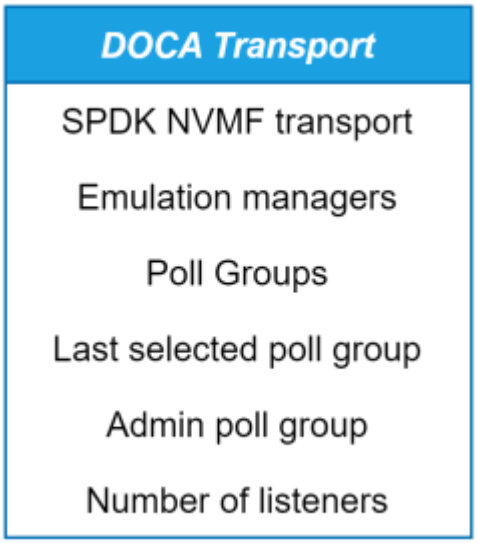
This object is a per-transport entity, functioning as a dedicated unit for polling PCIe events and managing Admin Queue activities.



- Progress engine – The DOCA used by the poller.
- SPDK Poller – This poller is used on SPDK application thread. It continuously monitors for PCIe events such as FLR, stateful region and hot plug events.
- SPDK thread – The application thread associated with the currently executing SPDK thread.
- PCIe admins – List of all the PCIe device admins.

DOCA Transport

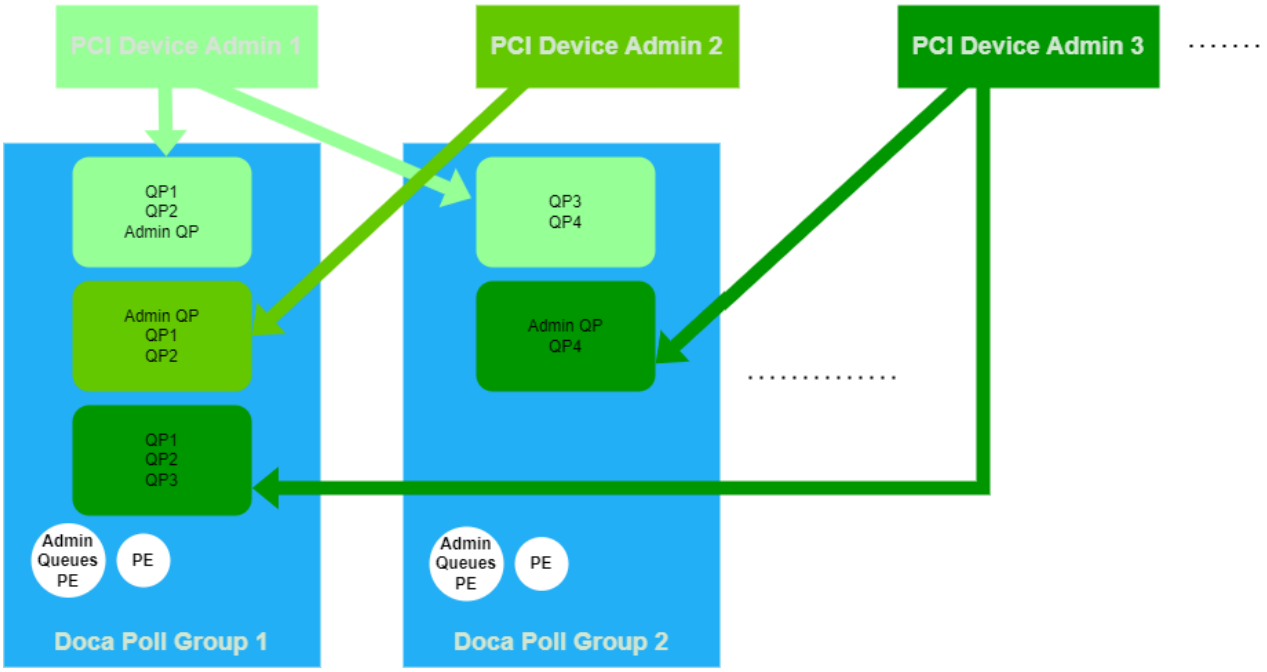
This structure holds the overall state and configuration of the transport as it includes:



- SPDK NVMe-oF transport – Defines the transport layer within SPDK NVMe-oF framework. It holds essential data for managing the transport, such as configuration parameters, operational states, and connections.
- Emulation managers – Includes all the devices managed by this transport
- Poll groups – Contains all the poll groups actively polling for this transport
- Last selected poll group – Used to assist with round-robin selection of poll groups, ensuring even distribution of workload across poll groups during transport operations
- Admin poll group – described previously
- Number of listeners – Number of devices within this transport

PCIe Device Poll Group

Based on the previous descriptions of the transport structures, the following diagram illustrates the relationship between PCIe devices and the poll groups within a transport. Each PCIe device contains I/O and admin queues, which are distributed across the poll groups.



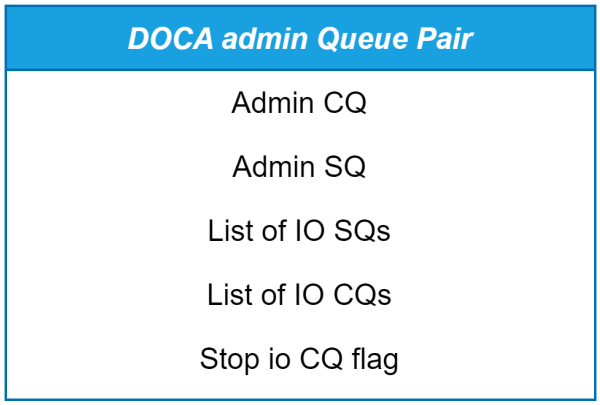
This relationship is managed by a structure called the PCIe dev poll group (`struct nvmf_doca_pci_dev_poll_group`), which holds the device's memory map (mmap), the PCIe device's admin details, admin QPs (if applicable), a list of I/O queues, and the poll group responsible for polling those queues.

DOCA PCI Dev Poll Group
Host mmap
PCI device admin
Admin QP
List of IO CQs
Parent Poll group

When creating a new I/O or admin queue for a specific device and poll group, we first check if a PCIe dev poll group structure already exists that links the two. If not, we create a new `struct nvmf_doca_pci_dev_poll_group` to combine them.

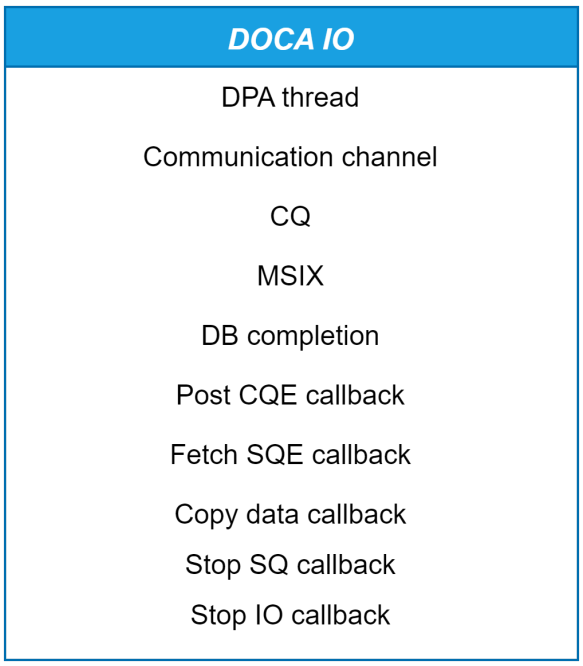
Admin QP

This structure manages the queues for a specific device. It holds an admin completion queue and admin submission queue to handle admin command operations. Additionally, it includes lists for I/O completion queues and I/O submission queues, which manage data-related operations. The structure also contains a flag `stopping_all_io_cqs` to indicate whether all completion queues should be stopped, used for gracefully halting the device's queue processing when needed.



DOCA IO

The I/O structure is responsible for managing I/O operations, including receiving doorbells on the completion queue (CQ) and its associated submission queues (SQs). It handles reading SQ entries (SQEs) from the host, writing SQEs back to the host, and raising MSI-X interrupts. This structure contains a single CQ, along with an MSI-X vector index that is raised by the DPA, and a doorbell completion that is polled by the DPA thread.



It also holds several callbacks:

- Post CQE – Invoked once CQE is posted to the host CQ, freeing resources afterward
- Fetch CQE – Invoked when an SQE is fetched from the host, parsing and executing the request
- Copy data – Invoked after data is copied to the host, completing the request and freeing resources
- Stop SQ – Invoked when an admin or I/O SQ is stopped, to fully release resources
- Stop IO – Invoked when an admin or I/O CQ is stopped, to complete resource cleanup

The function `nvmf_doca_io_create` synchronously creates an emulated I/O.

DOCA CQ

The CQ's main task is to write cookies to host. The main fields are:

DOCA CQ
SQ ID
DOCA queue
Doorbell
Consumer index (CI)
Producer index (PI)

- In case it is an Admin CQ then the CQ ID is zero.
- The DOCA queue is a shared structure that acts as both the completion queue (admin and I/O CQ) and submission queue (admin and I/O SQ), mirroring the host's queues with the same size. It is responsible for fetching Submission Queue Entries (SQEs) from the host and posting Completion Queue Entries (CQEs) back to the host. The NVMe driver issues commands through its submission queues and receives their completions via the completion queues. To facilitate efficient processing by the DPU, the DOCA queue leverages DMA to handle data transfers between the host and DPU in both directions. Each queue is equipped with a pointer to a DMA structure that contains the necessary resources for these operations. During initialization, DMA resources and local buffers are allocated based on the queue's size. The DOCA queue also maintains an array of tasks, with each task at index idx corresponding to and synchronized with the task at the same index in the host's queue. When DMA operations are required, these resources are utilized for data transfer. The below outlines its main fields:

DOCA Queue
Buffer Inventory
DMA
Local queue MMAP
Local queue address
Elements (DMA tasks)
Number of elements

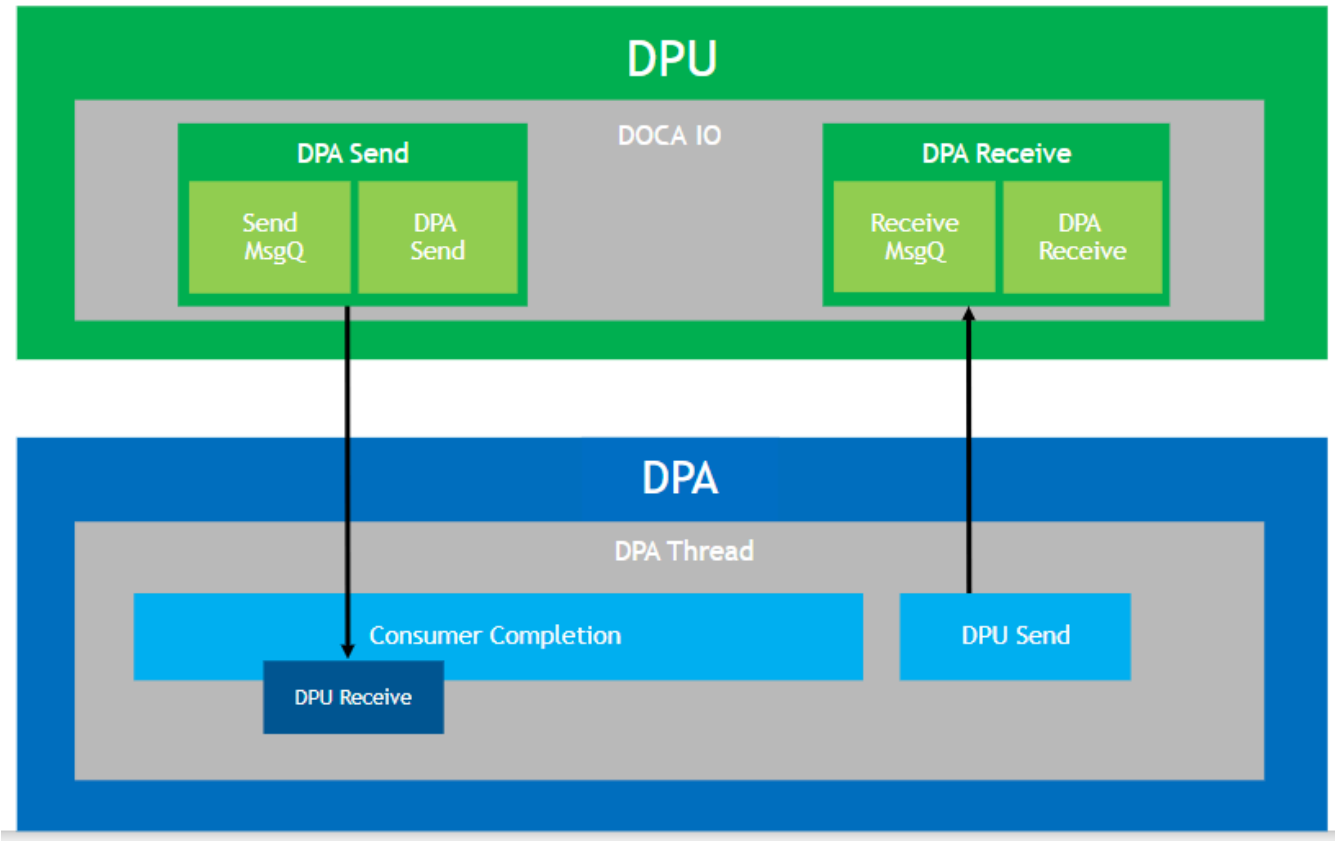
-
    - The buffer inventory is for allocating the queue elements

- DMA context handles data transfer to and from the host
- The queue MMAP represents the local memory where elements are stored for copying
- The local address for copying the elements to or from the host
- The elements themselves are DMA tasks for copy/write
- The maximum number of elements the queue can hold

DOCA Comch

A full-duplex communication channel is used to facilitate message passing between the DPA and DPU in both directions. This channel is contained within the DOCA IO and consists of two key components:

- A send message queue (`nvmf_doca_dpa_msgq`) with a producer completion (`doca_dpa_completion`) context.
- A receive message queue (`nvmf_doca_dpa_msgq`) with a consumer completion (`doca_comch_consumer_completion`) context.

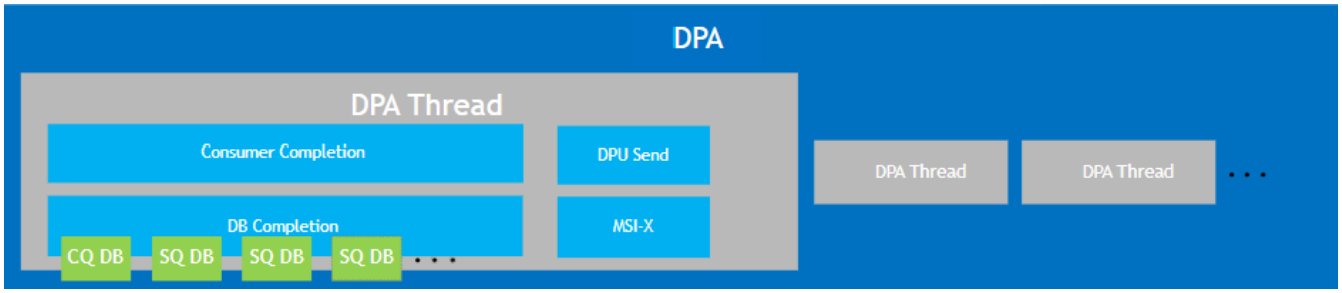


DOCA DPA Thread

In addition to the communication channel, the I/O structure also includes a DPA thread context, (with each thread dedicated to a single CQ). It consists of:

- A pointer to the DPA
- A pointer to the DPA thread
- The necessary arguments for the DPA thread, include:
    - Consumer Completion: Continuously polled by the DPA thread to detect new messages from the host.
    - Producer Completion: Monitored to verify if messages have been successfully sent.
    - DB Completion Context: Provides the doorbell values and connects to both CQ and SQ doorbells.

- MSIX: Allows the device to send interrupts to the host.

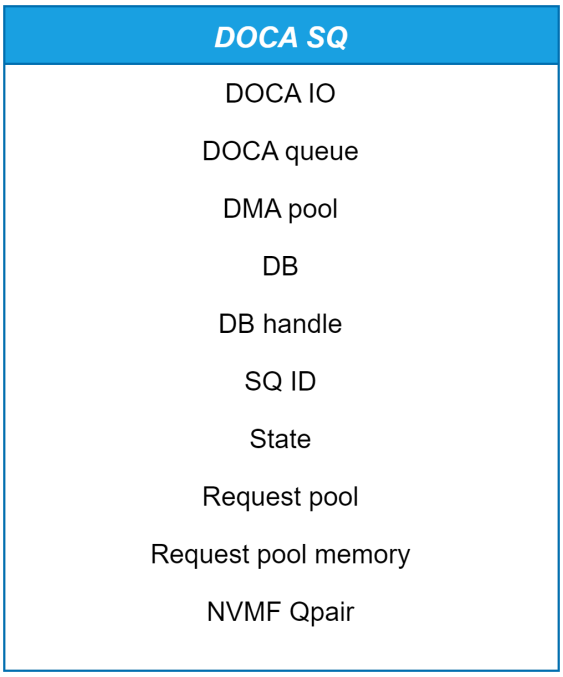


The DPA thread performs three main operations:

- `nvmf_doca_dpa_thread_create` – Creates a DPA thread by providing the DOCA DPA, the DPA handle, and the size of the arguments to be passed to the DPA. It also allocates memory on the device.
- `nvmf_doca_dpa_thread_run` – Copies the arguments to the DPA and runs the thread.
- `nvmf_doca_dpa_thread_destroy` – Deletes the thread and frees the allocated arguments."

DOCA SQ

The main fields that construct the DOCA submission queue (`nvmf_doca_sq`) include:

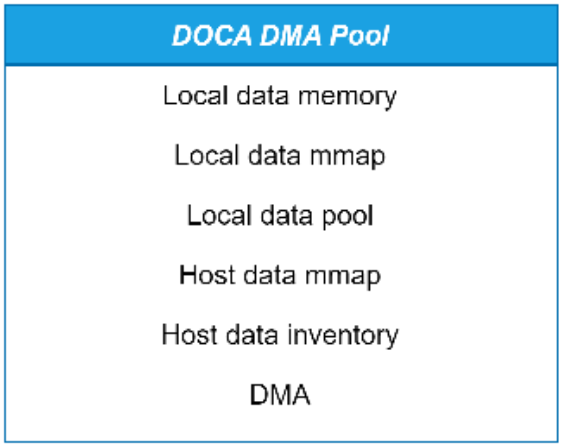


- A reference to the DOCA I/O to which this submission queue belongs, and where its completion is posted. Multiple SQs can belong to a single I/O.
- The DOCA queue, which was previously described, that handles copying Submission Queue Entries (SQEs) from the host.
- A pool of DMA (`nvmf_doca_dma_pool`) data copy operations (to be defined shortly).
- The doorbell associated with this submission queue.
- The DPA handle of the doorbell.
- The submission queue identifier (SQID).
- The state of the submission queue, used for monitoring purposes.
- NVMe-oF request pool memory (to be defined shortly).
- A list of NVMe-oF DOCA empty requests, used whenever a new SQE is received and a request needs to be prepared.
- The Queue Pair (QP) created for this submission queue by the NVMe-oF target, which is used to execute commands.

The submission queue creation is asynchronous because, when we create the completion queue (CQ), we also initialize the DPA. However, when creating the SQ, the DPA is already running, so we need to update the DPA about the newly added SQ. Directly modifying its state could lead to synchronization issues, which is why we use a communication channel, making the process asynchronous.

DOCA DMA Pool

As previously noted, a Submission Queue (SQ) includes the `nvmf_doca_dma_pool` structure, which manages data transfer operations between the host and the DPU, in both directions. It consists of the following elements:



- Memory allocated for local data buffers
- Memory-mapped region for the local data buffers
- A pool of local data buffers
- Memory mapping that provides access to host data buffers
- An inventory for allocating host data buffers
- A DMA context used for transferring data between the host and the DPU

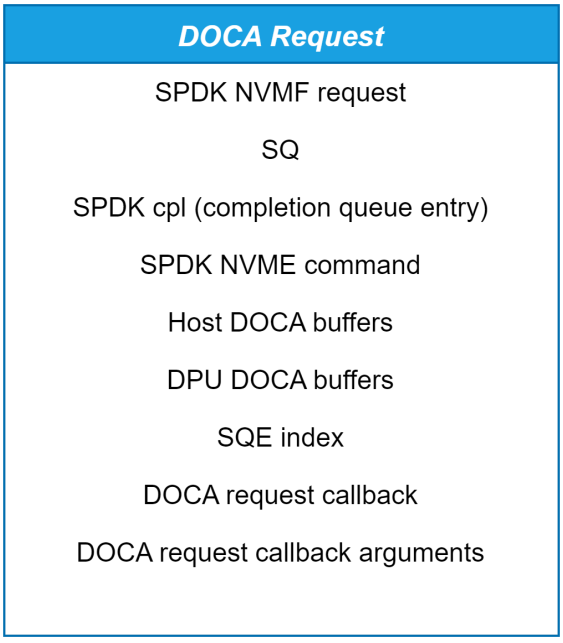
This structure is initialized whenever the SQ is created. The size of the local data memory is determined by multiplying the maximum number of DMA copy operations by the maximum size in bytes for each DMA copy operation. All these local buffers are allocated during the creation of the SQ. The key operations performed on the DMA are:

- `nvmf_doca_sq_get_dpu_buffer` – Retrieves a buffer in DPU memory, enabling data transfers between the host and the DPU.

- `nvmf_doca_sq_get_host_buffer` – Retrieves a buffer pointing to host memory, also used for data transfers between the host and the DPU.
- `nvmf_doca_sq_copy_data` – Copies data between the host and the DPU. This operation is asynchronous, and upon completion, it invokes the `nvmf_doca_io::copy_data_cb` callback function.

DOCA NVMe-oF Request

The NVMe-oF target utilizes requests to handle incoming commands. When a new command is received by any transport (not limited to the DOCA transport), it creates an instance of the struct `spdk_nvmsf_request`. This structure contains various elements, including the NVMe command, the Submission Queue (SQ) to which the command belongs, the Queue Pair (qpair), the IO Vector (IOV), and other relevant data. In our design, we introduce a new wrapper structure called `nvmf_doca_request`, which encapsulates the NVMe-oF request structure along with additional fields specific to the DOCA transport. The main fields included in this structure:



Where:

- SPDK request is and instance of struct `spdk_nvmf_request`, which represents the command being processed.

- Host and DPU DOCA buffers are pointers to data buffers located at the host or DPU, containing data associated with this command.
- DOCA request callback is the function invoked upon request completion, receiving the appropriate DOCA request callback arguments.

The key operations performed on requests include:

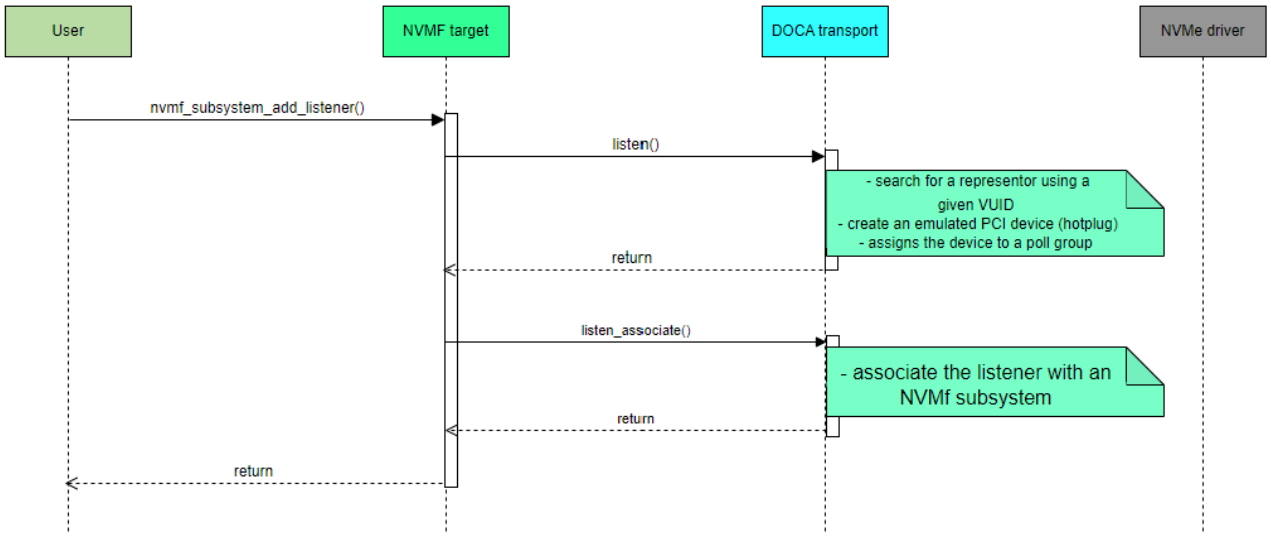
- `nvmf_doca_request_pool_create` – When the SQ is created, a request pool is allocated with a size matching the SQ depth.
- `nvmf_doca_request_pool_destroy` – This function destroys the request pool when the SQ is removed.
- `nvmf_doca_request_get` – Retrieves an NVMe-oF request object from the pool associated with a specific SQ, and is called after fetching an SQE from the host.
- `nvmf_doca_request_complete` – Completes an NVMe-oF request by invoking its callback and then releasing the request back to the pool.

Control Path Flows

DOCA Transport Listen

Hotplug and Hotunplug

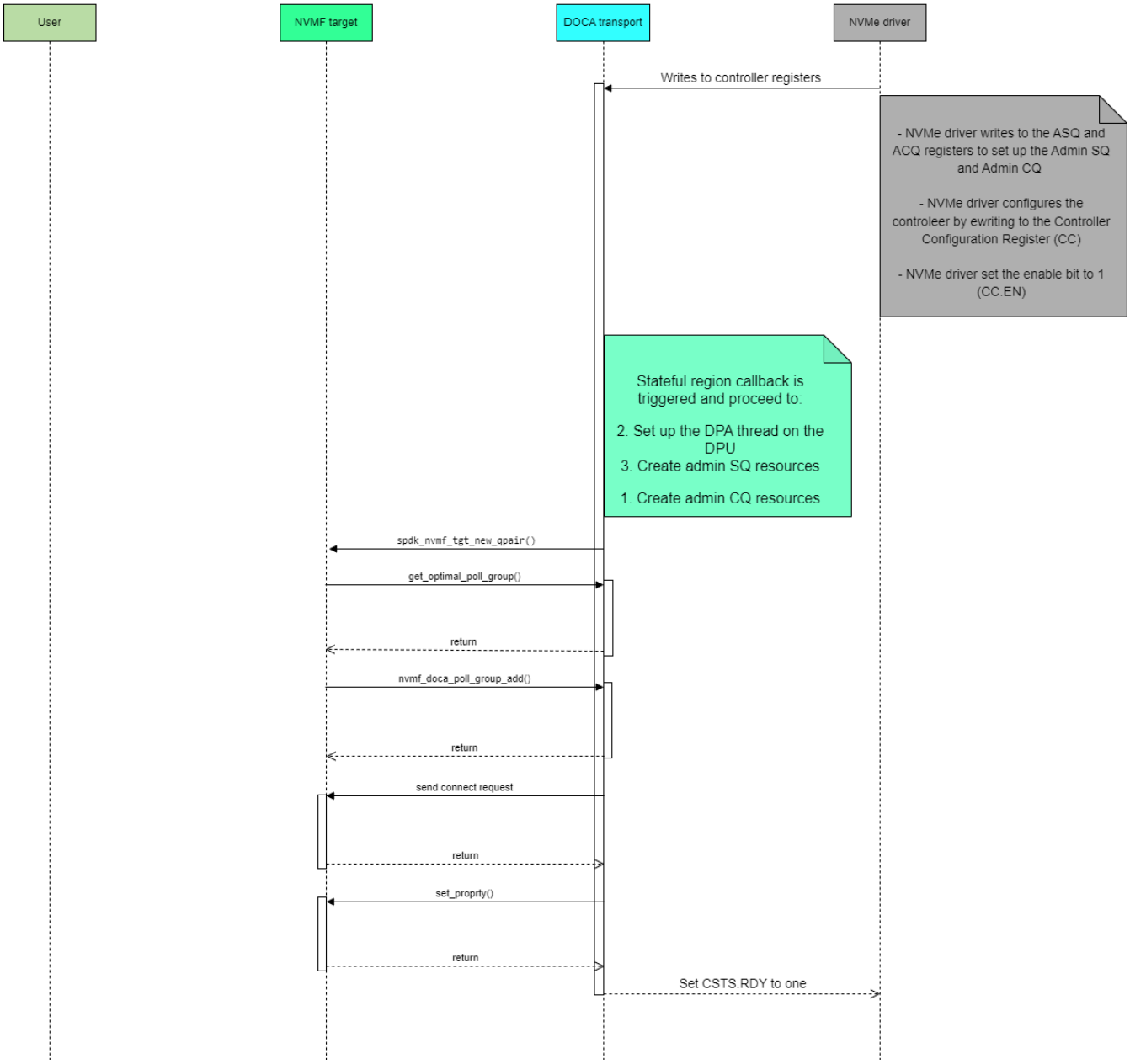
1. Start with `add_listener` RPC trigger – The flow begins when the `add_listener` RPC is called. SPDK then pauses all poll groups.
2. Lookup for representor by VUID – The transport searches for a representor using a given VUID.
3. Create emulated PCIe device – Once the representor is found, the transport creates an emulated PCIe device and assigns it to a poll group.
4. Initialize memory mapping (mmap) – For each poll group, the transport sets up a memory-mapped area representing the host memory.



At this stage, the NVMe driver detects the newly hot-plugged device.

Controller Register Events

Initialization



- Controller Initialization – The process begins by configuring the controller registers:
    - The NVMe driver writes to the ASQ and ACQ registers to set up the Admin Submission Queue and Admin Completion Queue.
    - The driver configures the controller by writing to the controller configuration (CC) register, setting parameters like memory page size, arbitration, and timeout values.
    - The driver sets the CC.EN bit in the CC register to 1, transitioning the controller from `Disabled` to `Enabled`.

    - The NVMe driver waits for the CC.RDY (Controller Status Register - Ready bit) to become 1. This indicates that the controller has successfully completed its initialization and is ready to process commands.

- Stateful Region Callback Trigger – At this point, the PCI-emulated device triggers a callback to the stateful region. This callback detects the host's initialization process by checking the changes in the enable bit. This callback may occur multiple times, but it only proceeds if the enable bit has been altered.

- CQ and DPU Setup – The callback proceeds to create the Completion Queue (CQ) resources and sets up the DPA thread on the DPU. The DPA thread is equipped with two message queues: one for sending and one for receiving.

- Binding CQ Doorbell to DB Completion – An RPC is sent to bind the CQ doorbell to the DB completion context. This is done while the DPA is not yet active, preventing synchronization issues.

- SQ Resource Creation – The Submission Queue (SQ) resources are created, including the SQE pools and the local buffer size needed for copying SQEs from the host to the DPU. The DOCA DMA is used for the data transfer operations.

- SQ Bind DB Message – The SQ sends a "bind DB" message to the DPA.

- DPA Receives Bind DB Message – The DPA processes the "bind DB" message and sends the SQ's doorbell information to the DB completion context.

- SQ Sends Bind Done Message – The SQ sends a "bind done" message to the DPU.

- Start SQ DB – The DPU receives the "bind done" message and starts the SQ DB.

- NVMe-oF QPair and Request Pool Creation – An NVMe-oF QPair and an NVMe-oF request pool are created.

- Asynchronous QPair Creation – The NVMe-oF library starts an asynchronous operation to create the QPair.

- NVMe-oF Library Calls –

    - The library creates the QPair and calls the transport to get the optimal poll group.

    - It then calls `poll_group_add` on the selected thread.

- NVMe-oF Connect Request – The transport sends an NVMe-oF connect request.

- Set Property Request – After the connect request is complete, a `set_property` request is sent to update the controller configuration as provided by the host during initialization.
- Callback Triggered – Once the `set_property` request is finished, the NVMe-oF triggers callbacks.
- Update Stateful Region – The transport updates the stateful region, setting the `CSTS.RDY` bit to 1.
- Host Polling Unblocked – With the CSTS.RDY set to 1, the host polling is now unblocked, completing the initialization process.

Reset and Shutdown Flow

The reset flow in NVMe using SPDK is crucial for maintaining the integrity and stability of the storage subsystem, to allow the system afterwards to recover gracefully.

The reset process can be initiated by:

- The Host: The host can initiate a shutdown or reset of the NVMe controller by configuring specific registers in the controller's register space. in this case the `handle_controller_register_events()` function is triggered.
  - Configuring Shutdown Notification (SHN): The host can write to the CC (Controller Configuration) register, specifically the SHN field, to specify how the controller should handle a shutdown:
    - Normal Shutdown – `SPDK_NVME_SHN_NORMAL` – Allows for a graceful shutdown where the controller can complete outstanding commands.
    - Abrupt Shutdown – `SPDK_NVME_SHN_ABRUPT` – Forces an immediate shutdown without completing outstanding commands.
  - Resetting the NVMe controller by setting the CC.enablebit to zero. Also `handle_controller_register_events()` function is triggered in this case.
  - The host can initiate a reset the NVMe controller by using Function-Level Reset (FLR). In this case the `flr_event_handler_cb()` is triggered.

- DOCA transport can initiate reset flow through `nvmf_doca_on_initialization_error()` when it detects an internal error condition during initialization flow.

Once a shutdown or reset is requested, the transport proceeds to destroy all resources associated with the controller. If it is a shutdown request, the shutdown status is updated accordingly. When the host performs a function-level reset (FLR) or a controller reset, the transport must take several actions: it destroys all submission and completion queues (SQs and CQs) across all poll groups for the specified device, destroys the admin SQ and CQ via the admin thread, stops the PCIe device, and then restart it.

Regardless of the reason for the reset flow, it all starts from `nvmf_doca_pci_dev_admin_reset()`. This function marks the beginning of the asynchronous process for resetting the PCIe device NVMe-oF context. The flow consists of callbacks that are triggered in sequence to track the completion of each process and to proceed to the next one. Let us now illustrate the rest of the flow:

- If the admin QP exists, the process first checks for any I/O submission queues (SQs).
- If I/O SQs are found, an asynchronous flow begins to stop all I/O SQs.
- For each I/O SQ associated with the admin queue, it retrieves the corresponding poll group responsible for destroying its specific SQ, as no other poll group can perform this action.
- Once all I/O SQs are stopped, if any I/O completion queues (CQs) remain, a message is sent to each poll group instructing them to delete their I/O CQs.
- After all I/O queues are destroyed, the flow proceeds to destroy the admin CQ and SQ.

After this flow is done, it moves to `nvmf_doca_pci_dev_admin_reset_continue()` to finalize the reset flow:

- If a reset is issued by configuring the NVMe controller registers, then set `CSTS.BITS.SHST` to `SPDK_NVME_SHST_COMPLETE` and `CSTS.BITS.RDY` to 0
- If the reset is triggered by a FLR, then stop the PCIe device context: `doca_ctx_stop(doca_devemu_pci_dev_as_ctx())`

I/O QP Create/Destroy

The process of creating and destroying I/O Queue Pairs (QPairs) begins with the initiator (host) sending an NVMe-oF connect command to the NVMe-oF target following the completion of transport initialization. The host sends an NVMe-oF connect command to the NVMe-oF target after the transport has completed its initialization. The NVMe-oF target receives an admin command and begins processing it through `nvmf_doca_on_fetch_sqe_complete()`. Based on the command opcode, the following steps are executed:

1. Creating an I/O Completion Queue – `SPDK_NVME_OPC_CREATE_IO_CQ` – `handle_create_io_cq()`:
    1. First, the system selects a poll group that is responsible for creating the CQ.
    2. The target searches for the `nvmf_doca_pci_dev_poll_group` entity within the selected poll group. If it is not found, it indicates that this is the first queue associated with the specific device managed by this poll group, necessitating the creation of a new entity.
    3. Next, the target allocates a DOCA IO (`nvmf_doca_io`) using the attributes and data provided in the command, such as CQ size, CQ ID, CQ address, and CQ MSI-X.
    4. Create the DMA context, the message queues, consumer and producer handle and the DPA thread.
    5. Once the asynchronous allocation and setup are complete, the target posts a Completion Queue Entry (CQE) to indicate that the operation has succeeded. it is done through static void `nvmf_doca_poll_group_create_io_cq_done()`.
2. Creating an I/O Submission Queue – `SPDK_NVME_OPC_CREATE_IO_SQ` – `handle_create_io_sq()`:
    1. Based on the CQ ID argument provided in the command, the system first identifies the I/O entity to which the new Submission Queue (SQ) should be added. From this entity, it retrieves the associated `nvmf_doca_pci_dev_poll_group`.
    2. Next, a DOCA Submission Queue `nvmf_doca_sq` is allocated using the attributes and data specified in the command, including the SQ size, SQ ID, and SQ address. This allocation is handled within the `nvmf_doca_poll_group_create_io_sq()` function.

3. The DMA context and the DB completions are also created.

4. Once the asynchronous process of creating the SQ is complete, a Completion Queue Entry (CQE) is posted via the `nvmf_doca_poll_group_create_io_sq_done()` function.

5. Next, the target begins the QPair allocation, which creates a new Queue Pair comprising a submission queue (SQ) and a completion queue (CQ).

6. Then, it determines the optimal poll group for the new QPair by calling `get_optimall_poll()` group ensuring that both the CQ and SQ attached to it runs on the same poll group.

7. After identifying the appropriate poll group, the target adds the newly created QPair to it using `nvmf_doca_poll_group_add()`, enabling management of the QPair's events and I/O operations.

8. After the connection is established, the initiator can start sending I/O commands through the newly created QPair.

3. Destroying an I/O Completion Queue – `SPDK_NVME_OPC_DELETE_IO_CQ` – `handle_delete_io_cq()`:

    1. The process starts by fetching the identifier of the queue that needs to be deleted from the NVMe request.

    2. Once the identifier is retrieved, the corresponding `nvmf_doca_io` entity is located.

    3. The associated poll group is then extracted from the I/O, as it is responsible for destroying the CQ associated with it. The thread retrieved schedules the execution of `nvmf_doca_pci_dev_poll_group_stop_io_cq()` using `spdk_thread_send_msg`.

    4. The `nvmf_doca_io_stop()` function is called to initiate the stopping process. If there are CQs in this I/O that are not idle, it triggers `nvmf_doca_io_stop_continue()` to advance the sequence. This flow then executes a series of asynchronous callback functions in order, ensuring that each step completes fully before the next begins, performing the following actions:

1. Stop the DOCA devemu PCIe device doorbell to prevent triggering completions on the associated doorbell completion context
2. Stop the NVMe-oF DOCA DMA pool
3. Stop the DMA context associated with the Completion Queue and frees all elements of the NVMe-oF DOCA Completion Queue.
4. Stop the NVMe-oF DOCA DPA communication channel, halting both the receive and send message queues.
5. Stop and destroy the PCIe device DB.
6. Stop the MSI-X.
7. Stop and destroy the DB completion.
8. Destroy the communication channel.
9. Destroy the DPA thread.

5. Finally, the target posts a Completion Queue Entry (CQE) to indicate that the operation has succeeded.

4. Destroying an I/O Submission Queue – `SPDK_NVME_OPC_DELETE_IO_SQ` – `handle_delete_io_sq()`:

    1. The process starts by fetching the identifier of the queue that needs to be deleted from the NVMe request.
    2. Once the identifier is retrieved, the corresponding `nvmf_doca_sq` entity is located.
    3. The associated poll group is then extracted from the I/O, as it is responsible for destroying the SQ associated with it. The thread retrieved schedules the execution of `nvmf_doca_pci_dev_poll_group_stop_io_sq()` using `spdk_thread_send_msg`.
    4. The `nvmf_doca_sq_stop()` function is called to initiate the stopping process.
    5. The stopping process begins by calling `spdk_nvmf_qpair_disconnect()` To disconnect an NVMe-oF queue pair (QP), clean up associated resources and

terminate the connection.

6. Once this step is complete, `nvmf_doca_sq_stop_continue()` is triggered to proceed with a sequence of asynchronous callback functions, ensuring that each step completes before moving to the next, performing the following actions:

    1. Disconnect an NVMe-oF queue pair (QP), cleaning up associated resources and terminate the connection.

    2. Stop the DOCA devemu PCIe device doorbell to prevent triggering completions on the associated doorbell completion context.

    3. Send unbind SQ doorbell message to DPA.

    4. Stop the NVMe-oF DOCA DMA pool

    5. Stop the DMA context associated with the Submission Queue and frees all elements of the NVMe-oF DOCA Submission Queue.

    6. Destroy the resources associated with the SQ: the DMA pool, the queue and the request pool.

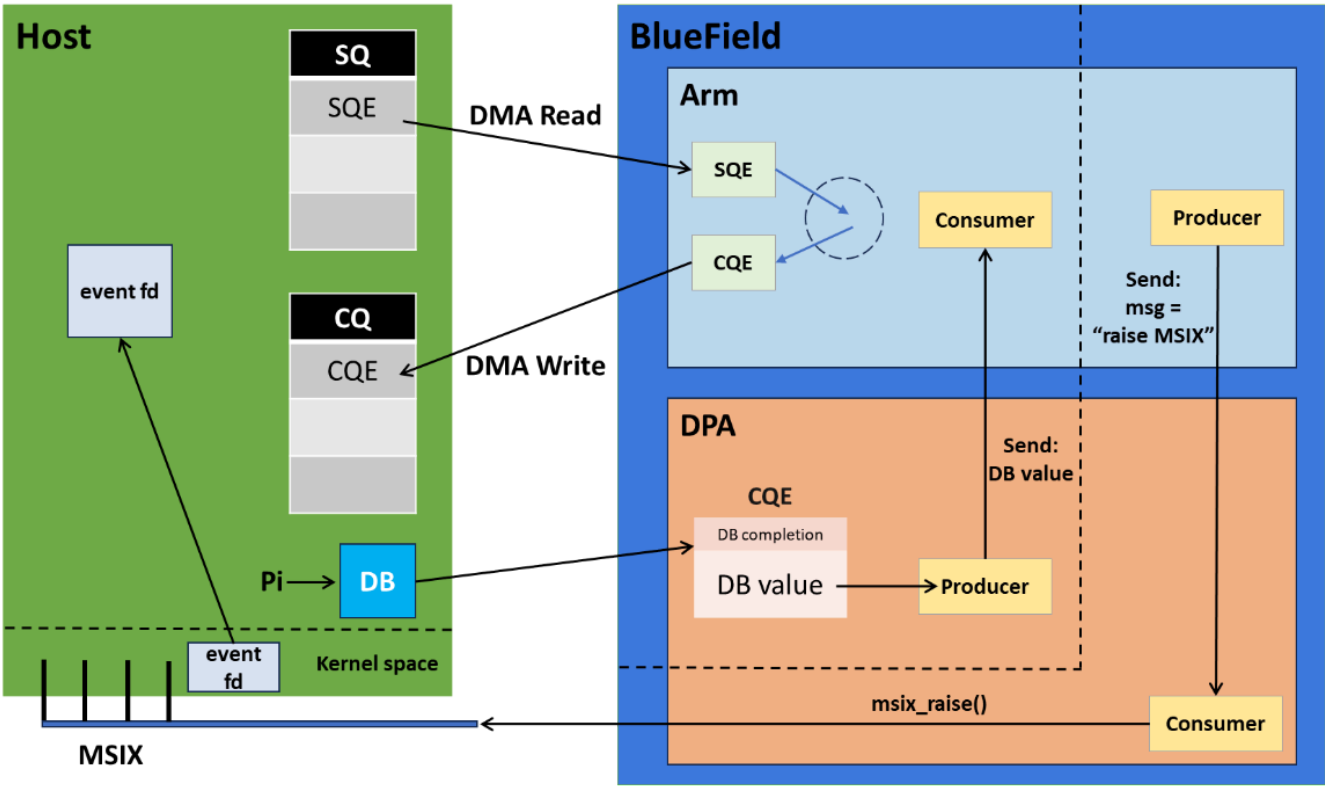
    7. Finally, the target posts a Submission Queue Entry (CQE) to indicate that the operation has succeeded.

Data Path Flows

From the host's perspective, it is communicating with a standard NVMe device. To create this experience, the DOCA transport uses the available NVMe-oF APIs to mimic the behavior of a real NVMe device effectively.

The data path flow involves a series of steps that handle the transfer and processing of commands between the host and the NVMe-oF target. It begins with the host writing commands to the Submission Queue (SQ) entries, specifying operations like read or write requests that the NVMe-oF target processes.

The diagram below provides a holistic view and data path steps:



1. The host writes the Submission Queue Entry (SQE).
2. The host rings the doorbell (DB), and the DB value is received by the DPA.
3. The producer forwards the DB value to the ARM processor.
4. The system reads the SQE from the host.
5. The SQE is processed.
6. A Completion Queue Entry (CQE) is written back.
7. If MSI-X is enabled, the producer triggers the MSI-X interrupt.
8. The DPU's consumer raises the MSI-X interrupt to the host.

Each step is described in depth below:

Retrieve Doorbell

The process starts by retrieving the doorbell values, which indicate new commands submitted by the host, allowing the system to identify and process pending commands:

The DPA wakes up and checks the reason for activation—it could either be that the DPU has posted something for the DPA consumer, or a new doorbell value needs to be passed to the DPU. In this case, the DPA detects a new doorbell value and sends it to the DPU via message queues. The DPU then calculates the number of Submission Queue Entries (SQEs) that need to be fetched from the host's submission queue and retrieves the commands using DMA via the `nvmf_doca_sq_update_pi()` function.

Fetch SQE From Host

After retrieving the Submission Queue Entry (SQE), the system must translate this command into a request object that the NVMe-oF target can understand and process. The SQE contains crucial command details that need to be executed, such as read or write operations. This process involves populating a `spdk_nvmf_request` structure, which includes:

- Command parameters extracted from the SQE.
- Associated data buffer locations (if any data is to be read from or written to).
- Metadata and additional information necessary for processing the command.

For admin commands, the SQEs are handled by the `nvmf_doca_on_fetch_sqe_complete()` function, while I/O NVMe-oF commands are managed by the `nvmf_doca_on_fetch_nvm_sqe_complete()` function. Both functions are responsible for filling the `nvmf_doca_request` structure.

A request is obtained from the SQ request pool, as described previously, and it is populated by setting various attributes based on the specific command being issued. These attributes may include the Namespace ID, the length of the request, and the queue to which the request belongs.

Next, there are three options for data direction handled in the command:

1. No data transfer:
    1. After preparing the request, the system sets the callback function to `post_cqe_from_response()`
    2. It then executes the request using `spdk_nvmf_request_exec()`
    3. Finally, the system posts the CQE to indicate completion

2. Data transfer from host to DPU:

    1. After preparing the request, the system retrieves buffers from the SQ pool and initializes them with the details of the data to be copied from the host

    2. It invokes `nvmf_doca_sq_copy_data()`, which performs a DMA copy from the host to the DPU

    3. Once the asynchronous copy completes, `spdk_nvmf_request_exec()` is called to continue processing

    4. Finally, the system posts the CQE to signal completion

3. Data transfer from DPU to host:

    1. After preparing the request, the system retrieves buffers from the SQ pool and initializes them with the data to be copied and the destination address on the host.

    2. It invokes `nvmf_doca_sq_copy_data()` to perform a DMA copy from the DPU to the host

    3. Once the asynchronous copy finishes, `spdk_nvmf_request_exec()` is called to complete processing

    4. The system then posts the CQE to indicate the operation's completion

While the overall flow for NVMe commands and admin commands is similar, there are subtle differences in the transport implementation to address the unique requirements of each command type. For I/O commands like read and write, the system may involve large data blocks transfer. Here the PRPs (Physical Region Pages) come into play, as they are used to describe the memory locations for the data to be read or written. The PRPs provide a list of physical addresses that the NVMe device uses to access the host's memory directly.

In this scenario, there could be multiple DMA operations required for copying the data. Therefore, after preparing the request, the function `nvme_cmd_map_prps()` is invoked to iterate over the entire PRP list, preparing the retrieved buffers from the pool and initializing them with the corresponding data and destination addresses. Once the buffers are properly set up, the function `buffer_ready_copy_data_host_to_dpu()` is called, which iterates through all the buffers and invokes `nvmf_doca_sq_copy_data()` for each one. Only after all asynchronous copy tasks for the buffers are completed does the

function `nvmf_doca_request_complete()` get called to signal the end of the request processing.

Dispatch Command to NVMe-oF Target

Once the request is built, the function `spdk_nvmf_request_exec()` is called to execute it. `spdk_nvmf_request_exec()` initiates the processing of the request by determining the command type and dispatching it to the appropriate handler within the NVMe-oF target for processing. The NVMe-oF target interprets the command, allocates necessary resources, and performs the requested operation.

When a request is dispatched to the NVMe-oF target for execution via `spdk_nvmf_request_exec()`, a completion callback function is usually configured. This callback is invoked once the request has been fully processed, indicating to the transport layer that the NVMe-oF target has completed handling the request.

Post CQE to Host

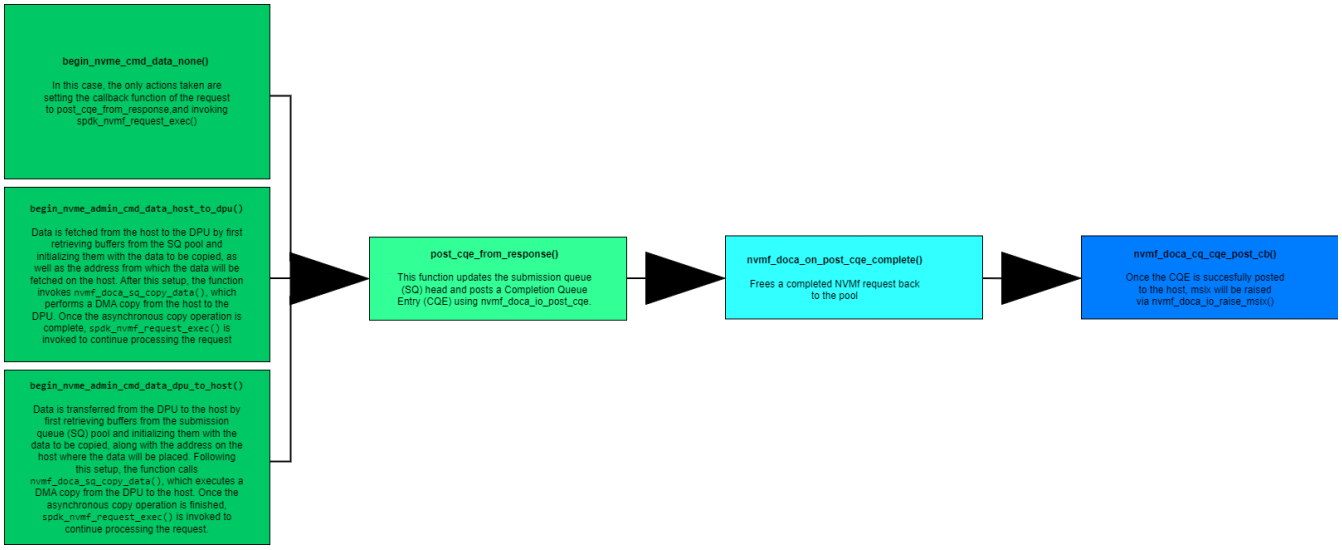
After the command has been processed, we create a Completion Queue Entry (CQE) and we posted back to the host to indicate the completion status of the command. This entry includes details about the operation's success or any errors encountered.

Raise RMSI-X

If the DMA for posting the CQE is completed successfully, MSIX is raised to the host, to inform the host that a completion event has occurred allowing to it read the CQE and process the result of the request.

In this scenario, the DPU calls `nvmf_doca_io_raise_msix()`, which in turn sends a message through `nvmf_doca_dpa_msgq_send()`. This action prompts the DPA to wake up and attempt to retrieve the consumer completion context. Then, DPA receives a message from the DPU instructing it to raise MSIX.

The flow diagram illustrates the steps from fetching SQEs and preparing the request to posting the CQE, highlighting the three possible data scenarios (case where PRP is not involved):



Limitations

Supported SPDK Versions

The supported SPDK version is 23.01.

Supported Admin Commands

Currently, not all admin commands are supported. The transport supports the following commands: `SPDK_NVME_OPC_CREATE_IO_CQ`, `SPDK_NVME_OPC_DELETE_IO_CQ`, `SPDK_NVME_OPC_CREATE_IO_SQ`, `SPDK_NVME_OPC_DELETE_IO_SQ`, `SPDK_NVME_OPC_ASYNC_EVENT_REQUEST`, `SPDK_NVME_OPC_IDENTIFY`, `SPDK_NVME_OPC_GET_LOG_PAGE`, `SPDK_NVME_OPC_GET_FEATURES`, `SPDK_NVME_OPC_SET_FEATURES`.

Supported NVM Commands

Currently, the transport supports only the following NVMe commands: `SPDK_NVME_OPC_FLUSH`, `SPDK_NVME_OPC_WRITE`, `SPDK_NVME_OPC_READ`.

SPDK Stop Listen Flow

The stop listener flow (`spdk_nvmf_stop_listen`) can be initiated through the `remove_listener` RPC. The current version of SPDK has limitations regarding the asynchronous handling of `remove_listener` requests. For this reason, it is recommended that the stop listener function be called only after freeing up resources like memory buffers and queue pairs. This can be accomplished on the host side by issuing the unbind script:

```
python3 samples/doca_devemu/devemu_pci_vfio_bind.py --unbind
0000:62:00.0
```

DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA Compress
- DOCA Comch

For additional information about the used DOCA libraries, please refer to the respective programming guides.

Dependencies

- BlueField-3 DPU is required
- SPDK version 23.0 is required

Compiling the Application

Info

Please refer to the DOCA Installation Guide for Linux for details on how to install BlueField-related software.

The installation of DOCA's reference application contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

Tip

For more information about the applications as well as development and compilation tips, refer to the DOCA Reference Applications page.

The sources of the application can be found under the application's directory: `/opt/mellanox/doca/applications/nvme_emulation/.`

Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

1. To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

   `doca_nvme_emulation` is created under `/tmp/build/applications/nvme_emulation/`

2. Alternatively, one can set the desired flags in the `meson_options.txt` file instead of providing them in the compilation command line:

1. Edit the following flags in `/opt/mellanox/doca/applications/meson_options.txt`:
   - Set `enable_all_applications` to `false`
   - Set `enable_nvme_emulation` to `true`
2. The same compilation commands should be used, as were shown in the previous section:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```

Compiling With Custom SPDK

If you plan to use a custom or alternative SPDK version, update the paths in the following variables via Meson:

- `spdk_lib_path`
- `spdk_incl_path`
- `spdk_dpdk_lib_path`
- `spdk_isal_prefix`

Troubleshooting

Please refer to the NVIDIA BlueField Platform Software Troubleshooting Guide for any issue you may encounter with the compilation of the DOCA applications.

Running the Application

Prerequisites

From the server on the DPU:

The user is required to allocate hugepages, and then run the application that runs continuously, remaining active and processing incoming RPC requests.

```
$ echo 1024 > /sys/kernel/mm/hugepages/hugepages-
2048kB/nr_hugepages $ sudo mount -t hugetlbfs -o pagesize=2M
nodev /mnt/huge
$ sudo
/tmp/ariej_build/applications/nvme_emulation/doca_nvme_emulation
```

From the client on the DPU:

The user can send various RPC requests to the application during its execution. For example, to remove a listener, the user could send the following command:

```
$ sudo PYTHONPATH=/doca/applications/nvme_emulation/
/usr/bin/spdk_rpc.py nvmf_subsystem_remove_listener nqn.2016-
06.io.spdk:cnode1 -t doca -a MT2306XZ00AYGES1D0F0
```

Application Execution

The NvMR emulation application is provided in source form, hence a compilation is required before the application can be executed.

1. Application usage instructions:

```
Usage: doca_nvme_application [DOCA Flags] [Program Flags]
```

```
DOCA Flags:
  -h, --help                        Print a help synopsis
  -v, --version                     Print program version
information
  -l, --log-level                   Set the (numeric) log
level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  --sdk-log-level                   Set the SDK (numeric) log
level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR,
40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
  -j, --json <path>                 Parse all command flags
from an input json file

Program Flags:
  -p, --pci-addr                    DOCA Comm Channel device
PCIe address
  -r, --rep-pci                     DOCA Comm Channel device
representor PCIe address
  -f, --file                        File to send by the
client / File to write by the server
  -t, --timeout                     Application timeout for
receiving file content messages, default is 5 sec

 -c, --config <config>     JSON config file (default none)
     --json <config>       JSON config file (default none)
     --json-ignore-init-errors
                           don't exit on invalid config entry
 -d, --limit-coredump      do not set max coredump size to
RLIM_INFINITY
 -g, --single-file-segments
                           force creating just one hugetlbfs
file
 -h, --help                show this usage
 -i, --shm-id <id>         shared memory ID (optional)
 -m, --cpumask <mask or list>    core mask (like 0xF) or core
list of '[]' embraced (like [0,1,10]) for DPDK
```

```
 -n, --mem-channels <num>  channel number of memory channels
used for DPDK
 -p, --main-core <id>      main (primary) core for DPDK
 -r, --rpc-socket <path>   RPC listen address (default
/var/tmp/spdk.sock)
 -s, --mem-size <size>     memory size in MB for DPDK (default:
0MB)
     --disable-cpumask-locks    Disable CPU core lock files.
     --silence-noticelog   disable notice level logging to
stderr
     --msg-mempool-size <size>  global message memory pool
size in count (default: 262143)
 -u, --no-pci              disable PCIe access
     --wait-for-rpc        wait for RPCs to initialize
subsystems
     --max-delay <num>     maximum reactor delay (in
microseconds)
 -B, --pci-blocked <bdf>
                           PCIe addr to block (can be used
more than once)
 -R, --huge-unlink         unlink huge files after
initialization
 -v, --version             print SPDK version
 -A, --pci-allowed <bdf>
                           PCIe addr to allow (-B and -A
cannot be used at the same time)
     --huge-dir <path>     use a specific hugetlbfs mount to
reserve memory from
     --iova-mode <pa/va>   set IOVA mode ('pa' for IOVA_PA and
'va' for IOVA_VA)
     --base-virtaddr <addr>      the base virtual address for
DPDK (default: 0x200000000000)
     --num-trace-entries <num>   number of trace entries for
each core, must be power of 2, setting 0 to disable trace
(default 32768)
```

```
     --rpcs-allowed        comma-separated list of permitted
RPCS
     --env-context         Opaque context for use of the env
implementation
     --vfio-vf-token       VF token (UUID) shared between SR-
IOV PF and VFs for vfio_pci driver
 -L, --logflag <flag>    enable log flag (all, accel, aio,
app_config, app_rpc, bdev, bdev_concat, bdev_ftl, bdev_group,
bdev_malloc, bdev_null, bdev_nvme, bdev_raid, bdev_raid0,
bdev_raid1, bdev_raid5f, blob, blob_esnap, blob_rw, blobfs,
blobfs_bdev, blobfs_bdev_rpc, blobfs_rw, ftl_core, ftl_init,
gpt_parse, json_util, log, log_rpc, lvol, lvol_rpc,
notify_rpc, nvme, nvme_vfio, nvmf, nvmf_tcp, opal, rdma,
reactor, rpc, rpc_client, sock, sock_posix, thread, trace,
uring, vbdev_delay, vbdev_gpt, vbdev_lvol, vbdev_opal,
vbdev_passthru, vbdev_split, vbdev_zone_block, vfio_pci,
vfio_user, virtio, virtio_blk, virtio_dev, virtio_pci,
virtio_user, virtio_vfio_user, vmd)
 -e, --tpoint-group <group-name>[:<tpoint_mask>]
                           group_name - tracepoint group name
for spdk trace buffers (bdev, nvmf_rdma, nvmf_tcp, blobfs,
thread, nvme_pcie, nvme_tcp, bdev_nvme, nvme_nvda_tcp, all)
                           tpoint_mask - tracepoint mask for
enabling individual tpoints inside a tracepoint group. First
tpoint inside a group can be enabled by setting tpoint_mask
to 1 (e.g. bdev:0x1).
                            Groups and masks can be combined
(e.g. thread,bdev:0x1).
                            All available tpoints can be
found in /include/spdk_internal/trace_defs.h
```

Note

The above usage printout can be printed to the command line using the `-h` (or `--help`) options:

```
/tmp/build/applications/nvme_emulation/doca_nvme_emulatic
-h
```

Command Line Flags

The application uses the same command-line flags as SPDK, allowing for configuration and behavior control similar to standard SPDK applications.

For more details refer to SPDK's official Storage Performance Development Kit Documentation.

Troubleshooting

Please refer to the DOCA Troubleshooting for any issue you may encounter with the installation or execution of the DOCA applications.

References

- `/opt/mellanox/doca/applications/nvme_emulation/`
- `/opt/mellanox/doca/applications/nvme_emulation/build_device_code.`
- `/opt/mellanox/doca/applications/nvme_emulation/dependencies`
- `/opt/mellanox/doca/applications/nvme_emulation/device`
- `/opt/mellanox/doca/applications/nvme_emulation/host`
- `/opt/mellanox/doca/applications/nvme_emulation/meson.build`

- `/opt/mellanox/doca/applications/nvme_emulation/nvme_emulation.c`
- `/opt/mellanox/doca/applications/nvme_emulation/rpc_nvmf_doca.py`